

UEL: Unification Solver for \mathcal{EL}

Franz Baader, Stefan Borgwardt, Julian Mendez, and Barbara Morawska*
{baader, stefborg, mendez, morawska}@tcs.inf.tu-dresden.de

Theoretical Computer Science, TU Dresden, Germany

Abstract. UEL is a system that computes unifiers for unification problems formulated in the description logic \mathcal{EL} . \mathcal{EL} is a description logic with restricted expressivity, but which is still expressive enough for the formal representation of biomedical ontologies, such as the large medical ontology SNOMED CT. We propose to use UEL as a tool to detect redundancies in such ontologies by computing unifiers of two formal concepts suspected of expressing the same concept of the application domain. UEL provides access to two different unification algorithms and can be used as a plug-in of the popular ontology editor Protégé, or stand-alone.

1 Motivation

The description logic (DL) \mathcal{EL} , which offers the concept constructors conjunction (\sqcap), existential restriction ($\exists r.C$), and the top concept (\top), has recently drawn considerable attention since, on the one hand, important inference problems such as the subsumption problem are polynomial in \mathcal{EL} [1,10,4]. On the other hand, though quite inexpressive, \mathcal{EL} can be used to define biomedical ontologies, such as the large medical ontology SNOMED CT.¹

Unification in DLs has been proposed in [9] as a novel inference service that can, for instance, be used to detect redundancies in ontologies. For example, assume that one developer of a medical ontology defines the concept of a *patient with severe head injury* as

$$\text{Patient} \sqcap \exists \text{finding} . (\text{Head_injury} \sqcap \exists \text{severity} . \text{Severe}), \quad (1)$$

whereas another one represents it as

$$\text{Patient} \sqcap \exists \text{finding} . (\text{Severe_injury} \sqcap \exists \text{finding_site} . \text{Head}). \quad (2)$$

These two concept descriptions are not equivalent, but they are nevertheless meant to represent the same concept. They can obviously be made equivalent by treating the concept names `Head_injury` and `Severe_injury` as variables, and substituting the first one by `Injury` \sqcap $\exists \text{finding_site} . \text{Head}$ and the second one by `Injury` \sqcap $\exists \text{severity} . \text{Severe}$. In this case, we say that the descriptions are unifiable, and call the substitution that makes them equivalent a *unifier*. Intuitively, such

* Supported by DFG under grant BA 1122/14-1

¹ see <http://www.ihtsdo.org/snomed-ct/>

Name	Syntax	Semantics
concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role name	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
top	\top	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{x \mid \exists y : (x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
concept definition	$A \equiv C$	$A^{\mathcal{I}} = C^{\mathcal{I}}$

Table 1. Syntax and semantics of \mathcal{EL} .

a unifier proposes definitions for the concept names that are used as variables: in our example, we know that, if we define `Head_injury` as `Injury` \sqcap `finding_site.Head` and `Severe_injury` as `Injury` \sqcap `severity.Severe`, then the two concept descriptions (1) and (2) are equivalent w.r.t. these definitions. Of course, this example was constructed such that the unifier actually provides sensible definitions for the concept names used as variables. In general, the existence of a unifier only says that there is a structural similarity between the two concepts. The developer that uses unification as a tool for finding redundancies in an ontology or between two different ontologies needs to inspect the unifier(s) to see whether the suggested definitions really make sense.

In [6] it was shown that unification in \mathcal{EL} is an NP-complete problem. Basically, this problem is in NP since every solvable unification problem has a “local” unifier, i.e., one built from parts of the unification problem. The NP algorithm introduced in [6] is a brutal “guess and then test” algorithm, which guesses a local substitution and then checks whether it is a unifier. In [8], a more practical rule-based \mathcal{EL} -unification algorithm was introduced, which tries to transform the given unification problems into a solved form, and makes nondeterministic decisions only if triggered by the problem. Finally, the paper [7] proposes a third algorithm, which encodes the unification problem into a set of propositional clauses and then solves it using an existing highly optimized SAT solver.

Version 1.0.0 of our system UEL² used only the SAT translation to solve unification problems. This approach allowed us to get a fast unification algorithm—the unification problem only has to be translated into a propositional formula and the SAT solver is used to actually solve the problem. In contrast, for the implementation of the rule-based algorithm from [8] we had to find efficient methods to deal with the nondeterminism ourselves. As of version 1.2.0, UEL includes both implementations, as well as a variant of the SAT translation that tries to compute only small unifiers.

2 \mathcal{EL} and Unification in \mathcal{EL}

In order to explain what the algorithms implemented in UEL actually compute, we need to recall the relevant definitions and results for unification in \mathcal{EL} .

² All versions of this system are available at <http://uel.sourceforge.net>.

Starting with a finite set N_C of *concept names* and a finite set N_R of *role names*, \mathcal{EL} -*concept descriptions* are built from concept names using the constructors *conjunction* ($C \sqcap D$), *existential restriction* ($\exists r.C$ for every $r \in N_R$), and *top* (\top). On the semantic side, concept descriptions are interpreted as sets. To be more precise, an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that maps concept names to subsets of $\Delta^{\mathcal{I}}$ and role names to binary relations over $\Delta^{\mathcal{I}}$. This function is extended to concept descriptions as shown in the semantics column of Table 1.

A *concept definition* is of the form $A \equiv C$ for a concept name A and a concept description C . A *TBox* \mathcal{T} is a finite set of concept definitions such that no concept name occurs more than once on the left-hand side of a definition in \mathcal{T} . The TBox \mathcal{T} is called *acyclic* if there are no cyclic dependencies between its concept definitions. Given a TBox \mathcal{T} , we call a concept name A a *defined concept* if it occurs as the left-side of a concept definition $A \equiv C$ in \mathcal{T} . All other concept names are called *primitive concepts*. An interpretation \mathcal{I} is a *model* of a TBox \mathcal{T} if $A^{\mathcal{I}} = C^{\mathcal{I}}$ holds for all definitions $A \equiv C$ in \mathcal{T} .

Subsumption asks whether a given concept description C is a subconcept of another concept description D : C is *subsumed* by D w.r.t. \mathcal{T} ($C \sqsubseteq_{\mathcal{T}} D$) if every model \mathcal{I} of \mathcal{T} satisfies $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. We say that C is *equivalent* to D w.r.t. \mathcal{T} ($C \equiv_{\mathcal{T}} D$) if $C \sqsubseteq_{\mathcal{T}} D$ and $D \sqsubseteq_{\mathcal{T}} C$. For the empty TBox, we write $C \sqsubseteq D$ and $C \equiv D$ instead of $C \sqsubseteq_{\emptyset} D$ and $C \equiv_{\emptyset} D$, and simply talk about subsumption and equivalence (without saying “w.r.t. \emptyset ”).

In order to define unification, we partition the set N_C of concept names into a set N_v of concept variables (which may be replaced by substitutions) and a set N_c of concept constants (which must not be replaced by substitutions). Intuitively, N_v are the concept names that have possibly been given another name or been specified in more detail in another concept description describing the same notion. A *substitution* σ maps every variable to a concept description. It can be extended to concept descriptions in the usual way.

Unification in \mathcal{EL} was first considered w.r.t. the empty TBox [6]. In this setting, an \mathcal{EL} -*unification problem* is a finite set $\Gamma = \{C_1 \equiv? D_1, \dots, C_n \equiv? D_n\}$ of equations. A substitution σ is a *unifier* of Γ if σ *solves* all the equations in Γ , i.e., if $\sigma(C_1) \equiv \sigma(D_1), \dots, \sigma(C_n) \equiv \sigma(D_n)$. We say that Γ is *solvable* if it has a unifier. Without loss of generality, we can assume that the unification problem is *flat*, i.e., that all concept descriptions occurring in it are conjunctions of concept names or existential restrictions of the form $\exists r.A$ with $A \in N_C$. If a unification problem is not flat, we can transform it into a flat unification problem by introducing auxiliary variables.

As mentioned before, the main reason for solvability of unification in \mathcal{EL} to be in NP is that any solvable unification problem has a local unifier. Basically, any unification problem Γ determines a polynomial number of so-called *non-variable atoms*, which are concept constants or existential restrictions of the form $\exists r.A$ for a role name r and a concept constant or variable A . An *assignment* S maps every concept variable X to a subset S_X of the set At_{nv} of non-variable atoms of Γ . Such an assignment induces a relation $>_S$ on N_v , which is the transitive

closure of $\{(X, Y) \in N_v \times N_v \mid Y \text{ occurs in an element of } S_X\}$. We call the assignment S *acyclic* if $>_S$ is irreflexive (and thus a strict partial order). Any acyclic assignment S induces a unique substitution σ_S , which can be defined by induction along $>_S$:

- If $X \in N_v$ is minimal w.r.t. $>_S$, then we define $\sigma_S(X) := \prod_{D \in S_X} D$.
- Assume that $\sigma_S(Y)$ is already defined for all Y such that $X >_S Y$. Then we define $\sigma_S(X) := \prod_{D \in S_X} \sigma_S(D)$.

We call a substitution σ *local* if it is of this form, i.e., if there is an acyclic assignment S such that $\sigma = \sigma_S$. Consequently, one can enumerate (or guess, in a nondeterministic machine) all acyclic assignments and then check whether any of them induces a substitution that is a unifier. Using this brute-force approach, in general many local substitutions will be generated that only in the subsequent check turn out not to be unifiers.

The fact that any solvable unification problem has a local unifier was shown in [6] by proving that such a problem always has a *minimal* unifier and that every minimal unifier is equivalent to a local unifier. Minimality and equivalence of unifiers are determined by the following order \succeq on substitutions. For two substitutions σ and θ , we define $\sigma \succeq \theta$ iff $\sigma(X) \sqsubseteq \theta(X)$ holds for all variables X . We say that a unifier σ of a unification problem Γ is *minimal* if there is no unifier γ of Γ such that $\sigma \succeq \gamma$ and $\gamma \not\preceq \sigma$. Two unifiers σ, θ are *equivalent* iff $\sigma \succeq \gamma$ and $\gamma \succeq \sigma$. We introduce a similar order \succeq on assignments and write $S \succeq S'$ iff $S_X \supseteq S'_X$ holds for all $X \in N_v$. We call an assignment S *minimal* if σ_S is a unifier of Γ and there is no assignment S' different from S such that $\sigma_{S'}$ is a unifier of Γ and $S \succeq S'$. The following is easy to show: if $S \succeq S'$, then $\sigma_S \succeq \sigma_{S'}$. As shown in [3], this implies that all minimal unifiers are induced by minimal assignments, but the opposite need not hold. Computing only the minimal assignments is thus a first step towards computing only minimal unifiers. Computing only minimal unifiers is desirable since they are those among the local unifiers that substitute the variables by smaller concept descriptions and their corresponding assignments do not contain “irrelevant” non-variable atoms.

In [8], *unification w.r.t. an acyclic TBox* \mathcal{T} was introduced. In this setting, the concept variables are a subset of the primitive concepts of \mathcal{T} , and substitutions are applied both to the concept descriptions in the unification problem and to the right-hand sides of the definitions in \mathcal{T} . To deal with such unification problems, one does not need to develop a new algorithm. In fact, by viewing the defined concepts of \mathcal{T} as variables, one can turn \mathcal{T} into a unification problem, which one simply adds to the given unification problem Γ . As shown in [8], there is a 1–1-correspondence between the unifiers of Γ w.r.t. \mathcal{T} and the unifiers of this extended unification problem.

We will now describe the two unification algorithms implemented in UEL 1.2.0. Both algorithms have in common that they generate acyclic assignments, and thus output only local unifiers. They follow two different approaches to reduce the amount of blind guessing of the brute-force approach.

The SAT Translation

Instead of blindly generating all local substitutions, the reduction to the propositional satisfiability problem introduced in [7] ensures that only assignments that induce unifiers are generated. The set of propositional clauses $C(\Gamma)$ generated by the reduction contains two kinds of propositional letters: $[A \sqsubseteq B]$ for $A, B \in \text{At}_{\text{nv}}$ ³ and $[X > Y]$ for concept variables X, Y . Intuitively, setting $[A \sqsubseteq B] = 1$ means that the local substitution σ_S induced by the corresponding assignment S satisfies $\sigma_S(A) \sqsubseteq \sigma_S(B)$, and setting $[X > Y] = 1$ means that $X >_S Y$. The clauses in $C(\Gamma)$ are such that Γ has a unifier iff $C(\Gamma)$ is satisfiable. In particular, any propositional valuation τ satisfying $C(\Gamma)$ defines an assignment S^τ with $S_X^\tau := \{A \mid \tau([X \sqsubseteq A]) = 1, A \in \text{At}_{\text{nv}}\}$, which induces a local unifier of Γ . Conversely, any local unifier of Γ can be obtained in this way. Thus, by generating all propositional valuations satisfying $C(\Gamma)$ we can generate all local unifiers of Γ . The main advantage of this algorithm is the speed of the used SAT solver. However, the number of generated clauses is in general cubic in the size of the unification problem. Thus, the translation will generate huge SAT instances even from moderately sized unification problems, which might lead to memory problems even before the SAT solver is applied.

The Rule-Based Algorithm

The second unification algorithm implemented in UEL is based on the rule-based algorithm from [8]. However, internally it uses subsumptions of the form $C \sqsubseteq^? D$ instead of equivalences. This is without loss of generality since any equivalence $C \equiv^? D$ can be expressed by the two subsumptions $C \sqsubseteq^? D$ and $D \sqsubseteq^? C$. This variant of the algorithm has been described in more detail in [2].

The algorithm generates local unifiers by maintaining a set of current subsumptions Γ and a current acyclic assignment S , both of which are extended by applying certain rules. Initially, all subsumptions are marked as *unsolved* and rules apply only to unsolved subsumptions and mark them as *solved*. In the process, new subsumptions may be generated and the current assignment may be extended by adding non-variable atoms to some of the sets S_X . Once all subsumptions are solved, the substitution σ_S induced by the current assignment is a unifier of the unification problem.

Some of the rules are don't-know nondeterministic, i.e., they might apply in different ways to the same subsumption, but we do not know beforehand which application is the correct one. Also the choice between several applicable nondeterministic rules is don't-know nondeterministic. The algorithm additionally employs several *eager* rules that are always applied first and leave no choice in their application. They are mainly there to reduce the number of nondeterministic choices the algorithm has to make.

In contrast to the SAT reduction, this rule-based algorithm does not generate all local unifiers. A non-variable atom D will only be put in the set S_X if there

³ The reduction in [7] actually uses variables $[A \sqsubseteq B]$, but it turned out that the reduction using non-negated subsumptions behaves better in practice.

is a reason to do so in the unification problem, although there may be a local unifier whose assignment S' contains D in S'_X . However, it was shown in [8] that it can generate all minimal unifiers (up to equivalence). The converse is not true, i.e., it might also generate local unifiers that are not minimal.

The main advantage of this algorithm is that non-variable atoms are only added to the assignment if this is required by the unification problem, and thus fewer unifiers of relatively small size are generated. The space requirements are also quite low, since the current set of subsumptions and the current assignment are of size at most quadratic in the input size. The downside of this algorithm is that, without any of the optimizations implemented in modern SAT solvers, the algorithm naively traverses the search space. However, implementing such optimizations to improve its efficiency requires a huge effort.

3 Stuff not Mentioned in the Theoretical Papers

When implementing UEL, we had to deal with several issues that are abstracted away in the theoretical papers describing unification algorithms for \mathcal{EL} . Most of them are not specific to the used unification algorithm.

Primitive definitions In addition to concept definitions, as introduced above, biomedical ontologies often contain so-called *primitive definitions* $A \sqsubseteq C$ where A is a concept name and C is a concept description. Models \mathcal{I} of $A \sqsubseteq C$ need to satisfy $A^{\mathcal{I}} \subseteq C^{\mathcal{I}}$. Thus, primitive definitions formulate necessary conditions for concept membership, but these conditions are not sufficient. SNOMED CT contains about 350,000 primitive definitions and only 40,000 concept definitions.

By using a trick first introduced by Nebel [13], primitive definitions $A \sqsubseteq C$ can be turned into concept definitions $A \equiv C \sqcap A_UNDEF$, where A_UNDEF is a new concept name that stands for the undefined part of the definition of A . In the resulting acyclic TBox, these new concept names are primitive concepts, and thus can be declared to be variables. In this case, a unifier σ suggest how to complete the definition of A by providing the concept description $\sigma(A_UNDEF)$.

Unifiers as acyclic TBoxes Given an acyclic assignment S computed by one of the unification algorithms, our system UEL actually does not produce the corresponding local unifier σ_S as output, but rather the acyclic TBox $\mathcal{T}_S := \{X \equiv \prod_{D \in S_X} D \mid X \in N_v\}$. This TBox solves the input unification problem Γ w.r.t. \mathcal{T} in the sense that $C \equiv_{\mathcal{T} \cup \mathcal{T}_S} D$ holds for all equations $C \equiv? D$ in Γ . This is actually what the developer that employs unification wants to know: how must the concept variables be defined such that the concept descriptions in the equations become equivalent? Another advantage of this representation of the output is that the size of S and thus of \mathcal{T}_S is polynomial in the size of the input Γ and \mathcal{T} , while the size of the concept descriptions $\sigma_S(X)$ may be exponential in this size. In the following, we will also call the TBoxes \mathcal{T}_S unifiers.

Internal variables As mentioned before, the unification algorithms for \mathcal{EL} assume that the unification problem is first transformed into a flat form. This form

can easily be generated by introducing auxiliary variables. These new variables have system-generated names, which do not make sense to the user. Thus, they should not show up in the output acyclic TBox \mathcal{T}_S . By replacing such auxiliary defined concepts in \mathcal{T}_S by their definitions as long as auxiliary names occur, we can transform \mathcal{T}_S into an acyclic TBox that satisfies this requirement, actually without causing an exponential blow-up of the size of the TBox.

Reachable subontology As mentioned above, acyclic TBoxes are treated by viewing them as part of the unification problem. For very large TBoxes like SNOMED CT, adding the whole TBox to the unification problem is neither viable nor necessary. In fact, it is sufficient to add the reachable part of the TBox, i.e., the definitions on which the concept descriptions in the unification problem depend. This reachable part is usually rather small, even for very large ontologies.

Enumeration of unifiers While computing a single unifier is usually quite fast, computing all of them can take much longer. We alleviate this problem by enabling the user to compute and then inspect one unifier at a time. If this unifier makes sense, i.e., suggests reasonable definitions for the variables, then the user can stop. Otherwise, the computation of the next unifier can be initiated.

For the rule-based algorithm, we output all produced unifiers by a depth-first traversal of the search space. This means that we apply applicable nondeterministic rules as long as this is possible. If there are no more unsolved subsumptions, we return the corresponding unifier. If no rule is applicable, we backtrack and apply the last nondeterministic rule in a different way or apply another rule.

For the SAT reduction, the approach is different. If the SAT solver has provided a satisfying propositional valuation, we can add a clause to the SAT problem that prevents the re-computation of this assignment, and call the SAT solver with this new SAT instance. If the SAT solver determines that the current set of clauses is unsatisfiable, then there are no more unifiers.

Computing only minimal assignments The satisfying valuations of the propositional formula generated by the SAT translation yield *all* local unifiers of the unification problem. Depending on how many concept names are turned into variables, there can be many local unifiers. To address this problem, we implemented a variant of the SAT reduction that computes only the local unifiers induced by *minimal* assignments w.r.t. \succ , which are often significantly fewer.

This approach is still complete in the sense that it computes all minimal unifiers (see Section 2). It works by transforming the SAT problem into a special case of a partial MAX-SAT problem [11], where in addition to the clauses that are to be satisfied one can specify a subset V_{\min} of the propositional variables. The goal is to minimize the number of variables from the set V_{\min} that are set to 1. By setting $V_{\min} = \{[X \sqsubseteq A] \mid X \in N_v, A \in \text{At}_{nv}\}$, we ensure that the first valuation returned by the MAX-SAT solver induces a minimal assignment S .⁴

⁴ If $\{(X, A) \mid X \in N_v, A \in S_X\}$ has minimal cardinality among all assignments that induce a unifier of the unification problem, then it is also minimal w.r.t. set inclusion.

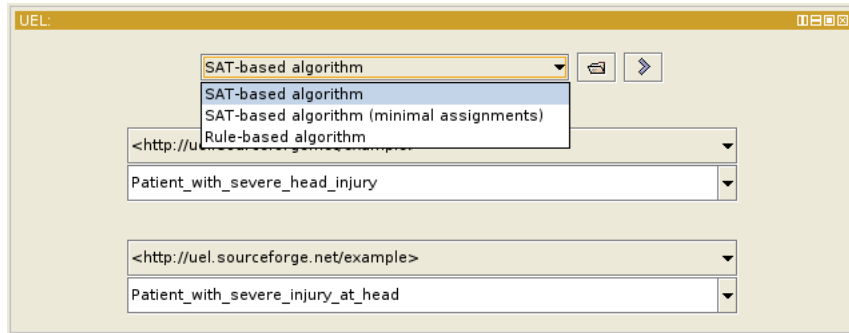


Fig. 1. The user can choose the unification algorithm and the concepts to be unified.

To ensure that subsequent calls to the MAX-SAT solver return no assignments larger than S , we add a clause to the problem instance that requires that at least one of the variables of the form $[X \sqsubseteq A]$ with $A \in S_X$ should be set to 0.

Of course, the reformulation of the problem as a MAX-SAT instance adds an overhead to the computation time. However, this approach guarantees that no “superfluous,” i.e., non-minimal, assignments are presented to the user.

4 The User Interface

UEL was implemented in Java 1.6 and is compatible with Java 1.7. It uses the OWL API 3.2.4⁵ to read ontologies. It has a visual interface that can be used as a Protégé 4.1 plug-in, or as a standalone application. For the SAT-based unification algorithm, we currently use SAT4J⁶ as (MAX-)SAT solver, which is implemented in Java. However, this configuration can easily be changed to any solver that accepts the popular DIMACS CNF⁷ (or WCNF⁸) format as input and returns the computed satisfying propositional valuation. For the rule-based algorithm, we have implemented everything from scratch, and thus have no external dependencies.

After opening UEL’s visual interface, the first step is to open one or two ontologies. The latter enables unification of concepts defined in different ontologies. Additionally, the user can choose between the three unification algorithms by selecting the “SAT-based algorithm [(minimal assignments)]” or the “Rule-based algorithm”. The user can then choose two concepts to be unified. This is done by choosing two concept names that occur on the left-hand sides of concept definitions or primitive definitions (see Figure 1). UEL then computes the subontologies reachable from these concept names, and turns the primitive definitions in these subontologies into concept definitions.

⁵ <http://owlapi.sourceforge.net>

⁶ <http://www.sat4j.org>

⁷ <http://www.satcompetition.org/2011/format-benchmarks2011.html>

⁸ <http://www.maxsat.udl.cat/11/requirements/index.html>

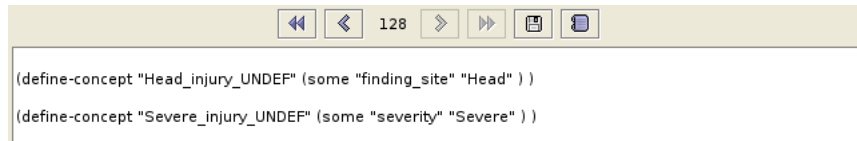









Fig. 2. The user can browse through the computed unifiers.

After choosing the concepts to be unified, pressing the button  opens a dialog window in which the user is presented with the primitive concepts contained in these subontologies (including the ones with ending `_UNDEF`). The user can then decide which of these primitive concepts should be viewed as variables in the unification problem.

Once the user has chosen the variables, UEL computes the unification problem defined this way and opens a dialog window with control buttons. By pressing the button , the user triggers the computation of the first (or next) unifier. Each computed unifier is shown as an acyclic TBox in KRSS format. The button  can be used to go back to the previously computed unifier. The button  can be used to trigger the computation of all remaining unifiers, and the button  allows to jump back to the first unifier (see Figure 2). Computed unifiers are stored, and thus need not be recomputed during navigation. Each unifier (i.e., the acyclic TBox representing it) can be saved using the RDF/OWL or the KRSS format by pressing the button . The file format is determined by the filename extension given by the user (`.owl` or `.krss`).

The user can use the button  to retrieve internal details about the computation process. The unification problem created internally by UEL is then shown in KRSS format in a separate dialog. Additionally, the number of all concept variables (those chosen by the user and internal variables) is given. Depending on the chosen algorithm, several other internal statistics can be viewed. If the SAT reduction is used, the number of propositional letters and the number of propositional clauses are listed. For the rule-based algorithm, the number of initial subsumptions, the maximal number of generated subsumptions (over all nondeterministic choices), and the size of the search tree (i.e., the number of nondeterministic choices) are shown, as well as the number of “dead ends” where the algorithm had to backtrack. These numbers always reflect the current status and might increase if more unifiers are computed.

5 Some Examples

To illustrate the behavior of the three approaches, we consider several example problems. The size of these problems as well as the size of the generated data structures (propositional clauses or subsumptions) is depicted in Table 2, together with the runtimes of the three algorithms on these problems.

The first example is a modified version of our example from the beginning, where the TBox gives (1) as definition for `Patient_with_severe_head_injury` and (2) as definition for `Patient_with_severe_injury_at_head`. In addition, the TBox

#	problem size						SAT		MAX-SAT		Rule	
	equ.	var.	clauses	prop.	subs. (max.)	dead ends	time	unif.	time	unif.	time	unif.
1	7	8	3,976	320	20 (52)	0	0.249	128	0.047	1	0.001	1
2	23	12	17,971	820	47 (217)	63,523	0.500	0	0.510	0	1.920	0
3	34	14	28,071	1,096	62 (320)	1,126,286	1.086	15	1.162	15	26.308	30

Table 2. These are the numbers of equations and variables of some example problems, as well as the numbers of generated clauses, propositional variables, and subsumptions, and the number of dead ends encountered by the rule-based algorithm. The second part contains the runtime and number of computed unifiers of each algorithm.

contains two primitive definitions, saying that `Head_injury` and `Severe_injury` are subconcepts of `Injury`. If we choose `Patient_with_severe_head_injury` and `Patient_with_severe_injury_at_head` as the concepts to be unified, the system offers us the primitive concepts `Patient`, `Severe`, `Head`, `Head_injury_UNDEF`, and `Severe_injury_UNDEF` as possible variables, of which we choose only the latter two.

The SAT translation generates a large SAT problem (4,000 clauses are created from only 7 equations) and first computes the following unifier:

$$\{\text{Head_injury_UNDEF} \mapsto \exists \text{finding_site.Head}, \\ \text{Severe_injury_UNDEF} \mapsto \exists \text{severity.Severe}\}.$$

This substitution completes the primitive definitions of the concepts `Head_injury` and `Severe_injury` to concept definitions $\text{Head_injury} \equiv \text{Injury} \sqcap \text{finding_site.Head}$ and $\text{Severe_injury} \equiv \text{Injury} \sqcap \exists \text{severity.Severe}$.

However, the unification problem has 127 additional local unifiers. Some of them are similar to the first one, but contain “redundant” conjuncts. Others do not make much sense in the application (e.g., ones where `Patient` occurs in the images of the variables). In contrast, the MAX-SAT variant of the translation has to deal with an even larger problem since it has to consider the additional set V_{\min} . However, it is much faster since it computes only the unifier shown above, which is the only minimal unifier of this unification problem. The rule-based algorithm shows an even better performance since the data structures it creates are orders of magnitude smaller than the SAT problem and the unification problem is very deterministic—in fact, no backtracking is necessary. It also returns only the minimal unifier.

The second unification problem was constructed starting with a “hard” SAT instance (according to the approach in [12]), which was translated into a unification problem using the construction in [5]. Even though a large number of propositional clauses are constructed in the reduction back to a SAT problem, the SAT solver takes little time to determine that the problem has no solution. The overhead incurred by the use of a MAX-SAT problem is negligible. In contrast, the rule-based algorithm works on a much smaller data structure (at most 217 subsumptions during the whole run), but lacks the optimizations of the SAT solver and naively traverses a large search tree looking for a unifier.

The last example shows even more extreme differences. Again, we started from a simple SAT problem (“Choose exactly 2 out of 6 variables.”) that has 15

models which correspond to 15 minimal unifiers. Both the SAT-based and the MAX-SAT-based approach compute these unifiers quite fast, but the rule-based algorithm takes much longer and even computes each of the solutions twice.

6 Conclusions

Our system UEL enables ontology engineers to test ontologies for redundancies and allows them to choose between different algorithms with different strengths. The rule-based algorithm has a big advantage over the SAT translation since it needs only small data structures, but currently lacks important search optimizations, which make this implementation unsuitable for large problems.

In the current version 1.2.0, UEL only supports checking two pre-selected concept names for similarities. In future versions, we plan to implement an automatic search feature that can scan (a part of) an ontology for concept names that unify. For this we also need to find an intelligent way to automatically select the variables from a set of primitive concept names.

References

1. F. Baader. Terminological cycles in a description logic with existential restrictions. In *Proc. IJCAI'03*, pages 325–330. Morgan Kaufmann, 2003.
2. F. Baader, S. Borgwardt, and B. Morawska. Unification in the description logic \mathcal{EL} w.r.t. cycle-restricted TBoxes. LTCS-Report 11-05, Chair of Automata Theory, TU Dresden, Germany, 2011. See <http://lat.inf.tu-dresden.de/research/reports.html>.
3. F. Baader, S. Borgwardt, and B. Morawska. Computing minimal \mathcal{EL} -unifiers is hard. LTCS-Report 12-03, Chair for Automata Theory, TU Dresden, 2012. See <http://lat.inf.tu-dresden.de/research/reports.html>.
4. F. Baader, S. Brandt, and C. Lutz. Pushing the \mathcal{EL} envelope. In *Proc. IJCAI'05*, pages 364–369. Professional Book Center, 2005.
5. F. Baader and R. Küsters. Matching concept descriptions with existential restrictions. In *Proc. KR'00*, pages 261–272. Morgan Kaufmann, 2000.
6. F. Baader and B. Morawska. Unification in the description logic \mathcal{EL} . In *Proc. RTA'09*, volume 5595 of *LNCS*, pages 350–364. Springer, 2009.
7. F. Baader and B. Morawska. SAT encoding of unification in \mathcal{EL} . In *Proc. LPAR'10*, volume 6397 of *LNCS*, pages 97–111. Springer, 2010.
8. F. Baader and B. Morawska. Unification in the description logic \mathcal{EL} . *Log. Meth. Comput. Sci.*, 6(3), 2010.
9. F. Baader and P. Narendran. Unification of concept terms in description logics. *J. Symb. Comput.*, 31(3):277–305, 2001.
10. S. Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In *Proc. ECAI'04*, pages 298–302. IOS Press, 2004.
11. C. M. Li and F. Manyà. Maxsat, hard and soft constraints. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 19, pages 613–631. IOS Press, 2009.
12. K. Markström. Locality and hard SAT-instances. *JSAT*, 2(1-4):221–227, 2006.
13. B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *LNAI*. Springer, 1990.