Ontological Reasoning for Natural Language Understanding

Fabian M. Suchanek

Master's Thesis in Computer Science
Saarland University/Germany

March 15, 2005

Supervisors: Prof. Dr. Gerhard Weikum and Dr. habil. Peter Baumgartner
Max Planck Institute for Computer Science Saarbrücken/Germany

Abstract

This thesis presents OntoNat, a prototypical system for answering Yes/No-questions on natural language sentences. Different from existing systems, OntoNat uses background knowledge from the Suggested Upper Model Ontology (SUMO)[NP01], so that it can perform some kind of common sense reasoning to answer a question. SUMO is translated to a disjunctive logic program (DLP). The input sentence and the Yes/No-question are also translated to DLPs, in cooperation with the Computational Linguistics Department of Saarland University. These DLPs are given to a first-order theorem-prover (KRHyper[Wer03]), which tries to answer the question.

Acknowledgement

This work would not have been possible without the persisting support of my supervisor Dr. habil. Peter Baumgartner. I would like to thank him for his patience and his fruitful comments.

Contents

1 Introduction

1.1 Motivation

One of the challenges of today's information society is to cope with the immense amount of digitally stored texts, as for instance provided by the Internet. Currently, over a hundred tera-bytes of data are online, distributed on billions of Internet pages. One can find journals, theses, reports and even books on the Internet.

All of these texts are written in natural language. But since computers cannot interpret natural language, their use is often limited to storing these texts and searching them for key-words. Computers are virtually blind to the tasks that humans are most interested in, namely tasks that require the *understanding* of a text and the *common sense reasoning* on it. A typical example is the task of answering a Yes/No-question on a text. Suppose the question is whether Elvis Presley is still alive. Assume that we find a text with the sentence "Elvis comes to Saarbruecken on January 1st". Common sense tells us that if Elvis comes to Saarbruecken, he must be alive. But can a computer know this and consequently give us the answer yes, Elvis is alive? This sounds close to impossible.

Existing Systems. It is true that today, powerful systems are being developed that can answer semantic questions on natural language texts to some degree. One of them is the COMPASS system[GBZ+04], which can find items in Web page data that match a given description. Another one is COGEX [MCHM03], which can find answers to questions in natural language texts. However, the world knowledge of existing question answering systems is often limited to a hyponym-structure. Hence, these systems cannot provide answers that require true common sense reasoning on the question.

OntoNat. This thesis presents OntoNat, a logic-based prototype for answering Yes/No-questions on natural language sentences with common sense reasoning. Obviously, question answering with common sense will never be entirely solvable by a computer on its own: Language analysis suffers from severe problems of ambiguity and incomplete information, the term "common sense reasoning" is only vaguely defined, an answer depends heavily on the available background knowledge and last, even humans cannot always tell whether the answer to a question is "Yes" or "No". These difficulties entail that the system

presented here can only have a tentative character. Its sole purpose is to show how a logic-based approach could contribute to automated question answering with common sense reasoning.

1.2 The OntoNat Approach

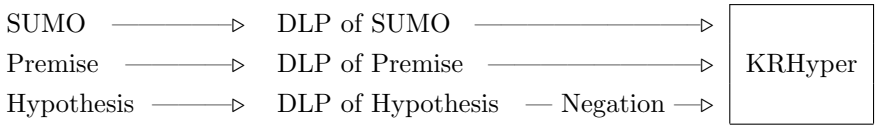
OntoNat. A sentence and an accompanying Yes/No-question can be seen as a premise ("Elvis comes to Saarbruecken") and a hypothesis ("Elvis is alive"). If the premise logically entails the hypothesis, the answer to the question is "Yes". This is how the OntoNat system works: It is given a premise and a hypothesis and it tries to find out whether the premise entails the hypothesis according to common sense. OntoNat bases on a logical analysis of the input sentences and it uses a theorem prover to verify or falsify the entailment. The system was developed together with Peter Baumgartner from the Max Planck Institute for Computer Science in Saarbrücken/Germany and the SALSA-group from the Computational Linguistics Department of Saarland University/Germany. The linguistic part of OntoNat was designed by the SALSA-group and the logical part was implemented by me.

Background Knowledge. Finding a deep semantic entailment between a premise and a hypothesis requires a huge amount of background knowledge. In the Elvis-example, the system has to know that *coming to some place* is a kind of *intentional process* and that any intentional process entails that its agent is alive. OntoNat exploits one of the richest ontologies available today, the Suggested Upper Merged Ontology (SUMO)[NP01]. SUMO provides a structure of concepts (such as *Human Being* or *Body Motion*), their interrelations (e.g. every human being is a mammal) and common sense knowledge in form of axioms (e.g. if somebody moves to some place, then he will be in this place afterwards).

Logical Representation. For the logical reasoning, OntoNat uses Disjunctive Logic Programs (DLPs). The simplest of these, the positive DLPs, form a subset of First Order Logic, for which efficient theorem provers are available. Unfortunately, SUMO is not written in First Order Logic, but in KIF, a Higher Order Knowledge Representation Language. Hence SUMO has to be translated (as far as possible) to a DLP. The component of OntoNat that performs this translation lies in the focus of this thesis.

Running OntoNat. If OntoNat is given a premise and a hypothesis, it translates them to DLPs. For this purpose, the sentences are first parsed syntactically. Then, each content word is disambiguated and mapped to its corresponding SUMO concept. This is the part of OntoNat that was crafted by the SALSA-group. A subsequent semantic analysis of the syntactic structure yields DLPs for the input sentences.

In the next step, the DLP of the hypothesis is negated. Finally, the DLP of SUMO, the DLP of the premise and the negated DLP of the hypothesis are given to a theorem prover. This amounts to querying whether it is possible that the premise holds, all background knowledge holds, but the hypothesis is false. If the theorem prover finds this to be impossible, then the premise entails the hypothesis. OntoNat uses the theorem prover KRHyper[Wer03]. The plan of attack is shown here:



1.3 Outline of This Thesis

The second chapter of this thesis introduces the logical components of OntoNat together with their theoretical background. This includes the Suggested Upper Model Ontology SUMO together with its representation language KIF as well as the theorem prover KRHyper together with the formal theory of DLPs.

The rest of the thesis proceeds as illustrated in the above picture: The third chapter treats the translation of SUMO to a DLP. This constitutes the main part of the thesis. The translation covers the treatment of higher order formulae, equality and functions. The fourth chapter describes how the premise and the hypothesis are translated to DLPs. It gives an overview of the part of OntoNat that was designed by the SALSA-group. Furthermore, the chapter explains the semantic analysis of the input sentences. The fifth chapter shows how the OntoNat components work together. By some experimental results, it points out the challenges that would have to be taken in order to elaborate the OntoNat prototype to a fully-grown system. A conclusion and an outlook complete the thesis.

2 Logical Components of OntoNat

This chapter introduces the logic-related components of OntoNat. The first section presents the knowledge representation language KIF. KIF is the basis of the ontology SUMO, which is described in the second section. The third section covers the theory of DLPs. The automated reasoner for DLPs, KRHyper, is introduced in the fourth section.

2.1 KIF

The Knowledge Interchange Format[Gen91] (KIF) is a powerful knowledge representation language, which was developed by M. Genesereth at Stanford University/California. The language is intended as a mediating formalism for different interacting knowledge processing systems. KIF can be seen as a superset of First Order Logic. SUMO is written in a variant of KIF, SUO-KIF[Pea04], which differs only slightly from the original language.

2.1.1 KIF Syntax

Identifiers. The syntax of KIF resembles the one of LISP[1]. Its basic units are *identifiers*. An identifier is a finite sequence of upper case characters, lower case characters, digits and alphanumeric symbols. Identifiers are classified into three categories:

- *operators*. These are all logical operators, i.e. `and`, `or`, `=>`, `<=>` and `not`.
- *variables*. These are identifiers starting with a question mark '`?`'.
- *symbols*. These are all identifiers that are neither operators nor variables. The set of symbols is called the *signature*. Although the original KIF specification does not impose any further syntactic constraints on symbols, this is useful in our case: We classify symbols into *constants*, *function symbols* and *relation symbols*. Furthermore, we assume a natural number associated to every function symbol and relation symbol (the *arity*). Function symbols end with '`Fn`' in SUMO.

[1] We use notions that differ slightly from the original KIF specification to have uniform notions for KIF and DLPs.

Terms. Identifiers are used to make up *terms*. The core of KIF, as used in SUMO, bases on the following kinds of terms:

- Every variable is a term.
- Every symbol is a term[2].
- The expression `(<functionsymbol> <term> ...)` is a term, if the number of terms complies with the arity of the function symbol. Terms of this kind are called *functional terms*.

Sentences. Terms are the constituents of *sentences*. A sentence can be

- a *relational sentence* of the form `(<relationsymbol> <term> ...)`, where the number of terms must comply with the arity of the relation symbol.
- a *logical sentence* of the form `(<operator> <sentence> ...)`, where the number of sentences must be appropriate for the operator.
- a *quantified sentence* of the form

```
(forall | exists (<variable> ...)  <sentence>)
```

If a sentence s contains a variable v that is not bound by a surrounding `forall` or `exists`, then this variable is said to be $free$. Mostly, such a sentence is to be read as an abbreviation of `(forall (`v`)` s`)`, so that we read all sentences as sentences without free variables.

As usual, the terms in an expression of the form `(<symbol> <term> ...)` are called the *arguments* of the symbol. If a sentence is *quoted* by a preceding apostrophe, then the sentence becomes a term. In the variant of KIF used in SUMO, any sentence that appears as an argument to a relation symbol or function symbol is implicitly quoted.

Advanced Features. KIF knows an impressive number of further syntactical features. These include *row variables*, which stand for a sequence of terms. They can be used to refer to multiple arguments of a function symbol. Since SUMO makes only sparse use of row variables, OntoNat ignores row variables. Furthermore, KIF defines a number of data-types, including strings, numbers and lists with their respective operators. SUMO repeats these definitions in its

[2]Note that hence all function symbols are terms.

own variation, but seldom applies the data types, so that OntoNat disregards them. Last, KIF allows the user to define functions, relations and objects. Since this mechanism is completely replaced by ontological definitions in SUMO, KIF definitions will not be treated here.

2.1.2 KIF Semantics

Universe of Discourse. The *universe of discourse* U is a finite set of *objects*. As usual, a *relation* is a set of tuples on U and a *function* is a complete right-unique relation. Different from the standard semantics of First Order Logic, all functions and relations on U must themselves be objects of the universe, because functions may be applied to functions in KIF. Originally, KIF does not require the elements of a relation to be tuples of the same dimension, i.e. the very same relation may have different numbers of arguments. However, SUMO complies with the usual convention that relations have fixed arities.

Interpretations. An *interpretation* I is a complete function that maps the constants of KIF to (non-relational) objects of the universe, the relation symbols to relations on the universe with the appropriate arity and function symbols to appropriate functions. KIF does not specify the interpretation of quoted sentences. We assume that sentences are part of the universe and that I maps a quoted sentence to the corresponding sentence in the universe.

Denotations. A *valuation* V is a partial function mapping a variable to an object of the universe. A *denotation* D is the union of an interpretation and a valuation, $D = I \cup V$. To formulate the semantics of KIF conveniently, we say that a denotation D' is an $(x_1, \ldots, x_n)$-*variant of* D, iff $D' = D \backslash \{(x_i, D(x_i)) | 1 \leq i \leq n\} \cup \{(x_i, z_i) | 1 \leq i \leq n\}$ for some $z_i \in U$. D is extended to functional terms of the form $(f\ t_1\ \ldots\ t_n)$:

$$D((f\ t_1\ \ldots\ t_n)) = D(f)(D(t_1), \ldots, D(t_n))$$

Evaluations. An *evaluation* E is a function mapping a denotation and a sentence to one of the truth-values **true** or **false**. It is defined as usual for relation symbols r, logical operators o, terms $t_1, \ldots, t_n$ and sentences $s_1, \ldots, s_n$:

$$E(D,(r\ t_1\ \ldots\ t_n)) = \begin{cases} \text{true} & \text{if } (t_1, \ldots, t_n) \in D(r) \\ \text{false} & \text{else} \end{cases}$$

$$E(D,\texttt{(}o\ s_1\ \ldots\ s_n\texttt{)}) = \begin{cases} \text{true} & \text{if } E(D,s_1)\ o\ \ldots\ o\ E(D,s_n) \\ \text{false} & \text{else} \end{cases}$$

$$E(D,\texttt{(forall (}v_1\ \ldots\ v_n\texttt{) }s\texttt{)}) = \begin{cases} \text{true} & \text{if } E(D',s) = \text{true for all} \\ & (v_1,\ldots,v_n)\text{-variants} \\ & D' \text{ of } D \\ \text{false} & \text{else} \end{cases}$$

$$E(D,\texttt{(exists (}v_1\ \ldots\ v_n\texttt{) }s\texttt{)}) = \begin{cases} \text{true} & \text{if there exists a} \\ & (v_1,\ldots,v_n)\text{-variant } D' \text{ of} \\ & D \text{ such that } E(D',s) = \text{true} \\ \text{false} & \text{else} \end{cases}$$

Since free variables in a sentence are supposed to be implicitly bound by a surrounding `forall`, the valuation V can be empty. Hence one simply writes $I \models s$ for $E(I,s) = \text{true}$. A KIF-sentence s is *satisfiable* iff there is an interpretation I with $I \models s$. A set of KIF-sentences $\{s_1,\ldots,s_n\}$ is *satisfiable*, iff `(and` s_1 `...` s_n`)` is satisfiable.

2.2 SUMO

The Suggested Upper Model Ontology SUMO[NP01] is the largest formal public ontology available these days. It was created at Teknowledge Corporation and is maintained today by Adam Pease. To introduce it properly, we first discuss some theoretical issues on Ontology in general.

2.2.1 The Computational Model of Ontology

The Study of Ontology. From a philosophical point of view, Ontology[3] is the study of "What is there?" (qtd. in [Sow00]). The universality of this question immediately entails a number of intrinsic problems. First, the study of Ontology exists itself. Hence it must be the subject of its own research. This phenomenon is called the problem of self-reflectivity [BHS93]. Furthermore, the goal of Ontology is the definition of all concepts and terms. Consequently, the study of Ontology, unlike any other study, cannot rely on any predefined concepts or notions. This entails in particular that it cannot define any term without being cyclic.

[3] I follow Guarino's distinction of "Ontology" (meaning the discipline) and "ontology" (meaning a certain conceptualization) [Gua98].

To circumvent these problems, we rely on the *Computational Model of Ontology*, as described in [Suc03][4].

Entities. The basic notion of the Computational Model is the term *entity*. Any abstract or concrete being, whether it exists or not, is an entity.

The Computational Model makes a number of simplifying assumptions on entities: First, entities are assumed discernible. This view is by no means trivial, since all kinds of variations, flows and transitions between entities have to be ignored. Furthermore, the Computational Model assumes entities atomic. They can only be created and destroyed as wholes. The set of all entities that are in the scope of interest is called the *domain*. We assume the relation of identity, i.e. we can tell whether two entities are the same. The relation of identity is connected to a number of problems, which are ignored in the Computational Model.

Properties and Concepts. The Computational Model assumes entities to have certain *properties*. These properties are seen as unary, binary and n-ary relations on the entities, although the choice to represent properties by relations is quite arbitrary. Next, entities with similar properties are grouped into sets called *concepts*. This grouping, called *conceptualization*, is by no means unique and subject to ongoing research. Concepts, as well as properties, are entities, but they need not be elements of the domain. If an entity is an element of a concept, the entity is said to be an *instance* of the concept. If a concept is a subset of another concept, the former is called a *subconcept* of the latter.

'o'ntology. An *ontology* (with lower case 'o') is the description of a domain, its concepts and properties by means of a formal language. Usually, the goal of an ontology is to constrain its symbols in such a way, that a great number of unintended interpretations are ruled out. It is common to call the symbols of the ontology "concepts", "instances" and "properties", although these are semantic notions.

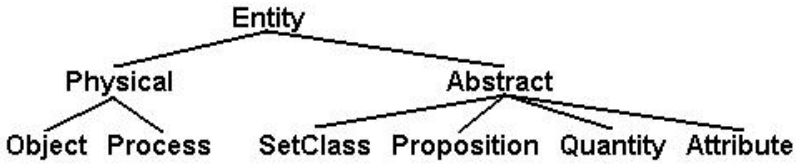
2.2.2 The Suggested Upper Model Ontology

Features. SUMO is an ontology in the above sense. It does not only provide a conceptualization, but also assertions on the properties of entities (so-called

[4]The reader is referred to this paper for a detailed description of the model with its philosophical problems and limitations.

axioms). SUMO is available in several formats. OntoNat uses the version written in KIF. Creating SUMO involved the merging of numerous, partially overlapping ontologies from different sources. The result is a huge database, which, together with its domain-specific extensions, contains more than 20'000 concepts and 60'000 axioms. The concepts include the most common geographic names, languages, financial terms and even the chemical elements.

Concept Hierarchy. The most general concepts of SUMO are illustrated in this picture:



The picture shows that in SUMO, even abstract entities like sets and propositions are elements of the domain. Concepts are called *classes* in SUMO. Hence the subconcept relation in Ontology becomes the `subclass` relation in SUMO. Binary relations between an entity and a value are called *attributes*.

Processes. In SUMO, processes and events are part of the domain. For example, if Gundolf sells a gondola to Gudrun, then SUMO would model this as a selling process. A *role* is a binary relation that links a process to its participating entities. The most important roles are `agent`, `patient`, `source` and `destination`. In the example, Gundolf is the agent of the selling process, the gondola is the patient and Gudrun is the destination. This simplifying approach has a number of well-known drawbacks, but it is appropriate for the abstract view of SUMO.

Axioms. The most impressive component of SUMO are its axioms. To say that the agent of an intentional process must be awake during the time of the process, SUMO states for example:

```
(=> (and (instance ?PROC IntentionalProcess)
         (agent ?PROC ?ANIMAL)
         (instance ?ANIMAL Animal))
    (holdsDuring (WhenFn ?PROC) (attribute ?ANIMAL Awake)))
```

2.3 DLPs

The formal language of Disjunctive Logic Programs (DLPs) can be seen as a relative of the programming language PROLOG. It allows defining facts and one can specify rules that determine how to deduce new facts from known facts. Unlike PROLOG, DLPs are described by a profound and exhaustive theory.

2.3.1 DLP Syntax

Identifiers. The building blocks of DLPs are *identifiers*. Identifiers are usually sequences of upper case characters, lower case characters and the underscore. A DLP uses two sets of identifiers:

- A set X of *variables*. Variables usually start with an upper case character or an underscore.
- A set Σ of *symbols*, called the *signature*. We distinguish *constants*, *function symbols* and *predicate symbols*, where each of the latter two is associated to a natural number (the *arity*)[5]. Σ always contains the special 0-ary predicate symbol `false`.

Terms and Atoms. A (Σ, X)-*term* is one of the following:

- a variable from X.
- a symbol from Σ.
- a sequence of the form $f(t_1,\ldots,t_n)$, where f is an n-ary function symbol from Σ and $t_1,\ldots,t_n$ are (Σ, X)-terms.

This thesis will assume a given set of variables X and a given signature Σ. Hence all symbols mentioned are implicitly symbols of Σ, all variables are implicitly elements of X and all notions implicitly depend on Σ and X. A term is *ground* iff it does not contain variables. The set of all ground terms is called the *Herbrand universe*. An *atom* takes the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms. An atom is *ground* iff all of its terms are ground. The set of all ground atoms makes up the *Herbrand base*.

[5] We deviate from the standard notions here by distinguishing 0-ary function symbols and constants. However, with an empty set of constants, the definitions are fully compatible with the standard notions.

Rules. A *rule* is an expression of the form

$$H_1 \texttt{ ; ...; } H_m \texttt{ :- } B_1 \texttt{ , ..., } B_k \texttt{, not(} B_{k+1} \texttt{), ..., not(} B_n \texttt{).}$$

where $m \geq 1$, $n \geq k \geq 0$ and all $H_1, \ldots, H_m, B_1, \ldots, B_n$, are atoms. Each H_i is called a *head atom* and each B_j is called a *body atom*. If there are no body atoms, one omits the `:-`. The body atoms $B_{k+1}, \ldots, B_n$ are called *negative atoms*. A rule without negative atoms is called a *positive rule*. Rules with only one head atom are called *normal rules*, the others are said to be *disjunctive*. Rules must be *domain-restricted*, i.e. every variable occurring in the rule must occur in a non-negated atom of the body. A rule is called *ground*, iff all of its terms are ground.

The intuitive meaning of a ground rule is given by its *applicability*: We say that a ground rule is *applicable* to a set M of ground atoms, iff $\{B_1, \ldots, B_k\} \subseteq M$ and $M \cap \{B_{k+1}, \ldots, B_n\} = \emptyset$. M is said to *satisfy* the rule, iff the rule is either not applicable or if at least one $H_i \in M, 1 \leq i \leq m$. This means that a positive ground rule can be read as an implication, where the presence of the body atoms in M implies the presence of at least one head atom in M. A finite set of rules is called a *Disjunctive Logic Program* (DLP). A DLP is *normal* iff all of its rules are normal, it is *positive* iff all of its rules are positive and it is *ground* iff all of its rules are ground.

2.3.2 DLP Semantics

Interpretations. A *domain* is a set of objects. An *interpretation* for a domain is a function that maps constants to objects of the domain, n-ary function symbols to n-ary functions on the domain and n-ary relation symbols to n-ary relations on the domain. A *denotation* D_I is an interpretation I, extended to ground terms of the form $f(t_1, \ldots, t_n)$ by

$$D_I(f(t_1, \ldots, t_n)) = D_I(f)(D_I(t_1), \ldots, D_I(t_n))$$

An *evaluation* is a function E that is defined for all predicate symbols p, for a denotation D_I and for ground terms $t_1, \ldots, t_n$ as

$$E(D_I, p(t_1, \ldots, t_n)) = \begin{cases} true & \text{if } D_I(p)(D_I(t_1), \ldots, D_I(t_n)) \\ false & \text{else} \end{cases}$$

As in First Order Logic, we write $I \models a$ for an interpretation I and a ground atom a, if $E(D_I, a) = \text{true}$.

Herbrand Interpretations. We restrict ourselves to *Herbrand interpretations*: A *Herbrand interpretation* is an interpretation that maps

- constants to themselves.
- n-ary function symbols f to functions $\{(t_1, \ldots, t_n, f(t_1, \ldots, t_n)) | t_1, \ldots, t_n \in U\}$, where U is the Herbrand universe, i.e. a denotation will map the term $f(t_1, \ldots, t_n)$ to itself.
- `false` to a false 0-ary relation.
- n-ary predicate symbols to n-ary relations on the Herbrand universe.

Since the only contingent component of a Herbrand interpretation is its definition for predicate symbols, we may represent a Herbrand interpretation I by a set M of ground atoms (i.e. a subset of the Herbrand base) with

$$I \models a \quad \Leftrightarrow \quad a \in M$$

Models. For the definition of models, we need some further notational tools. We define a *substitution* as a total function from variables to terms. If σ is a substitution, one extends σ to terms, atoms, rules and sets:

$\sigma(c) = c$ for symbols c

$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$ for function symbols f

$\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$ for predicate symbols p

$$\begin{aligned}\sigma(H_1 \texttt{ ;...; } H_m \texttt{ :- } B_1\texttt{, ..., } B_k\texttt{, not(}B_{k+1}\texttt{),..., not(}B_n\texttt{).}) = \\ \sigma(H_1) \texttt{ ; ...; } \sigma(H_m) \texttt{ :- } \sigma(B_1)\texttt{, ..., } \sigma(B_k)\texttt{,} \\ \texttt{not(}\sigma(B_{k+1})\texttt{), ..., not(}\sigma(B_n)\texttt{).}\end{aligned}$$

$\sigma(\{t_1, \ldots, t_n\}) = \{\sigma(t_1), \ldots, \sigma(t_n)\}$

It is common to write $s\sigma$ instead of $\sigma(s)$. Since a substitution often maps most variables to themselves, we note a substitution by giving only the variables that are not mapped to themselves. A substitution is *ground* for a rule, iff the substitution applied to the rule yields a ground rule. A *ground instance* of a rule is the result of the application of a ground substitution to it.

The notion of ground instances allows us to generalize the notion of applicability to non-ground rules: A non-ground rule is *applicable* to a subset M of the Herbrand base, iff one of its ground instances is applicable. M *satisfies* the rule, iff it satisfies all of its ground instances. M is a *model* for a DLP P, iff M satisfies all of its rules and if it does not contain `false`. We write $M \models P$. If C is a rule, then $M \models \{C\}$ means that M satisfies the rule, and we write $M \models C$. If a is an atom, $a \in M$ means that "$M \models a$." and we omit the dot.

Satisfiability. A DLP is said to be *satisfiable* iff it has a model. This unveils the relation of DLPs to First Order Logic: Any First Order Logic formula can be turned into an equisatisfiable positive DLP. First, the formula has to be skolemized and transformed to conjunctive normal form. Then, each disjunction makes up a positive rule, where the positive literals of the formula become head atoms and the negative literals become body atoms. The universal quantifiers can be omitted, because implicitly, each variable in a rule is universally quantified. This transformation will be analyzed in detail later in this thesis.

Possible Models. We define a special kind of models, the so-called *possible models*[Sak90]: Given a DLP P, we call the (possibly infinite) set of all ground instances of all rules of P the *ground DLP* P^{gr} of P. A *simple split DLP* of a ground DLP P^{gr} is a DLP that results from P^{gr} if all but one of the head atoms are deleted in every rule. A *split DLP* of a DLP is the union of at least one of its simple split DLPs. Consequently, a split DLP is always normal.

A subset M of the Herbrand base is said to be a *perfect model* of a ground normal DLP, iff it satisfies all of its rules and if there is no smaller set with this property. Models may not contain the special atom `false`. A *possible model* of a DLP is a set of atoms that is a perfect model for one of the split DLPs of its grounded DLP. Each possible model is a model and if there exists a model, then there exists a possible model.

2.4 KRHyper

KRHyper[Wer03] is a First Order Logic Theorem Prover, which was developed by Peter Baumgartner and Christoph Wernhard at the University of Koblenz-Landau/Germany. Its basic purpose is calculating models for DLPs.

2.4.1 KRHyper Functionality

Features. Compared to other theorem provers, KRHyper offers a number of features that make it particularly suited for OntoNat: First, it was designed to handle large sets of uniform data efficiently. This is crucial, because OntoNat requires reasoning on the whole SUMO. Next, KRHyper supports non-positive DLPs. This makes the translation of SUMO to a DLP easier. Last, the facility of calculating models is built-in in KRHyper. OntoNat makes extensive use of models for the translation of the input sentences.

Restrictions. KRHyper poses some restrictions on the input DLP. Most saliently, any variable occurring in a head atom of a disjunctive rule has to be bound in a positive body atom. Next, KRHyper accepts only DLPs with a restricted *call graph*. The call graph of a DLP is the graph of its atoms, where an atom B is linked by an edge to an atom H if B is a body atom in a rule that contains H as a head atom. If B is a negative body atom, the edge is marked as negative. KRHyper needs this graph to be *stratified*, i.e. there may not be a cycle that contains a negative edge.

2.4.2 KRHyper Algorithm

KRHyper bases on the Hyper Tableau Calculus[BFN96]. To generate a model for an input DLP, KRHyper starts with the set of those atoms that are head atoms of normal rules without body atoms. It arranges them in a (so far linear) tree, the *Hyper Tableau*. The set of all atoms from the bottom leaf to the root constitutes a candidate for a model. Whenever some ground instance of a normal rule is applicable to the current model, KRHyper adds the head atom to the model. This process is called *deriving*. KRHyper continues deriving new atoms from the normal rules, until no more normal rule is applicable.

Then, KRHyper turns to the disjunctive rules. If a ground instance of a disjunctive rule is applicable, KRHyper introduces a branching in the tree. Each branch assumes one of the head atoms to be in the model. Consequently, new rules become applicable in each branch. KRHyper chooses one of the branches and continues the procedure.

When KRHyper reaches a point where no more rule is applicable, a so-called *fixed point*, all atoms on the path to the root of the Hyper Tableau are output as a model. Whenever the special atom `false` is derived, KRHyper abandons the current branch and turns to the next. In this case, the branch is said to be

closed. The stratification of the input program enables KRHyper to choose the order of the rule applications in such a way, that no rule with a negative body atom B is applied, if B appears in the model at the end.

In essence, KRHyper performs a depth-first search on the Hyper Tableau until a model is found. To avoid creating atoms of infinite size, as it may happen with head atoms containing function symbols, KRHyper works with a threshold on the size of atoms. This threshold is increased iteratively and can also be bound by the user. Although models are sets of ground atoms, KRHyper may output models containing atoms with variables. This simply means that all ground instances of the atom are in the model. KRHyper can be made to calculate possible models, so that the properties of the output models are well defined.

3 Translation of SUMO to a DLP

OntoNat translates the KIF-sentences of SUMO to a DLP. The resulting DLP is *equisatisfiable*, i.e. it is satisfiable iff the original set of KIF-sentences is satisfiable. SUMO knows a number of features that cannot be expressed directly in a DLP. These are in particular

- Meta-Identifiers
- Class-Creating Functions
- Propositional Relations
- Equality

This chapter shows how SUMO can be translated to a DLP in spite of that, if some assumptions are made.

3.1 Meta-Identifiers

Meta-identifiers are those identifiers that describe classes or relations rather than "ordinary", non-relational entities. An example is the binary relation `disjoint`, which holds between classes that share no instance. In SUMO, meta-identifiers are part of the domain. They are defined as instances of classes and they may occur in axioms and sentences. In a DLP, however, meta-identifiers cannot be integrated that smoothly into the signature.

3.1.1 Meta-Identifiers in SUMO

Types of Meta-Identifers. We distinguish two types of meta-identifiers: *Meta-classes* are classes, the instances of which are functions or relations. Examples include `asymmetricRelation` or `equivalenceRelation`. *Meta-relations* are relations that take a class, a relation or a function as their argument. An example is the above-mentioned `disjoint` or the binary relation `range`, which defines the range of a function.

Problems With Meta-Identifers. For the translation to a DLP, meta-identifiers are connected to a number of problems:

1. Definitions of meta-identifiers are likely to cause difficulties with self-references. Consider for example the definition of the meta-relation `instance`, which reads

```
(instance instance binaryPredicate)
```

2. Reasoning on meta-identifiers would require higher order variables that range over relations. Higher order variables cannot be expressed in DLPs.

3. DLPs already possess a built-in semantics of notions like `predicate` and `arity`. Defining these notions again would uselessly inflate the knowledge base and might cause conflicts with the built-in semantics.

In addition to these problems, meta-identifiers are only of very limited use to OntoNat, because the system will rarely be applied to texts about abstract entities like relations and predicates. Hence, the translation of SUMO to a DLP omits meta-identifiers altogether. The sections of "structural ontology", "base ontology", and "set/class theory" in the SUMO ontology are cut away. Thereby, the definitions of most meta-identifiers get lost.

3.1.2 Translation of Meta-Identifiers

Since the definitions of meta-identifiers have been cut away, the identifiers have to be transcribed whenever they appear in later SUMO code. Meta-identifiers occur mostly in relational sentences that stand on their own. These sentences are translated directly to DLP rules, circumventing the problematic identifiers.

Meta-Classes. Meta-classes appear most prominently in instance definitions. For example, the following sentence says that the relation r is a reflexive relation.

```
(instance r reflexiveRelation)
```

It appears reasonable to ignore the relationship between the instance and the meta-class and to represent the essential consequences for the instance instead. Hence instances of asymmetric, irreflexive, symmetric, reflexive and equivalence relations become predicates with the respective properties. The above example is translated to

```
r(X,X) :- instance(X,d).
```
where d is the domain of r

The domain of r is given by domain-defining meta-relations (see below). Some meta-classes serve only to declare the arity of a relation. For example, the following sentence says that the arity of r is 3:

`(instance` r `ternaryRelation)`

The translation algorithm memorizes the arity, but does not produce a DLP rule for the sentence.

Domain-Defining Meta-Relations. Some meta-relations are used to define the domain of an argument position of another relation. For example, the following sentence says that the p^{th} argument of a relation r must be an instance of the class c:

`(domain` r p c`)`

This sentence is translated to a rule that deduces that any p^{th} argument of r will be an instance of c[6]:

`instance(X`p`,`c`) :- r(X1,...,X`n`).`
where n is the arity of r

The binary version of `domain` and the relation `range` can be translated similarly, as they are used shorthand to define the domain of the first and last argument of a relation, respectively. The relations `domainSubclass` and `rangeSubclass` declare that an argument must be a subclass of some given class. They can be translated analogously to `domain` and `range`, with the predicate `subclass` in place of `instance`.

Other Meta-Relations. Meta-relations are also used to define relationships between classes, relations or attributes. The names of the relations are self-explaining. Let the c_i stand for classes, the r_i for relations and the a_i for attributes. The translation is as follows:

`(disjoint` c_1 c_2`)`
$\rightsquigarrow$ `false :- instance(X,`c_1`), instance(X,`c_2`).`

`(inverse` r_1 r_2`)`
$\rightsquigarrow$ r_1`(X,Y) :-` r_2`(Y,X).`
r_2`(X,Y) :-` r_1`(Y,X).`

`(contraryAttribute` a_1 ... a_k`)`

[6] If the given class of the p^{th} argument conflicts with c, other axioms in SUMO make sure that a contradiction results.

$\rightsquigarrow$ `false :- attribute(X,`a_i`), attribute(X,`a_j`).`
for all distinct $i, j \in \{1, \ldots, k\}$

`(subattribute` a_1 a_2`)`
$\rightsquigarrow$ `attribute(X,`a_2`) :- attribute(X,`a_1`).`

`(exhaustiveAttribute` a_0 a_1 `...` a_k`)`
$\rightsquigarrow$ `attribute(X,`a_1`) ; ...; attribute(X,`a_k`) :- attribute(X,`a_0`).`

The second constraint of `exhaustiveAttribute`, namely that each $a_1, \ldots, a_k$ be a subattribute of a_0, is specified explicitly in the SUMO code. Hence it can be left away in the translation of `exhaustiveAttribute`. Similarly, the subclass relation does not need to be specified again in `exhaustiveDecomposition`:

`(exhaustiveDecomposition` c_0 c_1 `...` c_k`)`
$\rightsquigarrow$ `instance(X,`c_1`) ; ...; instance(X,`c_k`) :- instance(X,`c_0`).`

`(disjointDecomposition` c_0 c_1 `...` c_k`)`
$\rightsquigarrow$ `instance(X,`c_1`) ; ...; instance(X,`c_k`) :- instance(X,`c_0`).`
`false :- instance(X,`c_i`), instance(X,`c_j`).`
for all distinct $i, j \in \{1, \ldots, k\}$

`(disjointRelation` r_1 `...` r_k`)`
$\rightsquigarrow$ `false :-` r_i`(X1,...,X`n`),` r_j`(X1,...,X`n`).`
for all distinct $i, j \in \{1, \ldots, k\}$, where n is the arity of the relations

`(subrelation` r_1 r_2`)`
$\rightsquigarrow$ r_2`(X1,...,X`n`) :-` r_1`(X1,...,X`n`).`
where n is the arity of r_1 and r_2

The meta-relations `subclass` and `instance` are kept. They are constrained by adding the following axiomatization once to the DLP:

```
subclass(X,Z) :- subclass(X,Y), subclass(Y,Z).
instance(X,B) :- instance(X,A), subclass(A,B).
```

Equisatisfiability. It is easy to see that the above translation keeps the intended semantics of the meta-identifiers. Hence the translation preserves equisatisfiability – as long as the meta-identifiers do not appear in sentences not

covered by the above schemata. Since this is very rare, the translation omits this kind of sentences. Likewise, it ignores some other rare untranslatable meta-identifiers, such as for example `holds`, which states that a sentence given by a variable evaluates to **true**.

3.2 Class-Creating Functions

KIF knows a number of functions that return a new class. The axiomatizations of these functions require variables ranging over classes. Since these can cause problems on the level of the DLP[7], class-creating functions have to be treated apart. Functional terms that contain class-creating functions are replaced by simpler terms. For each such replacement, a supplementary KIF-sentence is added to the set of KIF-sentences. The most important class-creating functions are `UnionFn`, `ExtensionFn` and `KappaFn`.

UnionFn. `UnionFn` returns the union of two classes. In SUMO, it is only applied to constants. Hence each appearance of a functional term `(UnionFn` c_1 c_1`)` is replaced by a fresh class identifier c. Then c is constrained as follows:

```
(forall (?X) (<=> (instance ?X c)
                  (or (instance ?X c1) (instance ?X c2))))
```

ExtensionFn. `ExtensionFn` takes a (constant) attribute and returns the class of those entities that have the attribute. Again, each functional term `(ExtensionFn` a`)` is replaced by a fresh class identifier c and c is constrained by:

```
(forall (?X) (<=> (instance ?X c) (attribute ?X a)))
```

KappaFn. `KappaFn` takes two arguments: An *entity variable* x and a *class sentence* s. It returns the class of those entities e that make s evaluate to **true**, if the valuation maps x to e. The following functional term returns for instance the class of all gondola sellers:

```
(KappaFn ?X (exists (?G ?S) (and (instance ?G Gondola)
                                 (instance ?S Selling)
                                 (patient ?S ?G)
                                 (agent ?S ?X))))
```

[7] Inter alia, they may cause disjunctive rules to derive non-ground atoms.

The same functional term with `KappaFn` may return different classes if used in different sentences, because the class sentence may contain free variables. Hence, each functional term of the form `(KappaFn` x s`)` is replaced by a functional term of the form `(`c v_1 ... v_k`)`, where c is a fresh function symbol and $v_1, \ldots, v_k$ are the free variables of s, excluding x. Then c is constrained as follows:

`(forall (`x v_1 ... v_k`) (<=>` s `(instance` x `(`c v_1 ...v_k`))))`

In the example, the constraint is

```
(forall (?X) (<=> (exists (?G ?S) (and (instance ?G Gondola)
                                       (instance ?S Selling)
                                       (patient ?S ?G)
                                       (agent ?S ?X)))
                  (instance ?X (c))))
```

Equisatisfiability. Since the meta-sections of SUMO are cut away, class-creating functions appear exclusively in functional terms of the above forms. As all of the above transcriptions are bound to a fresh class identifier or function identifier, their effect is purely local. Consequently, it is easy to see that the transcription maintains the semantics of the functions.

3.3 Propositional Relations

In the variant of KIF used by SUMO, sentences may appear as arguments to relations. We will call these relations *propositional relations* and their sentence arguments *propositional arguments*. In a naive translation to a DLP, propositional arguments would simply become terms. Then they would lose their sentential character though.

3.3.1 Propositional Attitudes

Since propositional relations belong to higher order logic and DLP is first order, it is not possible to translate all propositional relations while preserving their meaning. This is why the translation concentrates on *propositional attitudes*, i.e. propositional relations that describe objects of human thinking. These include for instance `desires`, `believes`, `says` and `knows`. We say that an n-ary propositional relation is *i-positioned*, iff it allows a propositional argument as its i^{th} argument, where $1 \leq i \leq n$.

Invariance to Equivalence. Propositional attitudes have some striking properties. To analyze these, we define the notion of *invariance to equivalence*: An interpretation I is *invariant to equivalence* iff for all n-ary i-positioned propositional relations r, for all terms $t_j, 1 \leq j \leq n, j \neq i$ and for all sentences s, s' with $I \models s \Leftrightarrow I \models s'$:

$$\begin{aligned} & I \models \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s\ \ t_{i+1}\ \ldots\ t_n\texttt{)} \\ \Leftrightarrow\ & I \models \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s'\ \ t_{i+1}\ \ldots\ t_n\texttt{)} \end{aligned}$$

There exist propositional relations, for which such an interpretation is inappropriate. Consider for example syntactical properties of sentences, like `isConjunction`: In an interpretation I that is invariant to equivalence

$$I \models \texttt{(isConjunction (and a b))}$$

would entail

$$I \models \texttt{(isConjunction (not (or (not a) (not b))))}$$

because

$$\begin{aligned} & I \models \texttt{(and a b)} \\ \Leftrightarrow\ & I \models \texttt{(not (or (not a) (not b)))} \end{aligned}$$

However, for propositional attitudes, the intended interpretations seem invariant to equivalence. For example, it does not matter whether someone says `(and a b)` or `(and b a)`. This is because human utterances and thoughts are usually not taken by their exact wording, but by their meaning[8].

Partial Validity. Invariance to equivalence in the above sense entails unintended consequences: If someone says something contradictory (like `(and a (not a))`), invariance to equivalence would allow us to conclude for instance that he says that Elvis is alive. This is because in any interpretation I

$$\begin{aligned} & I \models \texttt{(and a (not a))} \\ \Leftrightarrow\ & I \models \texttt{false} \\ \Leftrightarrow\ & I \models \texttt{(and false (alive elvis))} \end{aligned}$$

But it often happens that someone thinks or says something contradictory (e.g. during a long speech) and we do not conclude each time that she says that

[8] All of this presupposes that it is possible to describe human utterances by logical formulae at all, for a discussion see 4.

Elvis is alive. We call this phenomenon *partial validity*. Formally speaking, an interpretation I *allows partial validity*, iff there exist sentences s_1, s_2, an n-ary i-positioned propositional relation r and terms $t_j, 1 \leq j \leq n, j \neq i$ such that

$$\begin{aligned} I \models \texttt{(and} \quad & \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ \texttt{(and}\ s_1\ \texttt{(not}\ s_1\texttt{))}\ t_{i+1}\ \ldots\ t_n\texttt{)} \\ \texttt{(not} \ & \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_2\ \qquad\qquad\ t_{i+1}\ \ldots\ t_n\texttt{)))} \end{aligned}$$

Our goal is now to find a transformation Φ of KIF-sentences with propositional arguments to KIF-sentences without propositional arguments. These sentences can later be translated easily to a DLP. The obvious constraint is that for any sentence s, $\Phi(s)$ be satisfiable iff s is satisfiable.

Propositional Relations and Modal Logic. The problem of propositional relations is closely related to modal logic. One possible translation approach uses one modality for each possible combination of a propositional relation and its non-propositional arguments. As an example, consider two entities, Gundolf and Gudrun, and two propositional relations, `says` and `thinks`. Then we would need 4 modalities:

$$\begin{aligned} \Phi(\texttt{(thinks gundolf}\ x\texttt{)}) &= \Box_{\texttt{thinks,Gundolf}}\ x \\ \Phi(\texttt{(says gundolf}\ x\texttt{)}) &= \Box_{\texttt{says,Gundolf}}\ x \\ \Phi(\texttt{(says gudrun}\ x\texttt{)}) &= \Box_{\texttt{says,Gudrun}}\ x \\ \Phi(\texttt{(thinks gudrun}\ x\texttt{)}) &= \Box_{\texttt{thinks,Gudrun}}\ x \end{aligned}$$

One would opt for a model in which the rule $\Box x \rightarrow x$ is not valid. Modal logic is well established and is often used for similar purposes. The resulting modal formulae could be transformed to First Order Logic and hence to KIF. However, the Rule of Necessitation $x \vdash \Box x$ seems inadequate (consider e.g. the propositional relation `desires` and the provable proposition that there are numerous wars on earth). Furthermore, no interpretation could allow partial validity, because for any modal interpretation I and for any modality $\Box$

$$\begin{aligned} & I \models \Box(s \wedge \neg s) && \wedge\ \neg\Box s' \\ \Leftrightarrow\ & I \models \Box\text{false} && \wedge\ \neg\Box s' \\ \Leftrightarrow\ & I \models \Box(\text{false} \wedge s') && \wedge\ \neg\Box s' \\ \Leftrightarrow\ & I \models \Box\text{false} \ \wedge\ \Box s' && \wedge\ \neg\Box s' \\ \Leftrightarrow\ & I \models \text{false} \end{aligned}$$

Hence modal logic is inappropriate for our purpose.

3.3.2 Semantics of Propositional Attitudes

Propositional Equivalence. We describe the semantics of propositional attitudes by defining when we regard two sentences with propositional attitudes equivalent. We define a *propositional equivalence* $\longleftrightarrow$ on KIF-sentences[9]:

- $s \longleftrightarrow \text{pcnf}(s)$, for any sentence s, where **pcnf** is a function that transforms its argument to prenex conjunctive normal form and eliminates double negations. This rule plays the role of invariance to equivalence: We consider two propositional arguments equivalent, if they are syntactic variants of each other.

- $$\begin{aligned} &(r\ t_1\ \ldots\ t_{i-1}\ (\texttt{and}\ s_1\ s_2)\ t_{i+1}\ \ldots\ t_n) \\ \longleftrightarrow\ &(\texttt{and}\ (r\ t_1\ \ldots\ t_{i-1}\ s_1\ t_{i+1}\ \ldots\ t_n) \\ &\quad\ (r\ t_1\ \ldots\ t_{i-1}\ s_2\ t_{i+1}\ \ldots\ t_n)) \end{aligned}$$

  for all sentences s_1, s_2, n-ary i-positioned propositional relations r and terms $t_j, 1 \leq j \leq n, i \neq j$. This rule mirrors our understanding of conjunctions in propositional arguments. If someone says `(and a b)`, this is commonly regarded the same as if she said `a` and she said `b`. Observe, however, that although this rule applies to propositional attitudes, it does not necessarily apply to all propositional relations. Consider e.g. the propositional relation `illegal`: If the conjunction of two facts A and B is illegal, this does not mean that A and B themselves are illegal.

- $$\begin{aligned} &(r\ t_1\ \ldots\ t_{i-1}\ (Q\ (\vec{v})\ s)\ t_{i+1}\ \ldots\ t_n) \\ \longleftrightarrow\ &(Q\ (\vec{v})\ (r\ t_1\ \ldots\ t_{i-1}\ s\ \ \ \ t_{i+1}\ \ldots\ t_n)) \end{aligned}$$

  for all sentences s, n-ary i-positioned propositional relations r, terms $t_j, 1 \leq j \leq n, i \neq j$, and variables $\vec{v} = v_1, \ldots, v_m$ and with $Q \in \{\texttt{exists}, \texttt{forall}\}$[10]. If someone says that there exists an entity with a certain property, then we may introduce a fictional entity, which we may call e. Then there exists an entity (namely e) that has the property in the view of the speaker. If someone believes that some property holds for all entities, then she believes for all entities, that they have the property. Strictly speaking, this can be problematic if someone believes that a number of entities have

[9] This equivalence is weaker than First Order Logic equivalence, but stronger than intuitionistic equivalence.

[10] Note that this rule allows a renaming of bound variables or a swapping of equal neighboring quantifiers inside a propositional argument. This can be achieved by moving the quantifiers outside, performing the desired equivalence transformation according to classical logic and moving the quantifiers inside again.

a property and if these entities happen to be all existing entities. Then the person does not believe that all entities have the property, because she does not know that she believes it for every existing entity. Propositional equivalence does not mirror our intuitive understanding of propositional attitudes exactly in this case.

- $$\begin{aligned}
&\texttt{(}r\ t_1\ \ldots\ t_{i-1}\ \texttt{(or}\ s_1\ s_2\texttt{)}\ t_{i+1}\ \ldots\ t_n\texttt{)}\\
\longleftrightarrow\ &\texttt{(and (=>(}r\ t_1\ \ldots\ t_{i-1}\ \texttt{(not}\ s_1\texttt{)}\ t_{i+1}\ \ldots\ t_n\texttt{)}\\
&\qquad\texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_2\ \ t_{i+1}\ \ldots\ t_n\texttt{))}\\
&\quad\texttt{(=>(}r\ t_1\ \ldots\ t_{i-1}\ \texttt{(not}\ s_2\texttt{)}\ t_{i+1}\ \ldots\ t_n\texttt{)}\\
&\qquad\texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_1\ \ t_{i+1}\ \ldots\ t_n\texttt{))}\\
&\quad\texttt{(or (}r\ t_1\ \ldots\ t_{i-1}\ s_1\ \ t_{i+1}\ \ldots\ t_n\texttt{)}\\
&\qquad\texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_2\ \ t_{i+1}\ \ldots\ t_n\texttt{)))}
\end{aligned}$$

  for all sentences s_1, s_2, n-ary i-positioned propositional relations r and terms $t_j, 1 \leq j \leq n, i \neq j$. This rule embodies the following understanding of disjunctions: If someone says `(or a b)`, then we know: If he later on says `(not a)`, then he must say `b` and if he later says `(not b)`, he must say `a`. Furthermore, he must say at least one of the two. Note that in this view, saying `(or a b)` is stronger than saying either `a` or `b`. Although thoughts and utterances may be inconsistent, they usually obey the above Modus Ponens-like rule.

- Last, $\longleftrightarrow$ is a congruence relation, i.e.

$$\begin{aligned}
&s_1 \longleftrightarrow s_1\\
&s_1 \longleftrightarrow s_2 \quad\Leftrightarrow\quad s_2 \longleftrightarrow s_1\\
&s_1 \longleftrightarrow s_2 \quad\text{and}\quad s_2 \longleftrightarrow s_3 \quad\text{implies}\quad s_1 \longleftrightarrow s_3\\
&s_1 \longleftrightarrow s_2 \quad\text{implies}\quad \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_1\ t_{i+1}\ \ldots\ t_n\texttt{)}\\
&\qquad\qquad\qquad\longleftrightarrow\ \texttt{(}r\ t_1\ \ldots\ t_{i-1}\ s_2\ t_{i+1}\ \ldots\ t_n\texttt{)}
\end{aligned}$$

  for all sentences s_1, s_2, s_3, n-ary i-positioned propositional relations r and terms $t_j, 1 \leq j \leq n, j \neq i$.

Adequate Interpretations. We say that an interpretation I is *adequate for propositional attitudes*, iff

$$I \models s \quad\text{and}\quad s \longleftrightarrow s' \quad\text{implies}\quad I \models s'$$

We will only consider interpretations that are adequate for propositional attitudes. These interpretations have some nice properties, which are demonstrated here by examples:

- {`(thinks gundolf (beautiful gudrun))`,
  `(thinks gundolf (not (beautiful gudrun)))`}

  is satisfiable. Gundolf's thoughts may be inconsistent.

- {`(thinks gundolf (beautiful gudrun))`,
  `(thinks gundolf (not (beautiful gudrun)))`,
  `(not (thinks gundolf (alive elvis)))`}

  is satisfiable. Although Gundolf's thoughts are contradictory, he does not necessarily think that Elvis is alive. This example shows that, unlike an interpretation that is invariant to equivalence, an interpretation that is adequate for propositional attitudes may allow partial validity.

- `(forall (?P ?X) (=> (knows ?P ?X) (believes ?P ?X)))`
  together with
  `(knows gundolf (alive elvis))`
  entails
  `(believes gundolf (alive elvis))`

  This is a very useful property, because it allows the axiomatization of propositional attitudes.

- `(thinks gundolf (=> (alive gudrun) (beautiful gudrun)))`
  together with
  `(thinks gundolf (alive gudrun))`
  entails
  `(thinks gundolf (beautiful gudrun))`

  Although propositional arguments may be inconsistent, they must obey the Modus Ponens-like semantics of `or`.

3.3.3 Transformation of Propositional Relations

Sentences with propositional arguments shall be transformed to sentences without propositional arguments, so that they can be translated to a DLP. We assume that each propositional relation has exactly one propositional argument. This is the case for all propositional attitudes in SUMO.

Basic Sentences. We say that a relational sentence is *basic*, iff it contains none of the identifiers `and`, `or`, `<=>`, `=>`, `forall`, `exists` and no double negations. A sentence is *basic*, iff all of its relational sentences are basic. It can be seen easily that each sentence is propositionally equivalent to a basic sentence: Observe that each rule of propositional equivalence can be used to distribute an operator or a quantifier out of a relational sentence. Together with the rule $s \longleftrightarrow \text{pcnf}(s)$, operators and quantifiers inside a negation can be transferred to the outside of the negation. This rule can be used as well to eliminate double negations. Repeating this process shifts all operators and quantifiers (except for `not`) out of the propositional relations, so that a basic sentence results after finitely many iterations. We assume a function **basic** that returns such a propositionally equivalent basic sentence for a given sentence.

Transformation. The transformation first reduces the sentence to its basic sentence and to prenex conjunctive normal form:

$$\Phi(s) = \phi(\text{pcnf}(\text{basic}(s)), \texttt{mainlevel})$$

ϕ is a helper function, which distributes over all operators and quantifiers:

$$\begin{aligned}
&\phi(\texttt{(forall (}v_1\ \texttt{...}\ v_n\texttt{)}\ s), z) = \texttt{(forall (}v_1\ \texttt{...}\ v_n\texttt{)}\ \phi(s, z)\texttt{)}\\
&\phi(\texttt{(exists (}v_1\ \texttt{...}\ v_n\texttt{)}\ s), z) = \texttt{(exists (}v_1\ \texttt{...}\ v_n\texttt{)}\ \phi(s, z)\texttt{)}\\
&\phi(\texttt{(and }s_1\ \texttt{...}\ s_n\texttt{)}, z) = \qquad \texttt{(and }\phi(s_1, z)\ \texttt{...}\ \phi(s_n, z)\texttt{)}\\
&\phi(\texttt{(or }s_1\ \texttt{...}\ s_n\texttt{)}, z) = \qquad \texttt{(or }\phi(s_1, z)\ \texttt{...}\ \phi(s_n, z)\texttt{)}
\end{aligned}$$

for variables $v_1, \ldots, v_n$ and sentences $s, s_1, \ldots, s_n$. For n-ary relations r and terms $t_1, \ldots, t_n$, ϕ is defined as follows:

$$\phi(\texttt{(}r\ t_1\ \texttt{...}\ t_n\texttt{)}, z) = \texttt{(}r\ t_1\ \texttt{...}\ t_n\ z\texttt{)}$$
if no $t_1, \ldots, t_n$ is a propositional argument

$$\phi(\texttt{(not (}r\ t_1\ \texttt{...}\ t_n\texttt{))}, \texttt{mainlevel}) = \texttt{(not }\phi(\texttt{(}r\ t_1\ \texttt{...}t_n\texttt{)}, \texttt{mainlevel})\texttt{)}$$

$$\phi(\texttt{(not (}r\ t_1\ \texttt{...}\ t_n\texttt{))}, z) = \phi(\texttt{(}r\ t_1\ \texttt{...}\ t_n\texttt{)}, \texttt{(neg z)})$$
if $z \neq \texttt{mainlevel}$

$$\begin{aligned}
&\phi(\texttt{(}r\ t_1\ \texttt{...}\ t_{i-1}\ s\ t_{i+1}\ \texttt{...}\ t_n\texttt{)}, z) =\\
&\qquad \texttt{(exists (}y\texttt{) (and (}r\ t_1\ \texttt{...}\ t_{i-1}\ y\ t_{i+1}\ \texttt{...}\ t_n\ z\texttt{)}\\
&\qquad\qquad \phi(s, y)\texttt{))}
\end{aligned}$$

where y is a new variable
if s is a propositional argument

ϕ adds an argument to the relational sentences, which identifiers their context[11]. Different from a naive transformation, negated propositional arguments are not negated, but placed into a specific negation context. As a result, positive arguments are collected in a context z, whereas negative arguments go to another context `(neg` z`)`. `mainlevel` is a newly introduced constant and `neg` is a newly introduced unary function symbol.

Semantics of Modified Signature. The Φ-transformation changed the signature by adding a context argument to each relation. Hence for each interpretation I, we define a new interpretation I_ϕ, which is identical to I but behaves as follows on $(n+1)$-ary relations r and terms $t_1, \ldots, t_n$:

$I_\phi \models$ `(`$r\ t_1\ \ldots\ t_n$ `(`$r\ t_1\ \ldots\ t_n$`))`	if no $t_1, \ldots, t_n$ contains `neg` or `mainlevel`
$I_\phi \models$ `(`$r\ t_1\ \ldots\ t_n$ `mainlevel)`	if $I \models$`(`$r\ t_1\ \ldots\ t_n$`)` and no $t_1, \ldots, t_n$ contains `neg` or `mainlevel`
$I_\phi \models$ `(`$r\ t_1\ \ldots\ t_n$ `(neg (not (`$r\ t_1\ \ldots\ t_n$`))))`	if no $t_1, \ldots, t_n$ contains `neg` or `mainlevel`
$I_\phi \not\models$ `(`$r\ t_1\ \ldots\ t_n\ z$`)`	in all other cases

Theorem 1 (Equisatisfiability) *The Φ-transformation preserves equisatisfiability with respect to an interpretation I that is adequate for propositional attitudes.*

$$I \models s \Leftrightarrow I_\phi \models \Phi(s)$$

Proof: Since I is adequate for propositional attitudes, it holds

$$\begin{aligned} & I \models s \\ \Leftrightarrow\ & I \models \mathrm{pcnf}(\mathrm{basic}(s)) \end{aligned}$$

With the definition of Φ, we have to show

[11] This transformation sees subjective attributes as relations between the speaker and the described object.

$$
\begin{aligned}
& I \models \text{pcnf}(\text{basic}(s)) \\
\Leftrightarrow\ & I_\phi \models \phi(\text{pcnf}(\text{basic}(s)), \texttt{mainlevel})
\end{aligned}
$$

Since ϕ distributes over all operators and quantifiers, we can apply structural induction and it is enough to show that

$$
\begin{aligned}
& I \models \texttt{(}r\ a_1\ \texttt{...}\ a_k\texttt{)} \\
\Leftrightarrow\ & I_\phi \models \phi(\texttt{(}r\ a_1\ \texttt{...}\ a_k\texttt{)}, \texttt{mainlevel})
\end{aligned}
$$

for all relations r and terms $a_1, \ldots, a_k$. If none of the $a_1, \ldots, a_k$ is a propositional argument, then

$$\phi(\texttt{(}r\ a_1\ \texttt{...}\ a_k\texttt{)}, \texttt{mainlevel}) = \texttt{(}r\ a_1\ \texttt{...}\ a_k\ \texttt{mainlevel)}$$

By the definition of I_ϕ, equisatisfiability follows immediately. Now consider the case where one a_i is a propositional argument (there may be at most one). To ease the writing, we always assume, without loss of generality, that a propositional argument is the last argument, i.e. a_k is the propositional argument. We treat the case where a_k contains again a propositional argument and this argument contains again one etc. Assume that a_k does not contain negations. Since the relational sentence is basic, it must be of the form

$$s = \texttt{(}r_1\ \vec{t_1}\ \texttt{(}r_2\ \vec{t_2}\ \texttt{(...(}r_n\ \vec{t_n}\ \texttt{(}q\ \vec{t_{n+1}}\texttt{))...)))}$$

where $r_1, \ldots, r_n$ are propositional relations, q is a relation symbol and $\vec{t_1}, \ldots, \vec{t_{n+1}}$ are sequences of terms. Observe that none of the $\vec{t_1}, \ldots, \vec{t_{n+1}}$ can be a propositional argument and hence none of them can contain nested propositional arguments. We prove equisatisfiability of s and $\phi(s, \texttt{mainlevel})$ by a sequence of equivalence transformations:

$$I_\phi \models \phi(s, \texttt{mainlevel})$$

By the construction of ϕ, this is equivalent to

$$
\begin{aligned}
I_\phi \models & \texttt{(exists (}x_1\texttt{) (and (}r_1\ \vec{t_1}\ x_1\ \texttt{mainlevel)} \\
& \quad \texttt{(exists (}x_2\texttt{) (and (}r_2\ \vec{t_2}\ x_2\ x_1\texttt{)} \\
& \quad\ \texttt{(exists (}x_3\texttt{) (and (}r_3\ \vec{t_3}\ x_3\ x_2\texttt{)} \\
& \qquad \vdots \\
& \qquad \texttt{(exists (}x_{n-1}\texttt{) (and (}r_{n-1}\ \vec{t_{n-1}}\ x_{n-1}\ x_{n-2}\texttt{)} \\
& \qquad\ \texttt{(exists (}x_n\texttt{) (and (}r_n\ \vec{t_n}\ x_n\ x_{n-1}\texttt{)} \\
& \qquad\qquad\qquad \texttt{(}q\ \vec{t_{n+1}}\ x_n\texttt{)))))...))))))}
\end{aligned}
$$

By definition of I_ϕ, this is equivalent to

$$\begin{array}{l}\text{There exists an } x_1 \text{ s.t. } I \models (r_1\ \vec{t_1}\ x_1) \text{ and}\\ \quad\text{there exists an } x_2 \text{ s.t. } x_1 = (r_2\ \vec{t_2}\ x_2) \text{ and}\\ \quad\quad\text{there exists an } x_3 \text{ s.t. } x_2 = (r_3\ \vec{t_3}\ x_3) \text{ and}\\ \quad\quad\quad\vdots\\ \quad\quad\quad\text{there exists an } x_{n-1} \text{ s.t. } x_{n-2} = (r_{n-1}\ \vec{t_{n-1}}\ x_{n-1}) \text{ and}\\ \quad\quad\quad\quad\text{there exists an } x_n \text{ s.t. } x_{n-1} = (r_n\ \vec{t_n}\ x_n) \text{ and}\\ \quad\quad\quad\quad x_n = (q\ \vec{t_{n+1}})\end{array}$$

This is equivalent to

$$I \models (r_1\ \vec{t_1}\ (r_2\ \vec{t_2}\ (\ldots(r_n\ \vec{t_n}\ (q\ \vec{t_{n+1}}))\ldots)))$$

This is nothing else than

$$I \models s$$

This proof can be generalized easily to the case where some propositional arguments occur negated. □

Pragmatic Considerations. It turned out that a literal Φ-transformation is inadequate for some relations that are supposed to hold in all contexts (such as for example `equal` and `instance`). For these relations, the additional argument is simply omitted. Furthermore, some rules should hold in all contexts (such as for instance the rules for meta-identifiers, as listed in 3.1.2). Hence these rules are translated with a variable in place of the context argument.

Besides the propositional attitudes, there is one other important propositional relation in SUMO: The relation `holdsDuring`. It takes two arguments, a time interval and a sentence, and it means that the sentence holds during the time interval. `holdsDuring` can be transformed like the propositional attitudes, but the following SUMO axiom simplifies the transformation:

```
(=> (holdsduring ?TIME (not ?SENTENCE))
    (not (holdsduring ?TIME ?SENTENCE)))
```

It turned out that, in practice, the conclusion of this axiom suffices as a translation of `(holdsduring ?TIME (not ?SENTENCE))`, so that the algorithm uses the following pragmatic modification of ϕ:

ϕ(`(holdsDuring` t `(not` s`))`,C) = `(not` ϕ(`(holdsDuring` t s`)`,C)`)`

for all terms t and sentences s. Furthermore, propositional arguments do not contain nested propositional arguments in the part of SUMO under consideration. This allows further simplifications of the algorithm.

3.4 Translation to a DLP

After all of the above transformation steps have been applied, the set of KIF sentences contains no meta-identifiers, no class-creating functions and no propositional arguments. Observe that it may still contain non-class-creating functions. The translation to a DLP proceeds in two steps: First, the KIF sentences are normalized and skolemized. Then, DLP-rules are generated.

3.4.1 Normalization and Skolemization

The KIF-sentences are transformed to prenex conjunctive normal form, i.e. each sentence takes the form

$$
\begin{array}{l}
(Q_1\ (v_{1,1}\ \ldots\ v_{1,n_1}) \\
\ (Q_2\ (v_{2,1}\ \ldots\ v_{2,n_2}) \\
\quad \vdots \\
\quad (Q_m\ (v_{m,1}\ \ldots\ v_{m,n_m})\ (\texttt{and}\ (\texttt{or}\ s_{1,1}\ \ldots\ s_{1,k_1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{or}\ s_{2,1}\ \ldots\ s_{2,k_2}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{or}\ s_{l,1}\ \ldots\ s_{l,k_l})))\ldots))
\end{array}
$$

where $Q_1, \ldots, Q_m \in \{\texttt{exists}, \texttt{forall}\}$, the $v_{i,j}, 1 \leq i \leq m, 1 \leq j \leq n_m$, are variables and the $s_{i,j}, 1 \leq i \leq l, 1 \leq j \leq k_l$, are relational sentences that may be negated. Then, each sentence is *skolemized* by replacing each existentially quantified subsentence as follows

$$
\begin{array}{ll}
 & (\texttt{exists}\ (v_1\ \ldots\ v_n)\ s) \\
\rightsquigarrow & s\{(v_i, (f_i\ x_1\ \ldots\ x_m)) | 1 \leq i \leq n\}
\end{array}
$$

where $x_1, \ldots, x_m$, are the variables bound by `forall` quantifiers preceding the `exists` and $f_1, \ldots, f_n$ are fresh function symbols. It is well known from First Order Logic that this transformation preserves satisfiability.

3.4.2 Generation of DLP Rules

We omit the universal quantifiers, so that the sentences are completely quantifier-free. We rearrange the disjunctions so that the positive relational sentences pre-

cede the negated ones. Then, each disjunction can be translated to a DLP-rule as follows:

$$\begin{array}{ll} & \texttt{(or } s_1 \texttt{ ... } s_k \texttt{ (not } s_{k+1}\texttt{) ... (not } s_n\texttt{))} \\ \rightsquigarrow & d(s_1) \texttt{ ; ...; } d(s_k) \texttt{ :- } d(s_{k+1})\texttt{, ..., } d(s_n)\texttt{.} \end{array}$$

where d is a function that converts KIF symbols to DLP symbols, KIF variables to DLP variables and relational sentences to atoms.

Equisatisfiability. A set of KIF sentences is satisfiable iff its translation to a DLP is satisfiable. To see this, assume that S is the set of KIF sentences and P is the resulting DLP. Be I an interpretation for S with $I \models S$. We construct a model M_I from I:

$$M_I = \{d(s) | s \text{ is a relational sentence with } I \models s\}$$

Consider each sentence in S and each disjunction D in its prenex conjunctive normal form.

$$\begin{array}{ll} D\texttt{:} & \texttt{(or } s_1 \texttt{ ... } s_k \texttt{ (not } s_{k+1}\texttt{) ... (not } s_n\texttt{))} \\ \text{Translation of } D\texttt{:} & d(s_1) \texttt{ ; ...; } d(s_k) \texttt{ :- } d(s_{k+1})\texttt{, ..., } d(s_n)\texttt{.} \end{array}$$

Since $I \models S$, it follows $I \models D$. We distinguish two cases: Either there is one $s_i, k+1 \leq i \leq n$, with $I \models \texttt{(not } s_i\texttt{)}$. Then the rule is not applicable in M_I. If there is no such s_i, then there must be an $s_j, 1 \leq j \leq k$ with $I \models s_j$. Hence the rule is satisfied in M_I. Since these considerations apply to all rules, $M \models P$. Similarly, one can construct an interpretation I_M for any model M of P with $I_M \models S$.

3.5 Equality

Equality is one of the most basic relations in SUMO. It is indispensable whenever functional terms are involved. However, equality is not built-in in the semantics of DLPs. Consequently, a special treatment of equality is necessary.

3.5.1 Equality in SUMO

The Relation 'equal'. SUMO knows the relation `equal` to express equality between entities. `equal` is declared as an equivalence relation. Additional axioms make sure that two `equal` terms can be exchanged in a sentence without affecting its truth value, so that `equal` is in fact a congruence relation. Since

these axioms involve variables ranging over relations, they cannot be expressed directly in a DLP.

Applications of 'equal'. `equal` mainly serves three purposes in SUMO:

1. It is used to claim that two things are different, as in

```
(=> (instance ?KISS Kissing)
    (exists (?PERSON1 ?PERSON2 ?LIP1 ?LIP2)
            (and (agent ?KISS ?PERSON1)
                 (agent ?KISS ?PERSON2)
                 (not (equal ?PERSON1 ?PERSON2))
                 ⋮
                 (meetsSpatially ?LIP1 ?LIP2))))
```

2. It defines function results, as in

```
(equal (CardinalityFn Continent) 7)
```

3. It is used to state that only one entity has a property, as in

```
(=> (instance ?BACTERIUM Bacterium)
    (exists (?CELL1)
            (and (component ?CELL1 ?BACTERIUM)
                 (instance ?CELL1 Cell)
                 (forall (?CELL2)
                  (=> (and (component ?CELL2 ?BACTERIUM)
                           (instance ?CELL2 Cell))
                      (equal ?CELL1 ?CELL2))))))
```

3.5.2 Equality in DLPs

E-Interpretations. We define DLP interpretations that give the intended semantics to the predicate `equal`: An *E-interpretation* is a (Herbrand) interpretation I that maps `equal` to a congruence relation on terms. This means that for all m-ary function symbols f, n-ary predicate symbols p and terms $a_1, a_2, a_3, t_1, \ldots, t_n$:

$$I \models \texttt{equal}(a_1, a_1)$$

$$\begin{array}{ll} & I \models \texttt{equal}(a_1, a_2) \\ \Leftrightarrow & I \models \texttt{equal}(a_2, a_1) \end{array}$$

$$\begin{array}{ll} & I \models \texttt{equal}(a_1, a_2) \\ \text{and} & I \models \texttt{equal}(a_2, a_3) \\ \text{implies} & I \models \texttt{equal}(a_1, a_3) \end{array}$$

$$\begin{array}{ll} & I \models \texttt{equal}(a_1, a_2) \\ \text{and} & I \models p(t_1, \ldots, t_{i-1}, a_1, t_{i+1}, \ldots, t_n) \\ \text{implies} & I \models p(t_1, \ldots, t_{i-1}, a_2, t_{i+1}, \ldots, t_n) \end{array}$$

$$\begin{array}{ll} & I \models \texttt{equal}(a_1, a_2) \\ \text{implies} & I \models \texttt{equal}(f(t_1, \ldots, t_{i-1}, a_1, t_{i+1}, \ldots, t_n), \\ & \qquad\qquad f(t_1, \ldots, t_{i-1}, a_2, t_{i+1}, \ldots, t_n)) \end{array}$$

The *equality closure* of a set M of atoms, denoted $M^\star$ is the smallest superset of M that defines an E-interpretation.

The Unique Name Assumption. In SUMO, `equal` is never used to establish equality between constants. This suggests to uphold the unique name assumption, i.e. we will assume that two different constants are never equal. This assumption is quite reasonable in the context of ontologies, where distinct identifiers (like `Gudrun` and `Gundolf`) are mostly supposed to mean different entities. Formally speaking, a DLP interpretation I *satisfies the Unique Name Assumption (UNA)*, iff for all distinct constants c_1, c_2, $I \not\models \texttt{equal}(c_1, c_2)$. Note that since we distinguish constants and 0-ary functions, we can easily declare `gundolf` a 0-ary function if we wish that `equal(gundolf,gudrun)` be satisfiable. E-interpretations that satisfy the UNA are called *UNA-E-interpretations*.

Known Equality Transformations. Our goal is to find a transformation from a DLP P to a DLP $P^{\texttt{eq}}$, such that there exists a model $M^{\texttt{eq}}$ with $M^{\texttt{eq}} \models P^{\texttt{eq}}$, iff there exists a UNA-E-model M with $M \models P$. In other words: We are compiling away the constraint that the intended interpretation of P is a UNA-E-interpretation. A naive transformation would simply add the above axioms for E-interpretations. However, this causes the search space to explode, especially

through the axioms of the last pattern, the *substitution axioms*. Several other transformations have been proposed (see for example [Bra75], which was later improved in [BGV97]). We will see a simpler transformation, which bases on the UNA.

3.5.3 Equality Transformation

Equivalence Axioms. The transformation follows the naive transformation by adding the equivalence axioms of `equal`:

```
equal(X,X).
equal(X,Y) :- equal(Y,X).
equal(X,Z) :- equal(X,Y), equal(Y,Z).
```

However, the substitution axioms are not added. Instead, the transformation relies on *flat rules*. A rule is called *flat* iff

1. the arguments of `equal` are terms, the only proper subterms of which are either variables or constants
2. all arguments to other predicate symbols are either variables or constants.

Flattening. Every rule can be turned into a flat one by recursively replacing an offending subterm t by a fresh variable x and adding the atom `equal(`t`,`x`)` to the rule body (see again [BGV97]). This is exemplified here:

```
    p(f(a)) :- r(g(h(d),e)).
⇝   p(X1) :- r(g(h(d),e)),
             equal(f(a),X1).
⇝   p(X1) :- r(X2),
             equal(f(a),X1),
             equal(X2,g(h(d),e)).
⇝   p(X1) :- r(X2),
             equal(f(a),X1),
             equal(X2,g(X3,e)),
             equal(X3,h(d)).
```

We will see that flat rules can play the role of the substitution axioms.

UNA Axioms. The Unique Name Assumption has to be axiomatized. This could be done by adding rules of the form `false :- equal(`c_1`,`c_2`)` for all distinct constants c_1, c_2. To abbreviate this procedure, the system translates KIF function terms of the form `(`f t_1 ... t_n`)` to DLP terms of the form `fun(`$f, t_1, \ldots, t_n$`)`. Then the UNA can be axiomatized by

```
identical(X,X).
false :- equal(X,Y),
         not(identical(X,Y)),
         not(isFunctionTerm(X)),
         not(isFunctionTerm(Y)).
isFunctionTerm(fun(X1)).
⋮
isFunctionTerm(fun(X1,...,X(n + 1))).
```

where n is the maximum function symbol arity in SUMO (we know that $n \leq 5$).

Theorem 2 (Equisatisfiability) *Let P be a positive disjunctive logic program and $P^{\mathtt{eq}}$ the result of the transformation on P. If there exists a UNA-E-model for P, then there exists a model for $P^{\mathtt{eq}}$ (soundness). If there exists a model for $P^{\mathtt{eq}}$, then there exists a UNA-E-model for P (completeness).*

Proof: We use the following formalization of the substitution axiom: If M is an E-model, then for all atoms a, variables X and terms t_1, t_2:

$$\begin{array}{ll} & M \models a\{(X, t_1)\} \\ \text{and} & M \models \mathtt{equal(}t_1\mathtt{,}t_2\mathtt{)} \\ \text{implies} & M \models a\{(X, t_2)\} \end{array}$$

Be M a UNA-E-model for P. Then M is also a model for $P^{\mathtt{eq}}$. To see this, consider each rule $C^{\mathtt{eq}}$ of $P^{\mathtt{eq}}$. If $C^{\mathtt{eq}}$ is one of the equality axioms or the UNA axiomatization, then M must satisfy $C^{\mathtt{eq}}$ because M is a UNA-E-model. If $C^{\mathtt{eq}}$ stems from a rule C, then we have to show $M \models C^{\mathtt{eq}}\gamma$ for any ground substitution γ. Pick any such γ. C and $C^{\mathtt{eq}}$ are of the form:

$$\begin{array}{lll} C & = H_1 \ ; \ \ldots ; \ H_m & \mathtt{:-}\ B_1\mathtt{,}\ \ldots\mathtt{,}\ B_n\mathtt{.} \\ C^{\mathtt{eq}} & = H'_1 \ ; \ \ldots ; \ H'_m & \mathtt{:-}\ B'_1\mathtt{,}\ \ldots\mathtt{,}\ B'_n\mathtt{,} \\ & & \quad \mathtt{equal(}X_1\mathtt{,}u_1\mathtt{),} \\ & & \quad \vdots \end{array}$$

$$\texttt{equal}(X_k,u_k).$$

where $H_1,\ldots,H_m$ and $B_1,\ldots,B_n$ are atoms, $H'_1,\ldots,H'_m$ and $B'_1,\ldots,B'_n$ are the corresponding flat atoms, $X_1,\ldots,X_k$ are (fresh) variables and $u_1,\ldots,u_k$ are terms such that the equations are flat. Due to the construction of the transformation, we may assume, without loss of generality, an ordering of the equations, such that no $u_j, 1 \leq j \leq k$, contains any X_l with $1 \leq l \leq j$. If $C^{\texttt{eq}}\gamma$ is not applicable in M, then $M \models C^{\texttt{eq}}\gamma$.

Now consider the case where $C^{\texttt{eq}}\gamma$ is applicable in M, i.e. $M \models B'_i\gamma$ for all $1 \leq i \leq n$ and $M \models \texttt{equal}(X_j, u_j)\gamma$ for all $1 \leq j \leq k$. We define a substitution σ that, intuitively speaking, undoes the effect of the flattening:

$$\begin{aligned}
&\sigma_1 = \{(X_1, u_1)\}\\
&\vdots\\
&\sigma_k = \{(X_k, u_k)\}\\
&\sigma = \sigma_1\sigma_2\ldots\sigma_k
\end{aligned}$$

By the construction of the transformation, it is easy to see that $H_i = H'_i\sigma$ for $1 \leq i \leq m$ and $B_j = B'_j\sigma$ for $1 \leq j \leq n$. We split γ to

$$\gamma = \gamma' \cup \textstyle\bigcup_{1\leq j\leq k} \gamma_j$$

where $\gamma_j = \{(X_j, X_j\gamma)\}$.

Our goal is to show that $C\gamma'$ is applicable in M, i.e. we have to prove $M \models B_i\gamma'$ for all $1 \leq i \leq n$. Pick any $i, 1 \leq i \leq n$. We show $M \models B'_i\sigma\gamma'$, which is the same as $M \models B_i\gamma'$. We prove by induction that

$$\begin{aligned}
&M \models \texttt{equal}(X_j\gamma_j,\ X_j\sigma_j\ldots\sigma_k)\gamma'\\
&M \models B'_i\ \ \gamma_1\ldots\gamma_{j-1}\ \ \sigma_j\ldots\sigma_k\ \ \gamma'
\end{aligned}$$

for all $j, 1 \leq j \leq k$. The borderline case, $j = 1$, entails our goal $M \models B'_i\sigma\gamma'$. First observe that no $X_j\gamma_j$ contains variables, since γ is ground for $C^{\texttt{eq}}$. Furthermore, we already saw that, by the ordering of the equations, $X_j\sigma_j$ cannot contain any variable X_l with $1 \leq l \leq j$. For $j = k$, the first formula

$$M \models \texttt{equal}(X_k\gamma_k,\ X_k\sigma_k)\gamma'$$

follows immediately from $M \models \texttt{equal}(X_k,\ u_k)\gamma$. The second formula follows from

$$
\begin{array}{ll}
& M \models B'_i \ \gamma \\
\Leftrightarrow & M \models B'_i \ \gamma_1 \ldots \gamma_{k-1} \ \gamma_k \ \gamma' \\
& \qquad\qquad\qquad\qquad \text{with } M \models \texttt{equal}(X_k\gamma_k, \ X_k\sigma_k)\gamma' \\
\Rightarrow & M \models B'_i \ \gamma_1 \ldots \gamma_{k-1} \ \sigma_k \ \gamma'
\end{array}
$$

Now assume that both formulae hold for $j, j+1, \ldots k$ with $1 < j \leq k$. We show that they also hold for $j-1$. We know

$$
\begin{array}{ll}
& M \models \texttt{equal}(X_{j-1} \quad , \ u_{j-1} \qquad\qquad)\gamma \\
\Leftrightarrow & M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ u_{j-1} \qquad \gamma_1 \ldots \gamma_k)\gamma' \\
\Leftrightarrow & M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ \gamma_1 \ldots \gamma_k)\gamma' \\
\Leftrightarrow & M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ \gamma_j \ldots \gamma_k)\gamma' \\
\Leftrightarrow & M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ \{(X_j, X_j\gamma_j)\} \ \gamma_{j+1} \ldots \gamma_k)\gamma' \\
& \qquad\qquad\qquad\qquad \text{with the induction hypothesis} \\
& \qquad\qquad\qquad\qquad M \models \texttt{equal}(X_j\gamma_j, \ X_j\sigma_j \ldots \sigma_k)\gamma' \\
\Rightarrow & M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ \{(X_j, X_j\sigma_j \ldots \sigma_k)\} \ \gamma_{j+1} \ldots \gamma_k)\gamma'
\end{array}
$$

The induction hypothesis can also be applied to all $X_l\gamma_l$ with $j < l \leq k$, so that

$$M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ \{(X_j, X_j\sigma_j \ldots \sigma_k)\} \ldots \{(X_k, X_k\sigma_k)\})\gamma'$$

This boils down to

$$M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ldots \sigma_k)\gamma'$$

This is the first formula we had to show. The second one follows from the induction hypothesis

$$
\begin{array}{ll}
& M \models B'_i \ \gamma_1 \ldots \gamma_{j-2} \ \gamma_{j-1} \qquad\qquad\qquad \sigma_j \ldots \sigma_k \ \gamma' \\
\Leftrightarrow & M \models B'_i \ \gamma_1 \ldots \gamma_{j-2} \ \{(X_{j-1}, X_{j-1}\gamma_{j-1})\} \ \sigma_j \ldots \sigma_k \ \gamma' \\
& \qquad\qquad \text{with} \quad M \models \texttt{equal}(X_{j-1}\gamma_{j-1}, \ X_{j-1}\sigma_{j-1} \ldots \sigma_k)\gamma' \\
\Rightarrow & M \models B'_i \ \gamma_1 \ldots \gamma_{j-2} \ \{(X_{j-1}, X_{j-1}\sigma_{j-1} \ldots \sigma_k)\} \ \sigma_j \ldots \sigma_k \ \gamma' \\
\Leftrightarrow & M \models B'_i \ \gamma_1 \ldots \gamma_{j-2} \ \sigma_{j-1} \ldots \sigma_k \qquad\qquad\qquad\qquad\qquad \gamma'
\end{array}
$$

This completes the induction, so that

$$
\begin{array}{l}
M \models \texttt{equal}(X_j\gamma_j, \ X_j\sigma_j \ldots \sigma_k)\gamma' \\
M \models B'_i \ \gamma_1 \ldots \gamma_{j-1} \ \sigma_j \ldots \sigma_k \ \gamma'
\end{array}
$$

for all $j, 1 \leq j \leq k$. For the case $j = 1$, it follows $M \models B'_i\sigma\gamma'$. Hence $M \models B_i\gamma'$ for all $1 \leq i \leq n$ and $C\gamma'$ is applicable in M.

As $M \models P$ and $C\gamma'$ is ground, M must satisfy $C\gamma'$. Hence there is a H_j with $M \models H_j\gamma'$. This entails

$$
\begin{array}{ll}
 & M \models H_j \ \gamma' \\
\Leftrightarrow & M \models H'_j \ \sigma \ \gamma' \\
\Leftrightarrow & M \models H'_j \ \{(X_1, X_1\sigma_1 \ldots \sigma_k)\} \ldots \ \{(X_k, X_k\sigma_k)\} \ \gamma' \\
 & \qquad \text{with} \quad M \models \texttt{equal}(X_l\gamma_l, \ X_l\sigma_l \ldots \sigma_k)\gamma' \\
 & \qquad \quad \text{for all} \ \ 1 \leq l \leq k \\
\Rightarrow & M \models H'_j \ \gamma_1 \ldots \gamma_k \ \gamma'
\end{array}
$$

Hence $M \models H'_j\gamma$, $M \models C^{\texttt{eq}}\gamma$ and $M \models P^{\texttt{eq}}$.

Now assume that there is a model M of $P^{\texttt{eq}}$. We will construct a subset R of M. Then we prove[12]

1. $R^\star \models P^{\texttt{eq}}$
2. $R^\star \models P$
3. $R^\star$ is a UNA-E-model of P

We start with constructing R. We need a *reduction ordering* on ground terms, i.e. a total binary relation $\succ$ on ground terms such that

- $\succ$ is irreflexive.
- $\succ$ is asymmetric.
- $\succ$ is transitive.
- $\succ$ is *well-founded* on terms, i.e. starting from a given term, there is no infinite sequence of terms, in which each term is smaller than its predecessor in the sense of $\succ$.
- $t_i \succ t'_i$ implies $f(t_1, \ldots, t_i, \ldots, t_n) \succ f(t_1, \ldots, t'_i, \ldots, t_n)$ for any n-ary function symbol f and terms $t_1, \ldots, t_n$.

[12] This proof follows the proof in [BS05]. It uses the model construction technique introduced for super-position calculi (see [BG98]).

We assume a constant **true** that does not occur in P and we define $t \succ \mathbf{true}$ for all terms. Furthermore, $\succ$ shall be chosen such that for any constant c and term t, if $c \succ t$, then t is a constant, too. We extend $\succ$ to multisets by defining $S_1 \succ S_2$ iff $\forall s_2 \in S_2 : \exists s_1 \in S_1 : s_1 \succ s_2$. From now on, we will read an equation $\texttt{equal(}t_1\texttt{,}t_2\texttt{)}$ as the multiset $\{t_1, t_2\}$. Hence $\succ$ is defined on equations. Next, we read each atom $p\texttt{(}t_1, \ldots, t_n\texttt{)}$ with $p \neq \texttt{equal}$ as an equation $\texttt{equal(}p\texttt{(}t_1, \ldots, t_n\texttt{), true)}$. Hence $\succ$ is defined on atoms. We extend $\succ$ to rules by defining $C_1 \succ C_2$ iff the multiset of all atoms appearing in C_1 is greater than the multiset of all atoms appearing in C_2 in the sense of $\succ$. Hence $\succ$ is defined on all components of a DLP.

Now, we construct a rewrite system R from M, with initially $R = \emptyset$. We walk through all atoms of M in the order of $\succ$, starting from the smallest atom. By the above interpretation, every atom can be considered to be of the form $\texttt{equal(}t_1, t_2\texttt{)}$. If $t_1 \not\succ t_2$ or t_1 is already reducible in the current R or t_2 is reducible in the current R, we proceed to the next atom. Else, we add $t_1 \longrightarrow t_2$ to R. By construction, R has no critical pairs. Since $\succ$ is a well-founded ordering, R is a convergent rewrite-system.

By reading $t_1 \longrightarrow t_2$ as $\texttt{equal(}t_1\texttt{, }t_2\texttt{)}$, and $\texttt{equal(}t_1\texttt{, true)}$ as an atom, we can say that R is a subset of M. We will show that the equality closure of R, $R^\star$, is a UNA-E-model of P.

First, we show that $R^\star \models P^{\texttt{eq}}$. If $R^\star$ is to be a model of $P^{\texttt{eq}}$, it must satisfy each rule $C^{\texttt{eq}}$ in $R^\star$. Hence it must satisfy all ground instances $C^{\texttt{eq}}\gamma$ of $C^{\texttt{eq}}$, where γ is a ground substitution for $C^{\texttt{eq}}$. Pick any rule $C^{\texttt{eq}}$ from $P^{\texttt{eq}}$ and any ground substitution γ for $C^{\texttt{eq}}$. We prove $R^\star \models C^{\texttt{eq}}\gamma$ by well-founded induction on γ.

Consider the base case: There is no variable X in $C^{\texttt{eq}}$, such that $X\gamma \longrightarrow_R t$ for some term t. We say that γ is *irreducible* for $C^{\texttt{eq}}$ (in R). Be H the set of head atoms of $C^{\texttt{eq}}\gamma$ and be B the set of body atoms. If $R^\star \not\models B$, then $C^{\texttt{eq}}\gamma$ is not applicable in $R^\star$ and $R^\star \models C^{\texttt{eq}}\gamma$ follows trivially. Hence suppose $R^\star \models B$. Since γ is irreducible for $C^{\texttt{eq}}$, it follows that γ is irreducible for each body atom in B. Lemma 1 implies that $M \models B\gamma$. Since $M \models C^{\texttt{eq}}$, it follows that there exists a head atom $h \in H$ with $h\gamma \in M$. Again by lemma 1, this time in the other direction, it follows that $R^\star \models h\gamma$ and hence $R^\star \models C^{\texttt{eq}}$.

Now assume that $R^\star \models C^{\texttt{eq}}\gamma'$ for all γ' with $\text{range}(\gamma) \succ \text{range}(\gamma')$. If the base case does not apply, then there is a variable X occurring in $C^{\texttt{eq}}$ and there is a term t, such that $X\gamma \longrightarrow_R t$. Be $\gamma' = \{(x, x\gamma) | x \neq X\} \cup \{(X, t)\}$, i.e. γ' is the "shortcut" that maps the variable directly to the reduction in R. Since

$X\gamma \longrightarrow_R t$ implies that $X\gamma \succ t$, it follows that $C^{\texttt{eq}}\gamma \succ C^{\texttt{eq}}\gamma'$. By the induction hypothesis, $R^\star \models C^{\texttt{eq}}\gamma'$. Since $X\gamma \longrightarrow_R t$ implies that $\texttt{equal}(X\gamma,t) \in R^\star$, we can conclude $R^\star \models C^{\texttt{eq}}\gamma$.

We proceed to the second part of the proof and show that $R^\star \models P$. We have to show that $R^\star \models C\gamma$ for every rule $C \in P$ and any ground substitution γ of C. Pick any rule $C \in P$ and any ground substitution γ for C. Be $C^{\texttt{eq}}$ the rule that results from C by the equality transformation. Then C and $C^{\texttt{eq}}$ are of the following forms:

$$
\begin{array}{lll}
C & \texttt{=}\ H_1\ \texttt{;}\ \ldots\texttt{;}\ H_m\ \texttt{:-}\ B_1\texttt{,}\ \ldots\texttt{,}\ B_n\texttt{.} \\
C^{\texttt{eq}} & \texttt{=}\ H'_1\ \texttt{;}\ \ldots\texttt{;}\ H'_m\ \texttt{:-}\ B'_1\texttt{,}\ \ldots\texttt{,}\ B'_n\texttt{,} \\
 & & \texttt{equal(}X_1\texttt{,}u_1\texttt{),} \\
 & & \vdots \\
 & & \texttt{equal(}X_k\texttt{,}u_k\texttt{).}
\end{array}
$$

where $H_1, \ldots, H_m$ and $B_1, \ldots, B_n$ are atoms, $H'_1, \ldots, H'_m$ and $B'_1, \ldots, B'_n$ are the corresponding flat atoms, $X_1, \ldots, X_k$ are (fresh) variables and $u_1, \ldots, u_k$ are terms such that the equations are flat.

By the preceding part of the proof, we know that $R^\star \models P^{\texttt{eq}}$ and hence $R^\star \models C^{\texttt{eq}}$. As before, we define the substitution σ as

$$
\begin{aligned}
&\sigma_1 = \{(X_1, u_1)\} \\
&\vdots \\
&\sigma_k = \{(X_k, u_k)\} \\
&\sigma = \sigma_1\sigma_2 \ldots \sigma_k
\end{aligned}
$$

We already saw that by the construction of the equality transformation, $H'_i\sigma = H_i$ for $1 \leq i \leq m$ and $B'_j\sigma = B_j$ for $1 \leq j \leq n$. Hence $C^{\texttt{eq}}\sigma$ is of the form

$$
\begin{array}{lll}
C^{\texttt{eq}}\sigma & \texttt{=}\ H_1\ \texttt{;}\ \ldots\texttt{;}\ H_m\ \texttt{:-}\ B_1\texttt{,}\ \ldots\texttt{,}\ B_n\texttt{,} \\
 & & \texttt{equal(}X_1\texttt{,}u_1\texttt{)}\sigma\texttt{,} \\
 & & \vdots \\
 & & \texttt{equal(}X_k\texttt{,}u_k\texttt{)}\sigma\texttt{.}
\end{array}
$$

Since σ eliminated all variables introduced by the equality transformation, γ is a ground substitution for $C^{\texttt{eq}}\sigma$. Since $R^\star \models C^{\texttt{eq}}$, $R^\star \models C^{\texttt{eq}}\sigma\gamma$. If an interpretation is a model for a positive rule, then it is also a model for the rule, if some of the body atoms are omitted. Hence

$$
\begin{array}{lll}
 & R^\star \models C^{\texttt{eq}}\sigma\gamma & \\
\Leftrightarrow & R^\star \models H_1\gamma \ ; \ \ldots; \ H_m\gamma \ \texttt{:-} & B_1\gamma, \ \ldots, \ B_n\gamma, \\
 & & \texttt{equal(}X_1\texttt{,}u_1\texttt{)}\sigma\gamma, \\
 & & \vdots \\
 & & \texttt{equal(}X_k\texttt{,}u_k\texttt{)}\sigma\gamma. \\
\Rightarrow & R^\star \models H_1\gamma \ ; \ \ldots; \ H_m\gamma \ \texttt{:-} & B_1\gamma, \ \ldots, \ B_n\gamma. \\
\Leftrightarrow & R^\star \models C\gamma &
\end{array}
$$

In the last part of the proof, it remains to show that $R^\star$ is a UNA-E-model for P. As $R^\star$ is the equality closure of R, it is by definition an E-interpretation. Now assume that $R^\star$ does not satisfy the UNA, i.e. $R^\star \models \texttt{equal(}c\texttt{,}d\texttt{)}$ for some distinct constants c and d. $\texttt{equal(}c\texttt{,}d\texttt{)}$ is flat and hence, by lemma 1, $M \models \texttt{equal(}c\texttt{,}d\texttt{)}$. But since $M \models P^{\texttt{eq}}$, M must also satisfy the axiomatization of the UNA in $P^{\texttt{eq}}$, which disallows $M \models \texttt{equal(}c\texttt{,}d\texttt{)}$. Hence $M \not\models \texttt{equal(}c\texttt{,}d\texttt{)}$, $R^\star \not\models \texttt{equal(}c\texttt{,}d\texttt{)}$ and $R^\star$ is a UNA-E-model[13]. □

Lemma 1 *Let* $\texttt{equal(}s\texttt{, }t\texttt{)}$ *be flat equation and let* γ *be a ground substitution irreducible for* $\texttt{equal(}s\texttt{, }t\texttt{)}$ *in* R. *Then,* $R^\star \models \texttt{equal(}s\gamma\texttt{, }t\gamma\texttt{)}$ *iff* $M \models \texttt{equal(}s\gamma\texttt{, }t\gamma\texttt{)}$.

Proof: For the if-direction suppose $\texttt{equal(}s\gamma\texttt{, }t\gamma\texttt{)} \in M$. We consider various cases, which together constitute an exhaustive case analysis.

If $s\gamma = t\gamma$ then $R^\star \models \texttt{equal(}s\gamma\texttt{,}t\gamma\texttt{)}$ follows trivially.

If both $s\gamma$ and $t\gamma$ are irreducible by R, then $\texttt{equal(}s\gamma\texttt{,}t\gamma\texttt{)}$ was added to R from M during the construction of R. Hence $R \models \texttt{equal(}s\gamma\texttt{, }t\gamma\texttt{)}$ and $R^\star \models \texttt{equal(}s\gamma\texttt{, }t\gamma\texttt{)}$.

Hence suppose, without loss of generality, that $s\gamma$ is reducible in R. Then $s\gamma$ cannot be a constant. To see this, recall first that M is a possible model for a program that was obtained by the equality transformation. The transformation adds an axiom to ensure that no model can contain $\texttt{equal(}c\texttt{, }d\texttt{)}$ for any two constants c, d. Hence M cannot contain such an equation and neither can R, which is a subset of M. Since by the definition of $\succ$, $c \succ t$ is only possible for a constant c if t is a constant as well, it follows that R does not contain any rule of the form $c \to t$. Hence constants are irreducible by R.

[13] Since $R \subseteq M$, $M^\star$ gives an upper bound for the smallest set of atoms that constitutes an UNA-E-model of P.

By this consideration, $s\gamma$ cannot be a constant. The term $s\gamma$ thus is of the form $f(\vec{v})\gamma$ where f is some (possibly nullary) function symbol and $\vec{v}$ is some list of terms. More specifically, as `equal(`s`, `t`)` is given as a flat equation, each term v in $\vec{v}$ must be a constant or a variable. Now, if v is a constant then $v = v\gamma$ is irreducible, as just concluded. And if v is a variable then $v\gamma$ is irreducible, too, because γ is given as irreducible for `equal(`s`, `t`)`.

Recall we are considering the case that $s\gamma = f(\vec{v})\gamma$ is reducible. Because $v\gamma$ is irreducible for each v in $\vec{v}$, $f(\vec{v})\gamma$ must be reducible at the top position. That is, R must contain a rule of the form $s\gamma \to u$, for some term u.

From $s\gamma \to u \in R$ it follows that $s\gamma \to u$ has been added during the construction of R and hence R could not contain a rule that rewrites u before $s\gamma \to u$ was added. Further, the ordering $\succ$ on equations is defined in such a way that any rule that could rewrite u must precede the rule $s\gamma \to u$. Thus, a rule that rewrites u was not in R before $s\gamma \to u$ was added and it will not be added thereafter. Hence u is irreducible. In other words, deriving the normal form of $s\gamma$ takes exactly one step. Notice that this fact is independent from whether `equal(`$s\gamma$`, `$t\gamma$`)`$\in M$ or not. It holds for any flat term s and irreducible substitution γ. This result will be used also in the proof of the only-if direction below.

Next consider $t\gamma$. If $t\gamma$ is reducible then by the same arguments as for $s\gamma$ it must be of the form $g(\vec{w})\gamma$ where g is some function symbol and $\vec{w}$ is a list of constants or variables. Further, there is a rule $t\gamma \to u' \in R$ for some irreducible term u'.

Recall that any program obtained from the equality transformation contains the axioms of reflexivity, symmetry and transitivity. Further recall that M is a model of some such program.

Because of $R \subseteq M$, from $s\gamma \to u \in R$ and $t\gamma \to u' \in R$ it follows `equal(`$s\gamma$`, `u`)` $\in M$ and `equal(`$t\gamma$`, `u'`)` $\in M$. The symmetric versions are also contained in M by the symmetry axioms.

Because `equal(`$s\gamma$`, `$t\gamma$`)` $\in M$, `equal(`$s\gamma$`, `u`)` $\in M$ and `equal(`$t\gamma$`, `u'`)` $\in M$ and the fact that M must be a model in particular for the symmetry and transitivity axioms it follows `equal(`u`, `u'`)` $\in M$.

Next we will show that $u = u'$. Suppose, to the contrary that u and u' are different terms. But then, either $u \succ u'$ or $u' \succ u$ holds. Without loss of generality suppose $u \succ u'$. Recall that both u and u' are irreducible. But then $u \to u'$ was added during the construction of R and hence $u \to u' \in R$, contradicting irreducibility of u. Hence it follows $u = u'$. Consequently we have

$t\gamma \to u \in R$. Together with $s\gamma \to u \in R$ it follows trivially $s\gamma \to_R u$ and $t\gamma \to_R u$. Because R is convergent it follows $R^\star \models \texttt{equal}(s\gamma,t\gamma)$ as desired.

The last open case, that $t\gamma$ is irreducible, is treated similarly: From $\texttt{equal}(s\gamma,t\gamma) \in M$ and $\texttt{equal}(s\gamma,u) \in M$ it follows by the symmetry and transitivity axioms that $\texttt{equal}(t\gamma,u) \in M$. By the same arguments as above it must hold $t\gamma = u$, because both terms are irreducible, and if they were different, a rule $t\gamma \to u$ or $u \to t\gamma$ would have been added to R, contradicting irreducibility of $t\gamma$ and of u. Thus, with $s\gamma \to u \in R$ and $t\gamma = u$ it follows $R^\star \models \texttt{equal}(s,t)$.

This completes the proof for the if-direction.

For the only-if direction suppose $R^\star \models \texttt{equal}(s\gamma,t\gamma)$. Because R is a convergent rewrite system there is a term w such that $s\gamma \to_R^\star w$ and $t\gamma \to_R^\star w$.

If both $s\gamma$ and $t\gamma$ are irreducible then $s\gamma = t\gamma$. Hence $\texttt{equal}(s\gamma,t\gamma)$ is an instance of the reflexivity axiom, and so $\texttt{equal}(s\gamma,t\gamma) \in M$ follows.

Hence suppose that $s\gamma$ or $t\gamma$ is reducible. Without loss of generality suppose $s\gamma$ is reducible. By exactly the same arguments as made in the proof of the if-direction, $s\gamma$ can only be rewritable at the top position. Thus, there is a rule of the form $s\gamma \to u \in R$. In the if-part of the proof we concluded that deriving the normal form of $s\gamma$ takes exactly one step. This implies $u = w$.

If $t\gamma$ is reducible, by the same arguments as for $s\gamma$, there is a rule of the form $s\gamma \to u' \in R$ with $u' = w$. Because $R \subseteq M$ we get $\texttt{equal}(s\gamma,t\gamma) \in M$ and $\texttt{equal}(t\gamma,w) \in M$. By the symmetry and transitivity axioms, M must also satisfy $\texttt{equal}(s\gamma,t\gamma)$ and $\texttt{equal}(t\gamma,s\gamma)$. This means that $\texttt{equal}(s\gamma,t\gamma) \in M$ and $\texttt{equal}(t\gamma,s\gamma) \in M$.

If $t\gamma$ is irreducible, we have $t\gamma = w$. From $s\gamma \to w \in R$, $R \subseteq M$ and $t\gamma = w$ it follows (with the symmetry axiom) $\texttt{equal}(s\gamma,t\gamma) \in M$ and $\texttt{equal}(t\gamma,s\gamma) \in M$. $\square$

4 Translation of the Input Sentences to DLPs

This chapter shows how OntoNat translates the input sentences from natural language to DLPs. For this purpose, the first section introduces some linguistic notions. The second section gives an overview of the part of OntoNat that was crafted by the SALSA-group. This part is responsible for the linguistic pre-processing of the input sentences. The third section shows how OntoNat extracts DLPs from the results of the pre-processing.

4.1 Linguistic Background

Language is one of the most complex achievements of human evolution. It is estimated that there exist more than 6000 languages in the world. Although, in principle, the OntoNat approach could be applied to other languages as well, we focus on English.

It is well known that it is hard to describe a natural language by formal means, because natural languages are subject to a process of continuous development and change. Furthermore, natural languages tend to be ambiguous, i.e. a sentence can often be interpreted in different ways, expressing different ideas. Last, the meaning conveyed by a sentence is often not a pure combination of the senses of its words: It depends on the situation where the sentence is uttered, on the context, on the tone of voice, on the stressing of words and on the personal association with the words by the listener. This section will show how a reasonable description of the central aspects of a sentence can often be achieved in spite of that.

4.1.1 Formal Grammars

Alphabets. An *alphabet* is a finite set of symbols. A *string* on an alphabet is a sequence of elements of the alphabet. The *Kleene-set* of an alphabet Σ is the set of all strings on the alphabet, denoted $\Sigma^\star = \bigcup_{i>0} \Sigma^i$. A *production system* on an alphabet Σ is a binary relation on $\Sigma^\star$. Each element of a production system is called a *production rule*. It is common to write a production rule of the form $((x), (y_1, \ldots, y_n))$ as $x \rightarrow y_1 \ldots\ y_n$. The *rewrite relation* $\rightarrow_P$ of a production system $P \subseteq \Sigma^\star \times \Sigma^\star$ is the following transitive relation on $\Sigma^\star$:

$$
\begin{array}{ll}
x \rightarrow_P y & \text{if } (x, y) \in P \\
axb \rightarrow_P ayb & \text{if } x \rightarrow_P y \\
 & \text{where } a \text{ and } b \text{ are strings on } \Sigma
\end{array}
$$

$x \rightarrow_P z$ if for some y, $x \rightarrow_P y$ and $y \rightarrow_P z$

Grammars. A *grammar* is a tuple of

- an alphabet N of *non-terminal symbols*
- an alphabet T of *terminal symbols*
- a production system P on $T \cup N$, where each production rule must contain at least one non-terminal symbol in its left-hand side
- a *start-symbol* $s_0 \in N$

The *generated language* of a grammar (N, T, P, s_0) is the set of all *generated strings*, i.e. all strings s on T with $s_0 \rightarrow_P s$.

4.1.2 Natural Language Grammars

Correct Natural Language Grammars. To describe natural languages, we restrict ourselves to the written equivalents of utterances. We use the notions *word* and *sentence* in their common meaning[14]. If one takes the set of words of a natural language as terminal symbols of a grammar, one can use the grammar to generate sentences of the natural language. We say that such a grammar is *correct*, if each string generated by the grammar constitutes a syntactically acceptable sentence of the natural language.

Sample Grammar. Here is an example of a correct grammar for a very restricted subset of the English language:

N = {<Sentence>,<NounPhrase>,<VerbPhrase>,<Noun>,<Verb>,
<ProperName>,<Determiner>,<Preposition>}
T = {Elvis, comes, to, Saarbruecken}
P = {<Sentence> → <NounPhrase> <VerbPhrase>,
<NounPhrase> → <ProperName>,
<ProperName> → Elvis,
<ProperName> → Saarbruecken,
<VerbPhrase> → <Verb>,
<Verb> → comes,
<VerbPhrase> → <Verb> <Preposition> <NounPhrase>}

[14] There are no definitions that all researchers agree on.

This grammar can generate for example

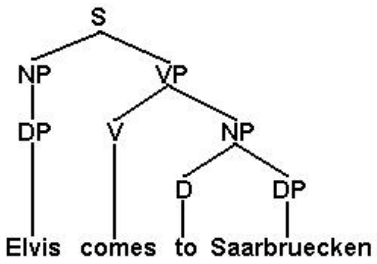
```
Elvis comes to Saarbruecken
Elvis comes
```

Note that the grammar is only correct concerning the syntax of the sentences, not concerning the semantics. The following sentences can be generated as well:

```
Saarbruecken comes to Elvis
Elvis comes to Elvis
```

Today, there exist correct grammars that cover a large part of the English language. Consequently, we may presuppose such a correct grammar (N, T, P, s_0) for English. A non-terminal x is a *syntactic category* of a word w, if $(x \rightarrow w) \in P$. We say that w *is a* x, e.g. `Elvis` is a proper name.

Syntax Trees. A *syntax tree* of a generated string is a tree, in which the inner nodes are non-terminal symbols. The leaf nodes are the terminal symbols of the string in the order given by the string. A parent node p may have the child nodes $c_1, \ldots, c_n$ in that order, iff there exists a production rule $(p \rightarrow c_1 \ldots\ c_n) \in P$. If one abbreviates the non-terminal symbols in the common way, the syntax tree of "Elvis comes to Saarbruecken" is for instance



Given a natural language sentence and assuming that the grammar can generate it, one can try to find a syntax tree of the natural language sentence. This process is called *parsing*.

4.1.3 Lexical Functional Grammars

The Theory of LFG. The *Lexical Functional Grammar (LFG)* is a theory of natural language, which was developed in 1982 by Bresnan [Bre82]. It bases on the following notions: An *attribute* is a symbol that is used to describe a

node in a syntax tree. Examples include syntactic properties such as `Gender`, `Number`, `Object` or `Subject`.

A *feature structure* is a partial function that maps attributes to *values*, where a value may be either a symbol or another feature-structure. Hence, one can apply a feature structure f to an attribute a_1, and if the result $f(a_1)$ is also a feature structure, one can apply $f(a_1)$ again to an attribute a_2 etc. Instead of writing $f(a_1)(a_2)\ldots(a_n)$, one writes a so-called *path* $(f\ a_1\ a_2\ \ldots\ a_n)$. Later, a path will always be associated to a node in a syntax tree. A node will always be associated to a feature structure. Then, a path may use $\downarrow$ to refer to the feature structure of its node and $\uparrow$ to refer to the feature structure of the parent node. A *path equation* is an expression of the form $p = c$ or $p_1 = p_2$, where c is a symbol and p, p_1, p_2 are paths.

Lexicons and C-Structures. A *lexicon* is a set of words, each of which is associated to its syntactic category and to a set of path equations. Here is a sample entry:

```
Elvis <ProperName>
      {(↓ Number) = sg}
```

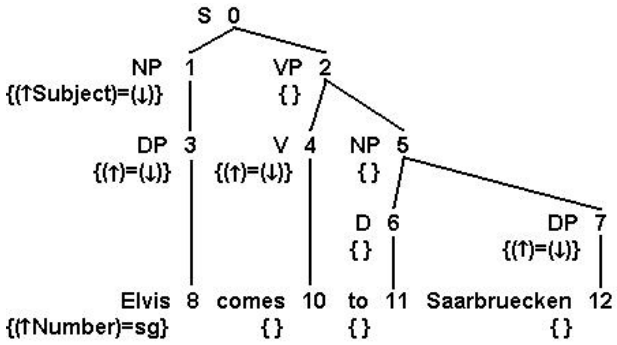
This example states that `Elvis` is a proper name. The path equation says that when `Elvis` becomes a leaf node in a syntax tree, then the feature structure of this leaf node shall map the attribute `Number` to the value `sg`.

A *c-structure* is a grammar, in which each non-terminal symbol in each production rule is associated to a set of path equations. Example:

```
<Sentence> → <NounPhrase>          <VerbPhrase>
             {(↑ Subject)=(↓)}     {}
```

The path equation says that if this rule is used to construct a syntax tree, then the feature structure of the parent node of `NounPhrase`, i.e. `Sentence`, shall map the attribute `Subject` to the feature structure of `NounPhrase`. The combination of a lexicon and a c-structure is called an *LFG-grammar*.

F-Structures. Given a natural language sentence, one may parse the sentence by the c-structure and generate a syntax tree. Each node is associated to a set of path equations: The words at the leaf nodes carry the path equations from the lexicon and the inner nodes carry the equations given by the c-structure. Here is a simple example, in which the nodes have been numbered:



Now assume that each node of the syntax tree is assigned to a feature structure. Then one can replace each $\downarrow$ in a path equation by the feature structure of the respective node and each $\uparrow$ by the feature structure of the parent node. The goal is to find an assignment of nodes to feature structures such that all equations hold. Such an assignment is called an *f-structure*.

Technically, an f-structure is a function f from nodes to feature structures. For the above example, f is as follows:

$f(8)$:
    $f(8)(\texttt{Number}) = \texttt{sg}$
$f(3) = f(8)$:
    $f(3)(\texttt{Number}) = \texttt{sg}$
$f(1) = f(3)$:
    $f(1)(\texttt{Number}) = \texttt{sg}$
$f(0)$:
    $f(0)(\texttt{Subject}) = f(1)$
All other feature structures are empty

F-structures may be constrained further by additional path equations and constraints. Depending on the powerfulness of the c-structure and the lexicon, the f-structure may provide a detailed analysis of the natural language sentence.

4.1.4 Word Senses

Sense is the relation between words and the entities the words refer to. For example, the sense of a proper name is the entity it denotes. The sense of a noun is the concept it refers to. Note that sense is not a function: A word may have multiple senses. In this case, we say that the word is *ambiguous*.

The process of finding out the intended sense of a word in a sentence is called *disambiguation*. Some words do not have a sense at all with this definition, such as for instance prepositions or conjunctions. The words that do have a sense are called *content words*.

4.2 Linguistic Pre-Processing

To translate the natural language input sentences to DLPs, the sentences need to be parsed. Furthermore, their words have to be disambiguated. This is the part of OntoNat that was designed by the SALSA-group.

4.2.1 Parsing

The Xerox Parser. To parse the input sentences, the SALSA-group used the LFG-grammar developed by the Parallel Grammar Project at the Palo Alto Research Center/California[BDK+02] and the accompanying LFG-parser of the Xerox Linguistics Environment. Given a natural language sentence, the parser constructs a syntax tree and calculates the f-structure. Then, the f-structure is normalized and transferred into a PROLOG-like format using the transfer system by Crouch[Cro05]. The resulting file lists only the feature structures of the leaf nodes. The feature structures are given by expressions of the form `a(f(`x`),`v`)`, meaning that the feature structure of node x maps the attribute a to the value v.

Sample Output. For the sentence "Elvis comes to Saarbruecken" with the leaf node numbering "Elvis" (8), "comes" (9), "to" (10), "Saarbruecken" (11), the parser yields for example (simplified):

```
pred(f(8),'Elvis').
num(f(8),sg).
proper(f(8),name).
pred(f(9),come).
tense(f(9),pres).
dsubj(f(9),f(8)).
obl(f(9),f(10)).
pred(f(10),to).
obj(f(10),f(11)).
pred(f(11),'Saarbruecken').
proper(f(11),location).
```

The `pred` attribute always identifies the corresponding word of a leaf node. Thus, we see for instance that the number of `Elvis` is singular. Furthermore, we learn that `Elvis` is a proper name, that the tense of "come" is present tense, that the the subject of "come" is Elvis (`dsubj`) and that "come" is connected to "Saarbruecken" by the proposition "to" (via `obl` and `obj`). We call the expressions of the form `f(`x`)` *f-identifiers*.

4.2.2 Disambiguation

WordNet Concepts. The LFG parser already disambiguates words to some extent, but OntoNat needs a disambiguation to ontological concepts. This kind of disambiguation requires a large database containing words and data on their senses. The SALSA-group used WordNet[Fel98]. WordNet is being developed at the Cognitive Science Laboratory of Princeton University/New Jersey under the direction of psychology professor George A. Miller.

WordNet knows about 140'000 words. It structures them in an ontology, where each word is associated to the concept given by the sense of the word. For instance, the words "answer", "reply" and "respond" are all associated to the concept of "making a statement in return to a question". Such a set of words that share one sense is called a *synset*. Words with multiple senses belong to multiple synsets. Thus, the task of disambiguation means essentially finding the appropriate synset for each word in the sentence.

Disambiguation to WordNet Concepts. The SALSA-group used a system developed by Branerjee and Pedersen[BP03]. This system first computes a vicinity measure on WordNet synsets. For example, two synsets are close, if the corresponding concepts are linked in the ontology or if the description of one synset uses words from the other synset.

For disambiguation, the system assumes that the words in a sentence belong to synsets that are close in WordNet. For instance, in a sentence like "The bank robbers got away with 1000 Dollars", it is highly probable that "bank" means "financial institution" instead of "sand bank". This can be estimated by finding out that in WordNet, "bank" in the sense of "financial institution" is much closer to "Dollar" than "bank" in the sense of "sand bank".

Disambiguation to SUMO Concepts. Although the WordNet-based disambiguation yields a concept for each word, these concepts cannot be used

directly, because they are WordNet concepts and OntoNat bases on SUMO concepts. Fortunately, there exists a mapping from WordNet concepts to SUMO concepts (see [NP03]). By looking up the corresponding SUMO concepts in this mapping, the SALSA-group could annotate each content word with a SUMO class. In the above example, the word "come" is annotated with the SUMO class `BodyMotion`. This completes the linguistic pre-processing of the SALSA-group.

4.3 Translation to DLPs

To translate the f-structures to DLPs, OntoNat proceeds in three steps: First, the f-structures are adjusted syntactically. Then, role relations are extracted from the grammatical relations given by the f-structure. The last step, negation, is only applied to the hypothesis.

4.3.1 Pre-Processing of the F-Structure

Syntactic Adjustments. First, OntoNat normalizes the f-structure: Deep pointer structures resulting from the class annotation are flattened. Identifiers are converted to lowercase, special characters are deleted and numbers are transformed to proper DLP identifiers.

Special predicates in the f-structure that the system makes no use of are filtered out. WordNet-references are converted to a KRHyper-compatible format. If one word is annotated with multiple SUMO-classes from different SUMO-subontologies, the class from the most specific ontology is kept and the others are omitted. Some known bugs of the SUMO mapping or the disambiguation are corrected.

F-Identifier Replacement. Since some words may be missing in WordNet or in the mapping to SUMO classes, there may be content words that do not have a SUMO class. Whenever the system finds such a word, it replaces the corresponding f-identifier by the word throughout the whole file. This facilitates a match between a hypothesis node and a premise node, even though the mediating SUMO concept is missing[15].

The same replacement is done for proper names. The underlying motivation is that an f-identifier stands for an arbitrary entity, whereas a proper name

[15]This may cause problems if a non-annotated word occurs twice in the sentence, because the occurrences cannot be distinguished any more.

always stands for the entity it denotes. In the example, the pre-processing yields:

```
pred(elvis,elvis).
num(elvis,sg).
sumo(elvis,human).
proper(elvis,name).
pred(f(9),come).
sumo(f(9),bodymotion).
tense(f(9),pres).
dsubj(f(9),elvis).
obl(f(9),f(10)).
pred(f(10),to).
obj(f(10),saarbruecken).
pred(saarbruecken,saarbruecken).
proper(saarbruecken,location).
```

4.3.2 Role Assignment

Role Instances. A *role instance* is an atom of the form $p(t_1, t_2)$, where p is a role relation symbol, t_1 identifies a process and t_2 identifies an entity that takes part in the process. For example, `agent(f(9),elvis)` is a role instance for the above sample sentence, because `Elvis` is the agent of the `BodyMotion`-process given by `f(9)`. The process of finding role instances and other semantically motivated atoms is called *role assignment*.

The SALSA-group is working on the automation of role assignment, but since these systems are under development, OntoNat uses heuristic rules as a first, tentative approach. The rules are written in form of a DLP. If this DLP is combined with the pre-processed f-structure file, KRHyper produces a model that contains the role instances.

Instances. We refer to f-identifiers and proper names collectively as *objects*. The heuristic rules make every object an `instance` of the corresponding SUMO class. This rule is not applied to SUMO-classes that identify classes of classes. For example, the class `OccupationalRole` has professions as its instances. Each profession is again a class, the instances of which are the persons practicing the profession. If a job title occurs in the text, like "journalist", it is disambiguated to the SUMO class `OccupationalRole`, because "journalist" is a profession.

But since the word "journalist" mostly refers to a person practicing this profession instead of the profession itself, we do not want the journalist to be an instance of `OccupationalRole`. This is why the rules create the ad-hoc class `class(journalist)`. They make the journalist an instance of this ad-hoc class and the ad-hoc-class a subclass of `Human`.

Contexts. Like all SUMO relations, roles may appear inside propositional predicates. Hence, role instances have to be equipped with a third argument that identifies their context. For this purpose, the role assignment associates each object to a context. The f-identifier of the main verb of the sentence, which is specially marked in the f-structure, gets the context `mainlevel`. If a role instance is found, it receives the context of whichever argument already has a context. The other argument is given this context as well. This way, the context is propagated from the main verb to all related objects and role instances.

If the rules encounter a SUMO class that denotes a propositional attitude (like `Stating`), they create a new context. This context is given instead of `mainlevel` to the propositional argument. Thus, the system can distinguish between "Elvis is alive" and "Gundolf thinks that Elvis is alive". If an object does not get a context by one of these mechanisms, it is given the `mainlevel` context as a default.

Ordinary Roles. Most heuristic rules react on special patterns in the f-structure. For example, there is a rule that detects expressions of the form `dsubj(f(`x`),`y`)`, which say that y identifies the grammatical subject of the verb identified by `f(`x`)`. The heuristics assumes that the grammatical subject of a verb constitutes the `agent` of the corresponding process. Consequently, the rule creates the role instance `agent(f(`x`),`y`)`.

Another rule detects attributive adjectives and makes them SUMO `attributes`. Again another rule reacts on complex adjuncts of the form "[something happens] thanks to [a cause]". It extracts a SUMO `causes` relationship from this adjunct. Of course, these rules cannot provide anything more than an interim solution, since they are purely heuristic in design. However, they produce reasonable results for simple input sentences.

Special Cases. If a negation is applied to an object, a special rule derives the atom `negated(`C`)`, where C is the context of the object. The heuristics also

handles verb tense specifications: An expression of the form `tense(`$f(x)$`,`t`)` dispatches a rule that derives `equal(whenfn(`$f(x)$`)),`t`)`, which means that the time of the process $f(x)$ is t. The time nomenclature of SUMO is simplified to the three times `now, future` and `past` and the tenses are mapped accordingly.

Another special case are phrases of the form "x is y", like "Elvis is the greatest artist of all times". The Unique Name Assumption prohibits an identification of "Elvis" with "artist". Instead, special substitution rules transfer all properties of "artist" to "Elvis". In the example, "Elvis" would get the attribute "great" and he would become an instance of the `artist` class. Since we are dealing with a limited number of properties, there are only 5 of these substitution rules.

Result of the Role Assignment. When KRHyper is run on the combination of the pre-processed f-structure and the heuristic rules, the resulting model contains the role instances. Furthermore, the model contains f-structure atoms and helper atoms from the role assignment. The predicates from the f-structure, the identifiers of SUMO and the helper predicates from the role assignment are all distinct, so that no conflict can occur. The essential new atoms for "Elvis comes to Saarbruecken" are[16]:

```
instance(f(9), bodymotion).
agent(f(9), elvis, mainlevel).
destination(f(9), saarbruecken, mainlevel).
instance(elvis, human).
instance(saarbruecken, geographicregion).
```

Note that, since each atom in this model can be regarded as a normal rule with no body atoms, the model file is already a DLP. Thus, we have found a translation from natural language sentences to DLPs.

4.3.3 Negation of the Hypothesis

Filtering. Role assignment is applied to both the premise and the hypothesis, which yields two DLPs. The next step is to negate the DLP of the hypothesis. For this purpose, OntoNat first scans the DLP and deletes atoms that are not essential to the hypothesis. This includes the f-structure, which is not needed any more, and the helper atoms added by the role assignment process. The

[16] Remember that, as discussed in 3.3.3, the relation `instance` does not get a context.

system also deletes some atoms that are relevant to the hypothesis, but do not contribute a major information.

This filtering is a trade-off between making the hypothesis too exact (and hence preventing a "Yes" answer) and making it too broad (thus enforcing a "Yes" answer). Best results can be achieved by keeping only `instance`, `attribute`, the major SUMO roles and the predicate `negated`. If the hypothesis does not contain a negation, OntoNat enforces the positive polarity of the `mainlevel` context by adding `not(negated(mainlevel))`. For the hypothesis "Elvis is alive", the filtering yields:

```
instance(elvis,human).
attribute(elvis,living,mainlevel).
not(negated(mainlevel)).
```

Negation. Now, the filtered DLP has to be negated. Since it consists of a sequence of atoms, its negation is a single rule with the atoms of the DLP as body atoms and the atom `false` as the head atom. In lack of deeper knowledge about the intended reading of the hypothesis, OntoNat considers all f-identifiers as existentially quantified. Thus, "A man steps on the moon" as well as "The man steps on the moon" are read as "There exists an instance of `Man` that steps on the moon". Hence in the negation, an f-identifier `f(`x`)` has to be universally quantified. This is achieved by simply converting it to a variable of the form `F`x.

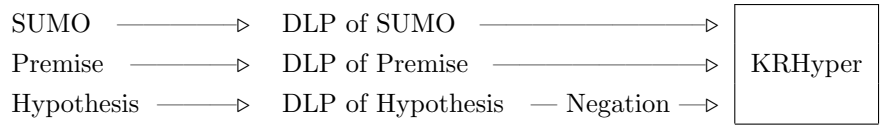
Last, the hypothesis may hold in any context, as long as this context is known to hold `now`. Hence the system makes `mainlevel` a universally quantified variable `Mainlevel` and constrains it by `holdsduring(now,Mainlevel,mainlevel)`. All in all, the negation of the above hypothesis yields:

```
false :- instance(elvis,human),
         attribute(elvis,living,Mainlevel),
         not(negated(Mainlevel)),
         holdsduring(now,Mainlevel,mainlevel).
```

This concludes the translation of the input sentences to DLPs.

5 Application of OntoNat and Conclusion

All components of the OntoNat system have now been explained:



This chapter concludes the thesis. The first section shows how OntoNat proves or disproves an entailment. The second section points out the difficulties that would lie ahead if the prototype was to be developed to a fully applicable system. The third section draws a conclusion and provides an outlook.

5.1 Functionality of OntoNat

Before the OntoNat system can be applied to input sentences, SUMO is compiled to a DLP, once and forever. When OntoNat is given a premise and a hypothesis, it translates these sentences to DLPs and negates the DLP of the hypothesis. Then KRHyper is run on the union of these three DLPs.

Proof Process with a True Hypothesis. During the proof process, KRHyper derives all knowledge it can extract from the premise by help of SUMO. Consider again the example of the premise "Elvis comes to Saarbruecken" and the hypothesis "Elvis is alive". From the premise, KRHyper deduces that Elvis is a human, that hence he is a mammal, an organism and an entity. It deduces that he has two lungs and a spine, that he must have been born, that he has two parents, that he will die, that he consists of cells, that each of these cells is an organism, that each of the cells is located in his body and so on.

Eventually, KRHyper will deduce all facts mentioned in the hypothesis. Then, the negated hypothesis rule dispatches and `false` is derived[17]. In the example, KRHyper knows that Elvis is the agent of a process (the `BodyMotion`). Since an axiom states that agents of processes are alive during the time of the process, KRHyper deduces that Elvis must be alive during the `BodyMotion`-process. The time of this process is `now`, so that Elvis must be alive `now`. This contradicts the negated hypothesis and `false` is produced. This means that the hypothesis is considered true.

[17] KRHyper will seek alternative models, but in practice, allowing or preventing this did not affect the performance.

Although SUMO contains thousands of rules and thousands of facts are derived, the proof process is usually fast. The Elvis example is proven in less than 2 seconds.

Proof Process with a False Hypothesis. If the hypothesis cannot be proven, KRHyper continues deriving facts. SUMO contains axioms that create entities to which the axioms may be applied again. For example, each organism has a parent, which is again an organism. Hence the parent also has a parent and so on. This entails that KRHyper does not terminate if it does not encounter a contradiction. Since in practice, most true hypotheses were derived within 5 seconds, it has proven reasonable to interrupt KRHyper after some time and to consider the hypothesis false in this case[18].

5.2 Discussion of Applicability

We were interested in the difficulties that our approach would face if the OntoNat prototype was to be elaborated to a fully applicable system. Hence we applied the system to a set of 420 premise/hypothesis pairs[19]. We identified three groups of hurdles that would have to be overcome: Grammar-related, SUMO-related and content-related ones.

Grammar-Related Hurdles. We found that the Xerox-parser occasionally produces corrupt f-structures. This shows that the problem of exhaustive and correct grammatical parsing is not yet entirely solved. The greatest challenge, though, would be the role assignment. The current, tentative role assignment is designed only for a fraction of the grammatical patterns that indicate role instances. A particularly tough task would be the detection of multi-word expressions or idiomatic phrases. Currently, OntoNat deduces for instance that the coal stocks receive an elevator, if the sentence is "Coal stocks got a lift".

SUMO-Related Hurdles. In some cases, the mappings from WordNet synsets to SUMO-classes require additional attention. For instance, the verb "to rescue" is mapped to the quite abstract `SubjectiveAssessmentAttribute` instead of some concrete action like `Removing`. For some other sentences, the ontology is

[18] Methods that stop the theorem prover in case the hypothesis is not provable are in preparation, see [BS05].

[19] We used the data from the Text Entailment Challenge of the PASCAL Network, see `http://www.pascal-network.org/Challenges/RTE/`.

too precise. For example, OntoNat said that the premise "The report showed the consequences" was inconsistent, because a report is a piece of paper and "showing" is an intentional process, which is done by humans. These cases are easy to correct. The cases that would consume much more work are those that require a more precise ontology. For example, the noun "assassin" is currently mapped to `Human`, which loses the information that is crucial to the word[20]. Overall, the more the ontology is accurate, the better the system will perform.

Content-Related Hurdles. It is clear that the target of a logic-driven system cannot be the poetic dimension of language. Metaphors, however, make up an important part of discourse and they currently pose an obstacle to OntoNat. Furthermore, the system faces all the pitfalls of language in the domain of semantics. These include for instance non-subsective adjectives. Currently, OntoNat assumes that a "former doctor" is a doctor with the property of being "former" – which is false.

Moreover, a logic-driven system has to cope with all the imprecisions of language in the domain of pragmatics. Currently, for example, OntoNat will not understand that in "Reagan recovered quickly after the assassination attempt", Reagan was probably not a by-stander to the assassination attempt, but the victim. Last, the OntoNat approach tends to be a bit hairsplitting on the understanding of the sentences. Consider for example the following pair:

Premise: "A woman hit her partner with a frying pan"

Hypothesis: "A man was hit with a frying pan".

This hypothesis seems true. However, the system said it is false – after all, the partner of the woman does not need to be a man.

5.3 Conclusion and Outlook

Conclusion. The OntoNat system could find out that the premise "Elvis comes to Saarbruecken" entails the hypothesis "Elvis is alive". This is a non-trivial entailment, which exhibits some kind of "understanding" of the sentences. Thereby, OntoNat demonstrated how a logic-based approach could contribute to question answering on natural language sentences with common sense reasoning.

The discussion of applicability, though, made clear that an enormous amount of work would be required to bring the OntoNat prototype anywhere near an

[20] It may be that "assassin" is defined in some other subontology of SUMO. However, the problem of imprecision will probably remain.

acceptable performance for a real application. In particular, the ontology and the role assignment would need to be elaborated. Furthermore, OntoNat would have to overcome numerous pitfalls in the domain of semantics and pragmatics. Nevertheless, much of the above discussion of applicability was already concerned with the content and the readings of the input sentences. This may indicate that OntoNat has reached a level worth of deliberation.

Outlook. The discussion of applicability pointed out that the components that would improve the performance most are the ontology and the role assignment. Fortunately, these components are steadily being elaborated: First, SUMO is continuously being extended and made more consistent. The transformation to a DLP made its contribution by discovering syntactic errors and ontological inconsistencies, which were reported to and resolved by the creators of SUMO.

Second, the SALSA-group is working on the assignment of roles by statistical means, machine learning techniques and rule-based methods. They assign the roles of FrameNet[BFL98]. We designed an algorithm that maps FrameNet roles to SUMO roles, so that the current heuristic rules can be replaced by a linguistically profound mechanism.

The pitfalls of language, though, will remain. But since they are the subject of intense research in computational linguistics, it is not hopeless to expect a better handling of these pitfalls in the future.

In summary, this thesis has shown how a logic-based approach could contribute to question answering with common sense reasoning. It is evident that years of further work are required to answer questions like a human. OntoNat could be a start.

A OntoNat Program Code

A.1 Translation of SUMO to a DLP

This PROLOG-program, `kif2tme.pl`, translates SUMO (without the abstract sections) and the mid-level-extension of SUMO (in its manually debugged version) to a DLP.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                                 %
%                     kif2tme.pl                                                  %
%                                                                                 %
%                                                                                 %
% Translates SUMO to a DLP                                                        %
%                                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Main Call                                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Switches:
suspicious(off). % Filters out suspicious (non-constrained) variables
nonground(on).   % Filters out non-ground disjuncts
skolemterms(on). % Uses Skolem terms (instead of Skolem constants)
axioms(on).      % Compiles the axioms of SUMO (or just the classes)

% Open, read and translate SUMO-file
main :-
    telling(OldTell),
    tell('sumo.tme'),
    writeHeader,!,
    ((member(F,['sumo.kif','Mid-level-ontology-corr.kif']),
      translate(F),
      fail) ; !),
    told,
    tell(OldTell).
```

```
% Translates a file
translate(File) :-
    seeing(OldSee),
    see(File),
    repeat,
      peek_char(C),
      (
        C = end_of_file -> ! ;
        C = ';' -> (skipLine,fail) ;
        isBlank(C) -> (get_char(_), fail) ;
        C = '(' -> (readFormula(F), processFormula(F),fail) ;
        write('Error: Illegal char '), char_code(C,CC),write(CC),!
      ),
    seen,
    see(OldSee).

% Writes static KRHyper-code
writeHeader :-
    writeln('equal(X,X).'),
    writeln('equal(X,Y) :- equal(Y,X).'),
    writeln('equal(X,Z) :- equal(X,Y), equal(Y,Z).'),
    writeln('identical(X,X).'),
    writeln('isFunction(fun(_)).'),
    writeln('isFunction(fun(_,_)).'),
    writeln('isFunction(fun(_,_,_)).'),
    writeln('isFunction(fun(_,_,_,_)).'),
    writeln('isFunction(fun(_,_,_,_,_)).'),
    writeln('isFunction(fun(_,_,_,_,_,_,_)).'),
    writeln('isFunction(fun(_,_,_,_,_,_,_)).'),
    writeln('false :- equal(X,Y), not(isFunction(X)),'),
    writeln('    not(isFunction(Y)), not(identical(X,Y)).'),
    writeln('instance(X,Super) :- '),
    writeln('    instance(X,Sub), subclass(Sub, Super).'),
    writeln('subclass(X,Z) :- '),
    writeln('    subclass(X,Y), subclass(Y,Z).'),
    writeln('equal(endfn(now),future).'),
    writeln('equal(endfn(future),future).'),
```

```
    writeln('equal(endfn(past),past).'),
    writeln('equal(beginfn(now),past).'),
    writeln('equal(beginfn(past),past).'),
    writeln('equal(beginfn(future),future).'),
    writeln('equal(futurefn(now),future).'),
    writeln('equal(futurefn(future),future).'),
    writeln('equal(futurefn(past),future).'),
    writeln('equal(futurefn(now),future).'),
    writeln('equal(pastfn(past),past).'),
    writeln('equal(pastfn(future),past).'),
    writeln('equal(pastfn(now),past).').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Reading                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Reads a formula (add list of quantified and global variables)
readFormula(F) :- readFormula(F2,[],GlobalVars),
    (GlobalVars = [] ->
         F=F2 ;
         F=[forall,GlobalVars,F2]).

% Read '(', arguments, ')'
% readFormula(-Formula,+Variables,-GlobalVariables)
readFormula(F,V,GV) :-
    peek_char('('), !,
    get_char(_),
    skipBlanks, argList(F,V,GV),
    get_char(')').

% Read list of arguments
% argList(-ArgumentList,+Variables,-GlobalVariables)
argList([],_,[]) :- peek_char(')'), !.

argList([Arg|R],V,GV) :-
    readFormula(Arg,V,GV1), !,
    skipBlanks, argList(R,V,GV2),
```

```
    union(GV1,GV2,GV).

argList([string|R],V,GV) :-
    peek_char('"'), !,
    get_char(_), skip('"'),
    skipBlanks, argList(R,V,GV).

argList(Result,V,GV) :-
    getIdCharList(Id), !, (
      Id=['?'|_] ->
          handleVar(Id,V,GV,Result) ;
      (Id=[f,o,r,a,l,l];Id=[e,x,i,s,t,s]) ->
          handleQuant(Id,V,GV,Result) ;
      handleAtom(Id,V,GV,Result)
    ).

argList([],_,[]).

% If a free variable occurs, make it global
% handleVar(+IdentifierCharList,+Variables,-GlobalVars,-Formula)
handleVar(['?'|IdCharList],V,[Id|GV],[Id|Result]) :-
    concat_atom([x|IdCharList],Id),
    not(member(Id,V)), !,
    skipBlanks, argList(Result,[Id|V],GV).

handleVar(['?'|IdCharList],V,GV,[Id|Result]) :-
    concat_atom([x|IdCharList],Id),
    skipBlanks, argList(Result,[Id|V],GV).

% If a quantifier occurs, register variables
% handleQuant(+QuantifierCharList,+Variables,+GlobalVars,-Formula)
handleQuant(QCharList,V,GV,[Q,QVars,Result]) :-
    concat_atom(QCharList,Q),
    skipBlanks, readFormula(QVars,[],_),
    append(QVars,V,V2),
    skipBlanks,readFormula(Result,V2,GV).
```

```
% If a function occurs, make it is a function term
% handleAtom(+IdentifierCharList,+Variables,+GlobalVars,-Formula)
handleAtom(IdCharList,V,GV,[fun,Id|Result]) :-
    isFn(IdCharList), !,
    concat_atom(IdCharList,Id),
    skipBlanks, argList(Result,V,GV).

handleAtom(IdCharList,V,GV,[Id|Result]) :-
    concat_atom(IdCharList,Id),
    skipBlanks, argList(Result,V,GV).

% Collect the chars of an identifier
% getIdCharList(-IdentiferCharList)
getIdCharList([F|R]) :-
    peek_char(C), not(isBlank(C)),C\=')',!,
    get_char(G),lcaseString(G,F),getIdCharList(R).

getIdCharList([]).

% Skips a line
skipLine :- skip('\n').

% Skips blanks
skipBlanks :- peek_char(C), isBlank(C), !, get_char(_), skipBlanks.
skipBlanks.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Processing Formulae                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Predicate "sumo" registers properties of SUMO identifiers
:- dynamic(sumo/3).
sumo(a,b,c).

% Process an untreatable formula
processFormula(F) :-
  flatten(F,Flat),
```

```
  intersection(Flat,[listfn,lastfn,firstfn,listlengthfn,
                     listorderfn,inlist,modalattribute,
                     entails,holds,streetaddressfn,
                     successorattributeclosure,true,
                     singlefamilyresidence,
                     documentation],I),
  I\=[],!.

% Ignore things other than "subclass" if "axioms" is off
processFormula(F) :- axioms(off), not(F=[subclass,_,_]), !.

% Split main level "<=>"
processFormula(['<=>',A,B]) :- !,
  processFormula(['=>',A,B]),
  processFormula(['=>',B,A]).

% Split main level "<=>" with "forall"
processFormula([forall,V,['<=>',A,B]]) :- !,
  processFormula([forall,V,['=>',A,B]]),
  processFormula([forall,V,['=>',B,A]]).

% Ignore 'fun'
processFormula([P,fun,Function,Class]) :- !,
    processFormula([P,Function,Class]).
processFormula([P,fun,Function,A,Class]) :- !,
    processFormula([P,Function,A,Class]).

% Retract previous data
processFormula([instance,Function,_]) :-
    sumo(_,F,_), F\=Function,
    retractall(sumo(_,F,_)),
    fail.

% Process function and relation declarations
processFormula([instance,_Function,unaryfunction]) :- !,
    funDeclaration(1).
processFormula([instance,_Function,binaryfunction]) :- !,
```

```
    funDeclaration(2).
processFormula([instance,Function,binarypredicate]) :- !,
    assert(sumo(arity,Function,3)).
processFormula([instance,Function,binaryrelation]) :- !,
    assert(sumo(arity,Function,3)).
processFormula([instance,_Function,ternaryfunction]) :- !,
    funDeclaration(3).
processFormula([instance,Function,ternarypredicate]) :- !,
    assert(sumo(arity,Function,4)).
processFormula([instance,Function,ternaryrelation]) :- !,
    assert(sumo(arity,Function,4)).
processFormula([instance,_Function,quaternaryfunction]) :- !,
    funDeclaration(4).
processFormula([instance,Function,quaternarypredicate]) :- !,
    assert(sumo(arity,Function,5)).
processFormula([instance,Function,quaternaryrelation]) :- !,
    assert(sumo(arity,Function,5)).
processFormula([instance,_Function,quintaryfunction]) :- !,
    funDeclaration(5).
processFormula([instance,Function,quintarypredicate]) :- !,
    assert(sumo(arity,Function,6)).
processFormula([instance,Function,quintaryrelation]) :- !,
    assert(sumo(arity,Function,6)).

% Process meta-classes
processFormula([instance,Function,asymmetricrelation]) :-
    skolemterms(on), !,
    writef("false :- %w(X,Y,C), %w(Y,X,C).\n",[Function,Function]).
processFormula([instance,Function,irreflexiverelation]) :-
    skolemterms(on), !,
    writef("false :- %w(X,X,C).\n",[Function]).
processFormula([instance,Function,symmetricrelation]) :- !,
    writef("%w(X,Y,C) :- %w(Y,X,C).\n",[Function,Function]).
processFormula([instance,Function,reflexiverelation]) :- !,
    assert(sumo(reflexive,Function,yes)).
processFormula([instance,Function,transitiverelation]) :- !,
    writef("%w(X,Z,C) :- %w(X,Y,C), %w(Y,Z,C).\n",
```

```
          [Function,Function,Function]).
processFormula([instance,Function,equivalencerelation]) :- !,
    processFormula([instance,Function,transitiverelation]),
    processFormula([instance,Function,reflexiverelation]),
    processFormula([instance,Function,symmetricrelation]).

% Process domain-defining meta-relations
processFormula([range,Function,Class]) :- !,
    (sumo(arity,Function,N) ->
        atom_number(A,N),
        processFormula([domain,Function,A,Class]) ;
        true).

processFormula([domain,Function,Class]) :- !,
    processFormula([domain,Function,'1',Class]).

processFormula([domain,Function,P,Class]) :-
    sumo(function,Function,yes), !,
    writef("instance(X%w, %w) :- equal(fun(%w,",[P,Class,Function]),
    (sumo(arity,Function,Arity) ->
         true ;
         Arity=2),
    writeXs(Arity),
    writef("),Y), not(identical(Y,fun(%w,",[Function]),
    writeXs(Arity),
    writeln('))).').

processFormula([domain,Function,P,Class]) :- !,
    writef("instance(X%w, %w) :- %w(",[P,Class,Function]),
    (sumo(arity,Function,Arity) ->
         true ;
         Arity=3),
    writeXs(Arity),
    writeln(').'),
    (sumo(reflexive,Function,yes) ->
         writef("%w(X,X,C) :- instance(X, %w).\n",
         [Function, Class]) ;
```

```
        true).

processFormula([rangesubclass,Function,Class]) :- !,
    (sumo(arity,Function,N) ->
        atom_number(A,N),
        processFormula([domainsubclass,Function,A,Class]) ;
        true).

processFormula([domainsubclass,Function,Class]) :- !,
    processFormula([domainsubclass,Function,'1',Class]).

processFormula([domainsubclass,Function,P,Class]) :-
    sumo(function,Function,yes), !,
    writef("subclass(X%w, %w) :- equal(fun(%w,",[P,Class,Function]),
    (sumo(arity,Function,Arity) -> true; Arity=2),
    writeXs(Arity),
    writef("),Y), not(identical(Y,fun(%w,",[Function]),
    writeXs(Arity),
    writeln('))).').

processFormula([domainsubclass,Function,P,Class]) :- !,
    writef("subclass(X%w, %w) :- %w(",[P,Class,Function]),
    (sumo(arity,Function,Arity) -> true; Arity=3),
    writeXs(Arity),
    writeln(').').

% Process other meta-relations
processFormula([disjoint,Class1,Class2]) :-
    %# Avoid problems with metaphors ("the text says that"):
    not(Class1=organicobject),!,
    writef("false :- instance(X, %w), instance(X, %w).\n",
           [Class1,Class2]).

processFormula([inverse,Relation1,Relation2]) :- !,
    writef("%w(X,Y,C) :- %w(Y,X,C).\n",[Relation1,Relation2]),
    writef("%w(X,Y,C) :- %w(Y,X,C).\n",[Relation2,Relation1]).
```

```
processFormula([disjointrelation|Relations]) :- !,
    (member(R1,Relations),
       member(R2, Relations),
         R1\=R2,
         writef("false :- %w(X,Y,C), %w(Y,X,C).\n",[R1,R2])
    ; true).

processFormula([contraryattribute|Attributes]) :-
    %# Avoid problem with "north-east London":
    not(Attributes=[north, south, east, west]),
    !,
    (member(R1,Attributes),
       member(R2, Attributes),
         R1\=R2,
         writef("false :- attribute(X,%w,C), attribute(X,%w,C).\n",
                [R1,R2])
     ; true).

processFormula([subattribute,Attribute1,Attribute2]) :- !,
    writef("attribute(X,%w,C) :- attribute(X,%w,C).\n",
           [Attribute2,Attribute1]).

processFormula([exhaustiveattribute, General|Attributes]) :- !,
    (member(A,Attributes),
         (Attributes=[A|_] ->
              writef("attribute(X,%w,C)",[A]) ;
              writef("; attribute(X,%w,C)",[A])),
      fail
    ; writef(" :- attribute(X,%w,C).\n",[General])).
    % Subattribute-relation is explicit in the sumo-data

processFormula([exhaustivedecomposition, General|Classes]) :- !,
    (member(A,Classes),
      (Classes=[A|_] ->
           writef("instance(X,%w)",[A]) ;
           writef("; instance(X,%w)",[A])),
      fail
```

```
    ; writef(" :- instance(X,%w).\n",[General])).
    % Subclass-relation is explicit in the sumo-data

processFormula([partition, General|Classes]) :-
    %# Bug in SUMO:
    not(General=organicobject),!,
    processFormula([disjointdecomposition, General|Classes]).

processFormula([disjointdecomposition, General|Classes]) :- !,
    processFormula([exhaustivedecomposition, General|Classes]),
    (member(R1,Classes),
       member(R2, Classes),
         R1\=R2,
         writef("false :- instance(X,%w), instance(X,%w).\n",
                [R1,R2])
    ; true).

processFormula([subrelation,R1,R2]) :- !,
    writef("%w(X,Y,C) :- %w(X,Y,C).\n",[R2,R1]).

% Process a normal formula
processFormula(F) :-
    nnf(F, R1),
    ccFunTrans(R1,R4),
    contextTrans(R4, R5),
    funTrans(R5, R6),
    skolemTrans(R6,R7),
    clauseTrans(R7,R8),
    (suspicious(on) -> suspiciousVariables(R8,V) ;
     nonground(on)  -> nongroundVariables(R8,V) ;
                       V=[]),
    ( V=[] -> rulepp(R8) ;
      (write('% kif2tme: Suspicious variables in formula: '),
       list_to_set(V,Vs),
       write(Vs),
       write('\n% '),
       pp(F),
```

```
        writeln(''),
        length(R8,R8L),
        (R8L < 100 -> write('% '),writeln(R8) ; !))), !.

% Gently fail on problematic formulae
processFormula(F) :-
    write('% kif2tme: Problem with formula\n% '),
    pp(F), writeln(' ').

% Get unconstrained universal variables in a clause
% suspiciousVariables(+Clause,-UnconstrainedVariables)
suspiciousVariables([D|R],L) :-
   checkDisj(D,P,N),
   findall(X,(member(X,P), not(member(X,N))),L1),
   suspiciousVariables(R,L2),
   append(L1,L2,L).

suspiciousVariables([],[]).

% Collect the variables in a disjunction
% checkDisj(+Disjunction,-PositiveVars,-NegativeVars)
checkDisj([[not,[_Pred|Args]]|R],P,N2) :- !,
    collectUV(Args,UVs),
    checkDisj(R,P,N),
    union(N,UVs,N2).

checkDisj([[_Pred|Args]|R],P2,N) :- !,
    collectUV(Args,UVs),
    checkDisj(R,P,N),
    union(P,UVs,P2).

checkDisj([],V,V).

% Collect nonground disjunct variables
% nongroundVariables(+Clause,-NongroundVars)
nongroundVariables([D|R],L) :-
    isHorn(D,0), !,
```

```
    nongroundVariables(R,L).

nongroundVariables([D|R],L) :-
    checkDisj(D,P,N),
    findall(X,(member(X,P), not(member(X,N))),L1),
    nongroundVariables(R,L2),
    append(L1,L2,L).

nongroundVariables([],[]).

% Tells whether a clause is a Horn-Clause
% isHorn(+Clause,0)
isHorn([[not|_]|R],N) :- !, isHorn(R,N).
isHorn([_|R],0) :- isHorn(R,1).
isHorn([],1).
isHorn([],0).

% Collect universal variables from an argument list
% collectUV(+ArgumentList,-Variables)
collectUV([V|R],[V|Vars]) :-
   atom(V),
   not(number(V)),
   ucaseVar(V,V), !, collectUV(R,Vars).

collectUV([[fun|FArgs]|R],Vars) :-
  collectUV(FArgs,FVars),
  collectUV(R,RVars),
  append(FVars,RVars,Vars).

collectUV([_|R],Vars) :- collectUV(R,Vars).

collectUV([],[]).

% Declares a function
% funDeclaration(+Arity)
funDeclaration(N) :-
    assert(sumo(arity,Function,N)),
```

```
    assert(sumo(function,Function,yes)).

% Writes a number of variables
% writeXs(+HowMany)
writeXs(1) :- !, write('X1').
writeXs(N) :- N2 is N - 1 , writeXs(N2),writef(",X%w",[N]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Helper routines                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Replaces the first matching element of a list
% replace(+List1,+Element1,+Element2,-Result)
replace([E1|L1], E1, E2, [E2|L1]) :- !.
replace([F1|L1], E1, E2, [F2|L1]) :- replace(F1, E1, E2, F2).
replace([F|L1], E1, E2, [F|L2]) :- replace(L1, E1, E2, L2).

% Tells whether a list of characters is a function identifier
% isFn(+IdentifierCharList)
isFn(['f','n']).
isFn([_|R]) :- isFn(R).

% Tells whether a character is blank
% isBlank(+Char)
isBlank(' ').
isBlank('\n').
isBlank('\t').
isBlank('\r').

% Upcase a char
% ucase(+Char,-UpcasedChar)
ucase(X,Y) :-
    char_code(X,N),
    N < 127, N > 96, !,
    N2 is N - 32, char_code(Y,N2).
ucase(X,X) .
```

```
% Lowcase a char
% lcase(+Char,-LowcasedChar)
lcase(X,Y) :-
    char_code(X,N),
    N =< 90, N > 64, !,
    N2 is N + 32,
    char_code(Y,N2).
lcase(X,X) .

% Lowcase a string of characters
% lcaseString(+String,-UpcasedString)
lcaseString(X,Y) :-
    char_code(X,N),
    N =< 90, N > 64, !,
    N2 is N + 32,
    char_code(Y,N2).
lcaseString(X,X) .

% Upcase a variable
% ucaseVar(+Variable,-UpcasedVariable)
ucaseVar(V,V2) :-
    concat_atom([F|R],V),F\='',!,
    ucase(F,F2),concat_atom([F2|R],V2).

% Upcase a list of variables
% ucaseVarList(+VarList,-UpcasedVariableList)
ucaseVarList([F|R],[F2|R2]) :-
    ucaseVar(F,F2),
    ucaseVarList(R,R2).

ucaseVarList([],[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Pretty Printer for formulae                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Prints a formula in human-readable format
% pp(+Formula)
pp([and,A,B|C]) :- !,
    pp(A), write(' and '), pp([and,B|C]).

pp([and,A]) :- !,
    pp(A).

pp([and]) :- !.

pp([not,A]) :- !,
    write('~'),pp(A).

pp([or,A,B|C]) :- !,
    write('('), pp(A), write(' or '), pp([or,B|C]), write(')').

pp([or,A]) :- !,
    pp(A).

pp([or]) :- !.

pp(['=>',A,B]) :- !,
    write('('), pp(A), write(' => '), pp(B), write(')').

pp(['<=>',A,B]) :- !,
    write('('), pp(A), write(' <=> '), pp(B), write(')').

pp([exists, Var, A]) :- !,
    write('Ex '), ppArgs(Var), write(': '), pp(A).

pp([forall, Var, A]) :- !,
    write('All '), ppArgs(Var), write(': '), pp(A).

pp([Name]) :- !,
    write(Name).

pp([Name|Args]) :- !,
```

```
    write(Name), write('('),ppArgs(Args), write(')').

pp(N) :- write(N).

% Pretty prints a list of arguments
ppArgs([F,S|R]) :- !,
    pp(F), write(','), ppArgs([S|R]).

ppArgs([F]) :- !,
    pp(F).

ppArgs([]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Pretty Printer for clausal forms                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Print disjunction by disjunction
% rulepp(+Clause)
rulepp([F|R]) :- disjpp(F), rulepp(R).
rulepp([]).

% Collect negative and positive literals, print separately
% disjpp(+Disjunction)
disjpp(D) :-
    getPosNeg(D,Pos,Neg),
    (Pos=[] ->
         write('false') ;
         writeList('',Pos,'; ')),
    (Neg=[] ->
         writeln('.') ;
         writeln(' :- '),
         writeList('    ',Neg,',\n'),
         writeln('.')).

% Collects positive and negative literals
% getPosNeg(+Disjunction,-Positive,-Negative)
```

```
getPosNeg([[not,P]|R],Pos,[P|Neg]) :- !, getPosNeg(R,Pos,Neg).

getPosNeg([P|R],[P|Pos],Neg) :- getPosNeg(R,Pos,Neg).

getPosNeg([],[],[]).

% Writes a list neatly
% writeList(+IdBefore,+List,+IdAfter)
writeList(Before,[F,S|R],After) :- !,
    write(Before),pp(F),write(After),
    writeList(Before,[S|R],After).

writeList(Before,[F],_After) :- !,
    write(Before),pp(F).

writeList(_,[],_).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Introducing new Variables                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-dynamic(newVarCounter/1).

% Holds the current variable number
newVarCounter(1).

% Returns a new variable
% newVar(-Variable)
newVar(X) :-
    newVarCounter(N),
    retract(newVarCounter(N)),
    N2 is N + 1,
    assert(newVarCounter(N2)),
    concat_atom([x,'_',N],X).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Creating the negation normal form                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% nnf(+Formula,-NegatioNormalForm)
nnf([and|As], [and|As2]) :- !,
    nnfList(As,As2).

nnf([or|As], [or|As2]) :- !,
    nnfList(As,As2).

nnf(['<=>',A,B],R) :- !,
    nnf([and,['=>',A,B],['=>',B,A]],R).

nnf(['=>', A,B], [or, NotA,B2]) :- !,
    nnf([not,A],NotA), nnf(B,B2).

nnf([exists, Var,A], [exists, Var,A2]) :- !,
    nnf(A,A2).

nnf([forall, Var,A], [forall, Var,A2]) :- !,
    nnf(A,A2).

nnf([not,[false]],[true]) :-!.

nnf([not,[true]],[false]) :-!.

nnf([not, [and|As]], [or|As2]) :- !,
    nnfListNegated(As,As2).

nnf([not, [or|As]], [and|As2]) :- !,
    nnfListNegated(As,As2).

nnf([not, ['=>', A,B]], [and, A2,B2]) :- !,
    nnf(A,A2), nnf([not, B],B2).

nnf([not, [exists, Var,A]], [forall, Var,A2]) :- !,
```

```
    nnf([not, A],A2).

nnf([not, [forall, Var,A]], [exists, Var,A2]) :- !,
    nnf([not, A],A2).

nnf([not, ['<=>',A,B]],R) :- !,
    nnf(['<=>',[not,A],B],R).

nnf([not, [not, A]], A2) :- !,
    nnf(A,A2).

nnf([not, [Name|Args]], [not, [Name|Args]]):- !.

nnf([Name|Args], [Name|Args]).

% Create the NNF of a list of formulae
% nnfList(+FormulaList,-NNFList)
nnfList([F|R],[F2|R2]) :-
    nnf(F,F2),
    nnfList(R,R2).

nnfList([],[]).

% Create the NNF of a list of formulae, negating each one
% nnfListNegated(+FormulaList,-NNFList)
nnfListNegated([F|R],[F2|R2]) :-
    nnf([not,F],F2),
    nnfListNegated(R,R2).

nnfListNegated([],[]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Translating functions                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tells whether a predicate-list is a function call
```

```
isFunction([fun|_]).

% Translate a list of predicates
% funTransList(+PredicateList,-ResultList)
funTransList([F|R],[F2|R2]) :-
   funTrans(F,F2),
   funTransList(R,R2).

funTransList([],[]).

% funTrans(+Formula,-ResultFormula)

% Operators: distribute
funTrans([and|As], [and|As2]) :- !,
    funTransList(As,As2).

funTrans([or|As], [or|As2]) :- !,
    funTransList(As,As2).

% Quantifiers: distribute
funTrans([forall, Var,A], [forall, Var,A2]) :- !,
    funTrans(A,A2).

funTrans([exists, Var,A], [exists, Var,A2]) :- !,
    funTrans(A,A2).

% Predicates: translate the arguments (negative polarity)
funTrans([not, [Name|Args]], Result):- !,
   funTransArgs(Args, NewArgs, [not,[Name|NewArgs]],Result).

% Predicates: translate the arguments (positive polarity)
funTrans([Name|Args], Result):-
   funTransArgs(Args, NewArgs, [Name|NewArgs],Result).

% funTransArgs(+ArgList,-ResultArgList,+FinalAtom,-Result)

% Functional argument: flatten
```

```
funTransArgs([[fun, FName|FArgs]|PArgs], [Y|NewPArgs],
             Head,Result) :- !,
    newVar(Y),
    funTransArgs(FArgs,NewFArgs,
                 [forall, [Y], [or,
                     [not,[equal,[fun,FName|NewFArgs],Y]],
                     Rest]],Result),
    funTransArgs(PArgs, NewPArgs, Head,Rest).

% Non-functional argument: copy
funTransArgs([PArg|PArgs], [PArg|NewPArgs], Head,Result) :-
    funTransArgs(PArgs,NewPArgs,Head,Result).

% If no more arguments are left, return the predicate
funTransArgs([], [], Head, Head).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Translating propositional predicates                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tells whether a predicate-list is a proposition
isProposition([Op|Args]) :- not(isFunction([Op|Args])).

% Predicates that do not get a context
% noContext(+Atom)
noContext([X|_]) :-
   member(X,[equal,instance,subclass,contraryAttribute,
             disjoint,disjointdecomposition,disjointrelation,domain,
             domainsubclass,exhaustiveattribute,
             exhaustivedecomposition,inverse,partition,range,
             rangesubclass,subattribute,subrelation]).

% Functions do not get a context
noContext(X) :- isFunction(X).

% Finds and returns the (only) propositional argument
```

```
% getPropArg(+ArgumentList,-PropositionalArgument)
getPropArg([F|_R],F) :-
   isProposition(F), !.

getPropArg([_F|R],Result) :-
   getPropArg(R, Result).

% Translates a list of formulae
% contextTransList(+FormulaList,+PropPredList,-ResultFList)
contextTransList([F|R],PropPreds,[F2|R2]) :-
   contextTrans(F,PropPreds,F2),
   contextTransList(R,PropPreds,R2).

contextTransList([],_PropPreds,[]).

% Add a list of embracing propositional predicates
% contextTrans(+Formula,-ResultFormula)
contextTrans(A, A2) :- contextTrans(A, [], A2).

% contextTrans(+Formula,+EmbracingPropPredList,-ResultFormula)

% "not" with "holdsduring": distribute
contextTrans([holdsduring,T,[not,A]],PropPreds,R) :- !,
   contextTrans([not,[holdsduring,T,A]],PropPreds,R).

% "and", "all" and "ex": distribute
% Use simplified translation, because propositional attitudes
% do not occur deeply nested in SUMO
contextTrans([and|As], PropPreds, [and|As2]) :- !,
   contextTransList(As,PropPreds,As2).

contextTrans([exists, Var,A], PropPreds, [exists, Var,A2]) :- !,
   contextTrans(A,PropPreds, A2).

contextTrans([forall, Var,A], PropPreds, [forall, Var,A2]) :- !,
   contextTrans(A,PropPreds, A2).
```

```
% "or" in mainlevel: distribute
contextTrans([or|As], [], [or|As2]) :- !,
   contextTransList(As,[],As2).

% "or" with "holdsduring": distribute
contextTrans([or|As], [[[holdsduring,T,A],P]|R], [or|As2]) :- !,
   contextTransList(As,[[[holdsduring,T,A],P]|R],As2).

% "or" in context: double implication
contextTrans([or, A,B], PropPreds, Result) :- !,
   nnf([not, A], NotA),
   nnf([not, B], NotB),
   contextTrans(NotA,PropPreds,NotA2),
   contextTrans(NotB,PropPreds,NotB2),
   contextTrans(A,PropPreds,A2),
   contextTrans(B,PropPreds,B2),
   R=[and, [or,[not,NotA2],B2],[or,[not,NotB2],A2],[or,A2,B2]],
   nnf(R,Result).

% "or" with multiple arguments: distribute
contextTrans([or,A|As], PropPreds, Result) :- !,
   contextTrans([or,A,[or|As]], PropPreds, Result).

% Negation on main level: negate
contextTrans([not,P],[],Result) :-
   contextTrans(P,[],R),
   nnf([not,R],Result).

% Propositional argument: Recurse on it
contextTrans(P,PropPreds,Result) :-
   (P=[not,[Pred|Args]]       -> getPropArg(Args,PropArg) ;
    (P=[Pred|Args],Pred\=not) -> getPropArg(Args,PropArg)), !,
   replace(P,PropArg,*,P2),
   append(PropPreds,[P2], PropPreds2),
   nnf(PropArg,PropArg2),
   contextTrans(PropArg2, PropPreds2,Result).
```

```
% No propositional argument: call contextTransPred
contextTrans(P,PropPreds,Result) :-
   contextTransPred(P,PropPreds,mainlevel,Result).

% contextTransPred(+Atom,+PropPredList,+Context,-Result)
% With propositional predicates: Put them, call again
contextTransPred(P,[PropPred|R],Context,Result) :-
    newVar(Z),
    replace(PropPred,*,Z,PropPred2),
    contextTransPred(PropPred2, [], Context, PropPred3),
    contextTransPred(P,R,Z,P2),
    Result=[exists, [Z], [and, PropPred3, P2]].

% Without propositional predicates: Set into context
contextTransPred([not,[Pred|Args]],[],_,[not,[Pred|Args]]) :-
   noContext([Pred|Args]), !.

contextTransPred([Pred|Args],[],_,[Pred|Args]) :-
   noContext([Pred|Args]), !.

contextTransPred([not,[Pred|Args]],[],mainlevel,
                 [not,[Pred|Args2]]) :- !,
   append(Args,[mainlevel],Args2).

contextTransPred([not,[Pred|Args]],[],Context,[Pred|Args2]) :- !,
   append(Args,[neg(Context)],Args2).

contextTransPred([Pred|Args],[],Context,[Pred|Args2]) :- !,
   append(Args,[Context],Args2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Translating Class Creating Functions                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ccFunTrans(+Formula,-ResultFormula)
ccFunTrans(A,Result) :-
    ccFunTrans(Kout,[],A,A2),
```

```
    (Kout=[] ->
         Result=A2 ;
         Result=[and,A2|Kout]).

% Translate a list of formulae
% ccFunTransList(-ClassDefs,+Variables,+FormulaList,-ResultFList)
ccFunTransList(Kout,V,[F|R],[F2|R2]) :-
    ccFunTrans(Kout1,V,F,F2),
    ccFunTransList(Kout2,V,R,R2),
    append(Kout1,Kout2,Kout).

ccFunTransList([],_V,[],[]).

% ccFunTrans(-ClassDefs,+Variables,+Formula,-ResultFormula)

% Operators: distribute
ccFunTrans(Kout,V,[and|As], [and|As2]) :- !,
    ccFunTransList(Kout,V,As,As2).

ccFunTrans(Kout,V,[or|As], [or|As2]) :- !,
    ccFunTransList(Kout,V,As,As2).

ccFunTrans(Kout,V,[not,A], [not,A2]) :- !,
    ccFunTrans(Kout,V,A,A2).

% Quantifiers: collect variables
ccFunTrans(Kout,V,[exists, Vars,A], [exists,Vars,A2]) :- !,
   append(Vars,V,Vars2),
   ccFunTrans(Kout,Vars2,A,A2).

ccFunTrans(Kout,V,[forall, Vars,A], [forall,Vars,A2]) :- !,
   append(Vars,V,Vars2),
   ccFunTrans(Kout,Vars2,A,A2).

% Predicates: translate the arguments
ccFunTrans(Kout,V,[Name|Args], [Name|Args2]) :-
    ccFunTransArgs(Kout,V,Args,Args2).
```

```
% ccFunTransArgs(-ClassDefs,+Variables,+Atom,-ResultAtom)

% extensionFn arguments: create class definition
ccFunTransArgs([ClassDef|Kout],V,[[fun,extensionfn,Attribute]|R],
               [ClassName|R2]):- !,
    newVar(ClassId),
    ClassName=class(ClassId),
    newVar(X),
    ClassDef=[forall, [X],
                [or,[not,[attribute,X,Attribute]],
                    [instance,X,ClassName]]],
    ccFunTransArgs(Kout,V,R,R2).

% unionFn arguments: create class definition
ccFunTransArgs([ClassDef|Kout],V,[[fun,unionfn,Class1,Class2]|R],
               [ClassName|R2]):- !,
    newVar(ClassId),
    ClassName=class(ClassId),
    newVar(X),
    ClassDef=[forall, [X],[and,
                            [or,[not,[instance,X,Class1]],
                                [instance,X,ClassName]],
                            [or,[not,[instance,X,Class2]],
                                [instance,X,ClassName]]
             ]            ],
    ccFunTransArgs(Kout,V,R,R2).

% kappaFn arguments: create class definition
ccFunTransArgs([ClassDef|Kout],V,[[fun,kappafn,X,KappaFormula]|R],
               [ClassName|R2]):- !,
    newVar(ClassId),
    flatten(KappaFormula,KappaFormulaFlat),
    intersection(V,KappaFormulaFlat,FreeVarsX),
    delete(FreeVarsX,X,FreeVars),
    (FreeVars=[] ->
         ClassName=class(ClassId) ;
```

```
          ucaseVarList(FreeVars,V3),
          concat_atom(V3, ', ',V4),
          concat_atom([class,'(',ClassId,', ',V4,')'],ClassName)
    ),
    nnf([not,KappaFormula],NotKappaFormula),
    ClassDef=[forall, [X|V],[and,
                      [or,KappaFormula,[not,[instance,X,ClassName]]],
                      [or,NotKappaFormula,[instance,X,ClassName]]]],
    ccFunTransArgs(Kout,V,R,R2).

% Function arguments: parse arguments
ccFunTransArgs(Kout,V,[[fun,F|FArgs]|R],[[fun,F|FArgs2]|R2]):- !,
    ccFunTransArgs(Kout1,V,FArgs,FArgs2),
    ccFunTransArgs(Kout2,V,R,R2),
    append(Kout1,Kout2,Kout).

% Other arguments: copy
ccFunTransArgs(Kout,V,[F|R],[F|R2]):-
    ccFunTransArgs(Kout,V,R,R2).

% Empty arguments: copy
ccFunTransArgs([],_V,[],[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Skolemization                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Creates a skolem term
% makeSkolem(+SkolemName,+UniversalVars,+FormulaId,-Result)
makeSkolem(F,[],Id,[fun,F,Id]) :-!.

makeSkolem(F,UV,Id,[fun,F,Id|UVUp]) :-
  ucaseVarList(UV,UVUp).

% Tell whether a variable is quantified, return preceding universals
% quantified(+Quantifier,+Variable,+VariableIdList,-PreceedingUVs)
quantified(Q,V,[(Q,L)|_],[]) :-
```

```
    member(V,L), !.

quantified(Q,V,[(forall,L)|R],L2) :-
    quantified(Q,V,R,L1),append(L1,L,L2).

quantified(Q,V,[(exists,_)|R],L1) :-
    quantified(Q,V,R,L1).

% Translate a list of formulae
% skolemTransList(+FormulaList,+VariableIdList,+FormulaId,-Result)
skolemTransList([F|R],Vars,Id,[F2|R2]) :-
    skolemTrans(F,Vars,Id,F2),
    skolemTransList(R,Vars,Id,R2).

skolemTransList([],_,_,[]).

% skolemTrans(+Formula,-Result)
skolemTrans(A,A2) :-
    newVar(Id),
    skolemTrans(A,[],Id,A2).

% skolemTrans(+Formula,+VariableIdList,+FormulaId,-ResultFormula)

% Operators: distribute
skolemTrans([and|As],Vars,Id,[and|As2]) :- !,
    skolemTransList(As,Vars,Id,As2).
skolemTrans([or|As],Vars,Id,[or|As2]) :- !,
    skolemTransList(As,Vars,Id,As2).
skolemTrans([not,A],Vars,Id,[not,A2]) :- !,
    skolemTrans(A,Vars,Id,A2).

% Universal quantifier: collect variables
skolemTrans([forall,V,A],Vars,Id,A2) :- !,
   append(Vars,[(forall,V)],Vars2),
   skolemTrans(A,Vars2,Id,A2).

% Single existential quantifier: "All x: equal(x,<skolem>) => ..."
```

```
% Skolemize to "<variableName>(<formulaIdentifier>, ...)"
% if skolemterms(off)
skolemTrans([exists,[V|RV],A],Vars,Id,
            [or, [not,[equal,Vup,Skolem]], A2]) :- !,
   append(Vars,[(exists,[V])],Vars2),
   ucaseVar(V,Vup),
   (skolemterms(on) ->
        quantified(exists,V,Vars2,UV) ;
        UV=[]),
   makeSkolem(V,UV,Id,Skolem),
   skolemTrans([exists,RV,A],Vars2,Id,A2).

% Empty sequence of existential quantifiers: distribute
skolemTrans([exists,[],A],Vars,Id,A2) :- !,
   skolemTrans(A,Vars,Id,A2).

% Scan arguments of predicates
skolemTrans([PName|Args],Vars,Id,[PName|Args2]) :-
   skolemTransArgs(Args,Vars,Id,Args2).

% skolemTransArg(+ArgList,+VariableIdList,+FormulaId,-Result)

% Function arguments: distribute
skolemTransArgs([[fun,FName|FArgs]|R],Vars,Id,
                [[fun,FName|FArgs2]|R2]) :-
  skolemTransArgs(FArgs,Vars,Id,FArgs2),
  skolemTransArgs(R,Vars,Id,R2).

% Quantified variables: upcase
skolemTransArgs([F|R],Vars,Id,[Fup|R2]) :-
  quantified(_,F,Vars,_), !,
  ucaseVar(F,Fup),
  skolemTransArgs(R,Vars,Id,R2).

% Other arguments: copy
skolemTransArgs([F|R],Vars,Id,[F|R2]) :-
  skolemTransArgs(R,Vars,Id,R2).
```

```
skolemTransArgs([],_,_,[]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Clausal transformation                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Translate a list of formulae to clausal form
% clauseTransList(+FormulaList,-ResultCNF)
clauseTransList([F|R],[F2|R2]) :-
   clauseTrans(F,F2),
   clauseTransList(R,R2).

clauseTransList([],[]).

% clauseTrans(+Formula,-ResultCNF)

% Conjunctions: Append
clauseTrans([and,F|R],Result) :- !,
   clauseTrans(F,FCNF),
   clauseTrans([and|R],RCNF),
   append(FCNF,RCNF,Result).

clauseTrans([and],[]) :-!.

% Disjunctions: Recurse
clauseTrans([or,F|R],Result) :- !,
   clauseTrans(F,FCNF),
   clauseTrans([or|R],RCNF),
   disjunctionCnfCnf(FCNF,RCNF,Result).

clauseTrans([or],[[]]) :-!.

% Predicates: Collect
clauseTrans(Predicate,[[Predicate]]).
```

```
% Handle a disjunction of CNFs
% disjunctionCnfCnf(+CNF,+CNF,-ResultCNF)
disjunctionCnfCnf([F|R],CNF,Result) :-
   disjunctionDisjCnf(F,CNF,FCNF),
   disjunctionCnfCnf(R,CNF,RCNF),
   append(FCNF,RCNF,Result).

disjunctionCnfCnf([],_CNF,[]).

% A disjunction of a disjunction and a CNF
% disjunctionCnfCnf(+Disjunction,+CNF,-ResultCNF)
disjunctionDisjCnf(Disj,[F|R],[FDisj|Result]) :-
   append(Disj,F,FDisj),
   disjunctionDisjCnf(Disj,R,Result).

disjunctionDisjCnf(_Disj,[],[]).
```

A.2 Pre-Processing of the F-Structure

This Java-program, `Fef2tme.java`, translates an F-Structure to a DLP.

```
/**        Class Fef2tme
   Translates a F-Structure-File (*.fef) to
   a KRHyper-file (*.tme)
*/

import java.io.*;
import java.util.*;

public final class Fef2tme {

  /** Contains the In-File */
  private static BufferedReader in;
  /** Contains the Out-File */
  private static BufferedWriter out;
  /** Counts lines in in-file */
  private static int lines=0;
  /** Holds predicates for SUMO-annotations*/
```

```
private static List sumoPreds=new LinkedList();
static {
  sumoPreds.add("milo_syn");
  sumoPreds.add("sumo_syn");
  sumoPreds.add("milo_sub");
  sumoPreds.add("sumo_sub");
  sumoPreds.add("milo_inst");
  sumoPreds.add("sumo_inst");
}
/** Holds predicates to be ignored when writing*/
private static Collection ignore=new TreeSet();
static {
  ignore.add("string");
  ignore.add("rel");
  ignore.add("ont");
  ignore.add("sslink");
  ignore.add("fe");
}
/** Holds specific SUMO-translations */
private static Map sumoTranslations=new HashMap();
static {
  sumoTranslations.put("crime_n_1", "criminalaction");
  sumoTranslations.put("accuse_v_1", "judicialprocess");
  sumoTranslations.put("confirm_v_1", "stating");
  sumoTranslations.put("asset_n_1", "object");
  sumoTranslations.put("bear_v_10", "sexualreproduction");
  sumoTranslations.put("name_n_1", "name");
  sumoTranslations.put("group_n_2", "group");
  sumoTranslations.put("population_n_4", "group");
  sumoTranslations.put("cancel_v_3", "removing");
  sumoTranslations.put("company_n_4", "organization");
  sumoTranslations.put("action_n_8", "process");
  sumoTranslations.put("activist_n_1", "human");
  sumoTranslations.put("spend_v_1", "giving");
  sumoTranslations.put("hurt_v_7", "injuring");
  sumoTranslations.put("fight_v_1", "destruction");
  sumoTranslations.put("raise_v_2", "increasing");
```

```
    sumoTranslations.put("boost_v_3", "increasing");
    sumoTranslations.put("assassinate_v_2", "killing");
    sumoTranslations.put("double_v_6", "increasing");
    sumoTranslations.put("rescue_v_1", "removing");
  }
  /** Holds SUMO-classes for F-identifiers */
  private static Map sumoClasses=new HashMap();
  /** Holds words corresponding to f-identifiers */
  private static Map preds=new HashMap();
  /** Holds all ont-relations */
  private static Map onts=new HashMap();
  /** Holds all sslink-relations */
  private static Map sslinks=new HashMap();
  /** Holds all names and locations */
  private static Collection names=new TreeSet();


  /*****************************************************************
   *                      Helper Routines                          *
   *****************************************************************/

  /** Normalizes a Sumo-identifier */
  private static String n(String s) {
    // Special cases
    if(s==null) return("null");
    if(s.length()==0) return("nil");
    // Convert numbers to identifiers
    if("-+0123456789.".indexOf(s.charAt(0))!=-1) s="num"+s;
    // Special characters
    //  We don't use isUpperCase etc. because we want umlauts to
    //  disappear
    StringBuffer result=new StringBuffer(s);
    for(int i=0;i<result.length();i++) {
      if(result.charAt(i)>='A' && result.charAt(i)<='Z') {
        result.setCharAt(i,Character.toLowerCase(result.charAt(i)));
        continue;
      }
```

```
    if(result.charAt(i)>='a' && result.charAt(i)<='z' ||
       result.charAt(i)>='0' && result.charAt(i)<='9' ||
       result.charAt(i)=='(' || result.charAt(i)==')') continue;
    result.setCharAt(i,'_');
  }
  while(result.length()>0 && result.charAt(0)=='_')
    result.delete(0,1);
  while(result.length()>0 && result.charAt(result.length()-1)=='_')
    result.setLength(result.length()-1);
  if(result.length()==0) return("nil");
  if(s.charAt(0)=='?')
    result.setCharAt(0,Character.toUpperCase(result.charAt(0)));
  return(result.toString());
}

/** Returns next item in the file, as delimited by any character
    of <delims>. Returns the item as a string and the actual
    delimiter in delim[0]. */
private static String getItem(String delims, int[]delim)
                                                  throws IOException {
  StringBuffer b=new StringBuffer();
  int c=' ';
  int brdepth=0;
  // Read to next item
  while(c==' ' || c=='\n' || c=='\r')
     if((c=in.read())=='\n') lines++;
  // Read until we reach a delim
  while(((c==')' && brdepth>=0) || delims.indexOf(c)==-1)
                                                    && c!=-1) {
    if(c=='\'' || c=='"') {
      while(true) {
        c=in.read();
        if(c=='\'' || c=='"') {
          if(b.length()>0 && b.charAt(b.length()-1)=='\\')
            b.setLength(b.length()-1);
          else
            break;
```

```
        }
        b.append((char)c);
      }
      c=in.read();
    } else {
      b.append((char)c);
      if((c=in.read())=='\n') lines++;
    }
    if(c=='(') brdepth++;
    if(c==')') brdepth--;
  };
  delim[0]=c;
  return(b.toString());
}

/** Reads an atom functor(first, second, third) */
private static String[] readClause() throws IOException {
    int[]delim=new int[1];
    String functor=n(getItem("(",delim));
    if(delim[0]==-1) return(null);
    String first=n(getItem(",)",delim));
    String second=null;
    if(delim[0]==',') second=n(getItem(",)",delim));
    String third=null;
    if(delim[0]==',') third=n(getItem(")",delim));
    getItem(".",delim);
    return(new String[]{functor,first,second,third});
}

/*****************************************************************
 *                      Parsing                                  *
 *****************************************************************/

/** Reads the FEF-file and collects relations */
private static void firstParse() throws IOException {
  while(true) {
    String[] s=readClause();
```

```
if(s==null) break;
// Register onts
if(s[0].equals("ont")) {
  onts.put(s[2],s[1]);
  continue;
}
// Register sslinks
if(s[0].equals("sslink")) {
  sslinks.put(s[2],s[1]);
  continue;
}
// Register SUMO annotations
if( s[0].startsWith("sumo_") ||
    s[0].startsWith("milo_")) {
  // Register translation, if better
  String t=(String)sumoClasses.get(s[1]);
  int myQuality=sumoPreds.indexOf(s[0])+1;
  if(t==null || myQuality<((int)t.charAt(0))-'0')
     sumoClasses.put(s[1], myQuality+s[2]);
  continue;
}
// Register a wordnet synonym as SUMO translation,
// if it appears in sumoTranslations
if(s[0].equals("wn_syn") && sumoTranslations.get(s[2])!=null){
  sumoClasses.put(s[1], "0"+sumoTranslations.get(s[2]));
  continue;
}
// Register preds
if(s[0].equals("pred")) {
  preds.put(s[1],s[2]);
  continue;
}
// Register proper names
if(s[0].equals("proper")) {
  if(s[2].equals("name") || s[2].equals("location"))
     names.add(s[1]);
  continue;
```

```
    }
  }
}

/** Reads FEF-file again and writes it */
private static void secondParse() throws IOException {
  // Collect and print SUMO-classes
  Set hasSumoClass=new TreeSet();
  Iterator si=sumoClasses.keySet().iterator();
  while(si.hasNext()) {
    String id=(String)si.next();
    String sumo=(String)sumoClasses.get(id);
    // Undo ont
    id=(String)onts.get(id);
    // Undo sslink
    id=(String)sslinks.get(id);
    // Register
    hasSumoClass.add(id);
    // Write SUMO-class
    if(names.contains(id)) id=(String)preds.get(id);
    out.write("sumo("+id+", "+sumo.substring(1)+").\n");
  }
  // Collect all identifiers without SUMO-class
  si=preds.keySet().iterator();
  while(si.hasNext()) {
    String id=(String)si.next();
    if(!hasSumoClass.contains(id) &&
      ((String)preds.get(id)).length()>3) names.add(id);
  }
  // Run through the FEF-file again
  while(true) {
    String[] s=readClause();
    if(s==null) break;
    if(ignore.contains(s[0]) || s[1]==null
       || s[2]==null || s[3]!=null
       || sumoPreds.contains(s[0])) continue;
    // Flatten the pointers
```

```
      for(int i=1;i<4 && s[i]!=null; i++) {
        // Undo onts
        if(onts.get(s[i])!=null) s[i]=(String)onts.get(s[i]);
        // Undo sslinks
        if(sslinks.get(s[i])!=null) s[i]=(String)sslinks.get(s[i]);
        // Replace, if name
        if(names.contains(s[i])) s[i]=(String)preds.get(s[i]);
      }
      // Normalize wordnet synsets
      if(s[0].equals("wn_syn")) {
        // Split WordNet synset     wn_syn(f(n), "xxx#x#n")
        // to                       wordnet(f(n), xxx, x, n)
        // Read from backwards, because xxx may contain '_'
        s[0]=s[1];
        s[1]=s[2];
        s[3]=s[1].substring(s[1].lastIndexOf('_')+1);
        s[1]=s[1].substring(0,s[1].lastIndexOf('_'));
        s[2]=s[1].substring(s[1].lastIndexOf('_')+1);
        s[1]=s[1].substring(0,s[1].lastIndexOf('_'));
        out.write("wordnet("+s[0]+", "+s[1]+", "+s[2]+", "+s[3]+").\n");
        continue;
      }
      out.write(s[0]+"("+s[1]+", "+s[2]);
      if(s[3]!=null) out.write(", "+s[3]);
      out.write(").\n");
    }
    // Write preds
    Iterator pi=preds.keySet().iterator();
    while(pi.hasNext()) {
      String id=(String)pi.next();
      String pred=(String)preds.get(id);
      if(names.contains(id)) id=pred;
      if(id.startsWith("f(")) out.write("pred("+id+", "+pred+").\n");
      else                    out.write("pred("+id+", "+pred+").\n");
    }
  }
```

```
  /*****************************************************************
   *                      Main                                     *
   *****************************************************************/

  /** Reads File and transforms it */
  public static void main(String[] argv) throws IOException {
    if(argv!=null && (argv.length==0 || argv[0].endsWith("?") ||
       argv[0].startsWith("-"))) {
      System.err.println(
       "\n                Fef2tme\n\n"+
       "    Translates an F-Structure-File (*.fef) to a\n"+
       "    KRHyper-file (*.tme).\n\n"+
       "Call: Fef2tme <FEF-file>[.fef]\n");
      return;
    }
    if(argv[0].indexOf('.')==-1) argv[0]=argv[0]+".fef";
    String outfile=argv[0].substring(0,argv[0].lastIndexOf("."))
                  +".tme";
    // First Parse
    System.err.print("First parse of "+argv[0]+"... ");
    in=new BufferedReader(new FileReader(argv[0]));
    firstParse();
    in.close();

    // Second Parse
    System.err.print("Done\nCreating "+outfile+"... ");
    out=new BufferedWriter(new FileWriter(outfile));
    System.err.print("Done\nSecond parse of "+argv[0]+"... ");
    in=new BufferedReader(new FileReader(argv[0]));
    secondParse();
    out.close();
    in.close();
    System.err.println("Done");
  }

}
```

A.3 Role Assignment

This DLP, `fparse.tme`, does a role assignment when run together with the pre-processed f-structure.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                     fparse.tme                                                %
%                                                                               %
%                                                                               %
% Heuristic role assigment from the F-structure to SUMO roles.                  %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Predicates that are local to this role assignment program carry
% an underscore

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   General                                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Make F-identifiers SUMO instances
instance(F,Class) :-
    sumo(F, Class),
    not(bad_class(Class)),
    not(is_attribute(F)).

% Classes which shall not create an "instance"
bad_class(subjectiveassessmentattribute).
bad_class(normativeattribute).
bad_class(nation).
bad_class(occupationalrole).
bad_class(socialrole).
bad_class(ethnicgroup).
bad_class(believes).
bad_class(wants).
bad_class(causes).
bad_class(consciousnessattribute).
```

```
bad_class(position).
bad_class(positionalattribute).
bad_class(soundattribute).

% Occupational roles create new classes
instance(F,class(Role)) :-
    sumo(F, occupationalrole),
    pred(F,Role).
subclass(class(X), human).

% Social roles create new classes
instance(F,class(Role)) :-
    sumo(F, socialrole),
    pred(F,Role).

% Positions create new classes
instance(F,class(Role)) :-
    sumo(F, position),
    pred(F,Role).

% Context is "mainlevel" at the root
given_context(f(0), mainlevel).
given_context(f(1), mainlevel).

% Default to "mainlevel" if there is no given context
context_(F,C) :-
    given_context(F,C).
context_(F,mainlevel) :-
    pred(F,_),
    not(has_given_context(F)).
has_given_context(F) :-
    given_context(F,_).

% mainlevel context holds now
holdsduring(now, mainlevel, mainlevel).

% Negate contexts
```

```
negated(C) :-
  context_(F,C),
  adjunct(F,Not),
  pred(Not,not),
  adjunct_type(Not,negative).

% Use transfer for "X is Y"
transfer_(X,Y) :-
   xcomp(Be, Y),
   pred(Be, be),
   dsubj(Be, X).

% Instance transfer
instance(X,C) :- transfer_(X,Y), instance(Y,C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Attributes                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Choose the attribute name for an attribute
attribute_name(F,Name) :-
    sumo(F,Name),
    not(bad_class(Name)).
attribute_name(F,Name) :-
    sumo(F,C),
    bad_class(C),
    pred(F,Name).
attribute_name(F,Name) :-
    not(has_sumo(F)),
    pred(F,Name).
has_sumo(F) :- sumo(F,_).

% Adjectives become attributes
attribute(Object, Attribute, Context) :-
    adjunct(Object, Adjective),
    attribute_name(Adjective, Attribute),
    atype(Adjective, attributive),
```

```
    context_(Object, Context).

% Adjectives with "to be" become attributes
attribute(Object, Attribute, Context) :-
    xcomp(Be, Adjective),
    pred(Be, be),
    dsubj(Be, Object),
    attribute_name(Adjective, Attribute),
    context_(Be, Context).

% "X was treated as a dignitary"
attribute(Object,Attribute, by(F)) :-
    subj(As,Object),
    ptype(As,semantic),
    pred(As,as),
    obj(As,Attribute),
    dobj(F,Object).
% Overwrite is_attribute-rule in this case
instance(F,C) :-
    attribute(_,F,by(_)),
    sumo(F,C).

% "mod" often indicates attributes
attribute(Object, ModifierLexeme, mainlevel) :-
    mod(Object, Modifier),
    not(proper(Object,name)),   % Lord XXX, Western Japan
    not(number_type(Modifier,cardinal)), % November 25
    pred(Modifier, ModifierLexeme).

% Any attribute is an is_attribute
is_attribute(F) :- attribute(Object, F        , Context).
is_attribute(F) :- attribute(Object, Attribute, Context),
    sumo(F, Attribute).

% Attribute transfer
attribute(X, Attribute, Context) :-
    transfer_(X,Y),
```

```
    attribute(Y, Attribute, Context).

% "Pleading innocent"
attribute(Agent, AdjectiveLexeme, stating(Agent, Context)) :-
    sumo(F, stating),
    agent(F, Agent, Context),
    xcomp(F, Adjective),
    pred(Adjective, AdjectiveLexeme),
    sumo(Adjective, Attribute),
    bad_class(Attribute).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Roles                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% dsubj -> agent
agent(F,Agent,Context) :-
    dsubj(F,Agent),
    not(pred(F,be)),
    not(pred(Agent,coord)),
    context_(F,Context).
given_context_a(Agent,Context) :-
    dsubj(F,Agent),
    not(pred(F,be)),
    not(pred(Agent,coord)),
    context_(F,Context).
context_(F,C):-given_context_a(F,C).

% adjunct with processes -> same agent
agent(F, Agent, Context) :-
    sumo(F, process),
    adjunct(F,F2),
    agent(F2, Agent, Context).

% dobj -> patient
patient(F, Patient,Context) :-
    dobj(F,Patient),
```

```
    not(pred(Patient,coord)),
    context_(F,Context).
given_context_p(Patient,Context) :-
    dobj(F,Patient),
    not(pred(Patient,coord)),
    context_(F,Context).
context_(F,C):-given_context_p(F,C).

% "about" -> patient
patient(F, Topic, Context) :-
    obl(F, About),
    ptype(About,semantic),
    pred(About,about),
    obj(About,Topic),
    context_(F, Context).

% "requesting for X" -> patient
patient(F, Patient, Context) :-
    sumo(F, requesting),
    obl(F, For),
    ptype(For,semantic),
    pred(For,for),
    obj(For,Patient),
    context_(F, Context).

% Passive newsthesis headline -> patient
patient(F, Patient, mainlevel) :-
    adjunct(Patient, F),
    dobj(F,Null),
    pron_type(Null,null).

% "X who is Y" -> patient
patient(F, Patient, Context) :-
   adjunct(Patient, F),
   dobj(F, Who),
   pron_form(Who, who),
   context_(Patient, Context).
```

```
% "to" as adjunct -> destination
destination(F, Destination, Context) :-
    adjunct(F,To),
    ptype(To,semantic),
    pred(To,to),
    adv_type(To,vpadv),
    obj(To,Destination),
    context_(F, Context).

% "to" as obl -> destination
destination(F, Destination, Context) :-
    obl(F,To),
    ptype(To,semantic),
    pred(To,to),
    obj(To,Destination),
    context_(F, Context).

% "as" -> destination
destination(F, Destination, Context) :-
    obl(F,As),
    ptype(As,semantic),
    pred(As,as),
    obj(As,Destination),
    context_(F, Context).

% "against" -> destination
destination(F, Destination, Context) :-
    adjunct(F,Against),
    ptype(Against,semantic),
    pred(Against,against),
    obj(Against,Destination),
    adv_type(Against,vpadv),
    context_(F, Context).

% Undo "a number of" obj
dobj(F,Object) :-
```

```
    pred(N,a_number),
    adjunct(N,Of),
    ptype(Of,semantic),
    pred(Of,of),
    obj(Of,Object),
    adjunct_type(Of,nominal),
    dobj(F,N).

% Undo "a number of" subj
dsubj(F,Object) :-
    pred(N,a_number),
    adjunct(N,Of),
    ptype(Of,semantic),
    pred(Of,of),
    obj(Of,Object),
    adjunct_type(Of,nominal),
    dsubj(F,N).

% Undo "coord" obj
dobj(F,Subject) :-
    dobj(F,Coord),
    pred(Coord,coord),
    coord_form(Coord,and),
    conj(Coord,Subject).

% Undo "coord" subj
dsubj(F,Subject) :-
    dsubj(F,Coord),
    pred(Coord,coord),
    coord_form(Coord,and),
    conj(Coord,Subject).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    Time and Location                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% proper location -> geographicregion
instance(Location, geographicregion) :-
    proper(Location, location).

% tense fut -> whenfn
equal(fun(whenfn,F), future) :-
    tense(F, fut),
    sumo(F, Something).

% tense past -> whenfn
equal(fun(whenfn,F), past) :-
    tense(F, past),
    sumo(F, Something).

% tense pres -> whenfn
equal(fun(whenfn,F), now) :-
    tense(F, pres),
    sumo(F, Something).

% On November 25, 2001
equal(fun(whenfn,F), date(YearNumber, MonthLexeme, DayNumber)) :-
    adjunct(F, On),
    pred(On,on),
    obj(On,Month),
    adv_type(On,vpadv),
    proper(Month,date),
    pred(Month,MonthLexeme),
    mod(Month,Day),
    pred(Day,DayNumber),
    number_type(Day,cardinal),
    adjunct(F, Year),
    pred(Year,YearNumber),
    number_type(Year,cardinal).

% "in" -> located
located(F, Location, Context) :-
    adjunct(F,In),
```

```
    ptype(In,semantic),
    adv_type(In,vpadv),
    pred(In, in),
    obj(In, Location),
    context_(F, Context).

% "at" -> located
located(F, Location, Context) :-
    adjunct(F,In),
    ptype(In,semantic),
    adv_type(In,vpadv),
    pred(In, at),
    obj(In, Location),
    context_(F, Context).

% "Guantanamo, Cuba" -> located
located(Location, Country, mainlevel) :-
    adjunct(Location,Country),
    proper(Country, location).

% at -> located
located(F, Location, Context) :-
   adjunct(F,At),
   ptype(At,semantic),
   pred(At,at),
   obj(At,Location),
   adv_type(At,vpadv),
   context_(F, Context).

% in -> located
located(F, Location, Context) :-
   adjunct(F,In),
   ptype(In,semantic),
   pred(In,in),
   obj(In,Location),
   context_(F, Context).
```

```
% Location transfer
located(X, Place, Context) :-
    transfer_(X,Y),
    located(Y, Place, Context).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Additional Information                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% "thanks to X" -> causes
causes(Agent, F, Context) :-
    adjunct(F,Thanks),
    pred(Thanks,thanks),
    adjunct_type(Thanks,parenthetical),
    adjunct(Thanks,To),
    ptype(To,semantic),
    pred(To,to),
    obj(To,Agent),
    adjunct_type(To,nominal),
    context_(F, Context).

% "because of X" -> causes
causes(Agent, F, Context) :-
    adjunct(F,Because),
    pred(Because,because),
    obj(Because,Of),
    ptype(Of,semantic),
    pred(Of,of),
    obj(Of,Agent),
    context_(F,Context).

% Causing -> causes
causes(Cause,Result,Context) :-
    sumo(F,causes),
    dsubj(F,Cause),
    dobj(F,Result),
```

```
    context_(F,Context).

% wants -> desires
desires(Agent, wants(Agent), Context) :-
    sumo(F, wants),
    dsubj(F, Agent),
    xcomp(F, Fact),
    context_(Agent, Context).
given_context(Fact, wants(Agent)) :-
    sumo(F, wants),
    dsubj(F, Agent),
    xcomp(F, Fact).

% communication -> set context
patient(F, said(F), Context) :-
    communication_(F),
    context_(F, Context).
given_context(Fact, said(F)) :-
    communication_(F),
    comp(F, Fact).
communication_(F) :- sumo(F, communication).
communication_(F) :- sumo(F, reasoning).
communication_(F) :- sumo(F, writing).
communication_(F) :- sumo(F, stating).

% Believes -> believes
believes(Agent, believes(Agent), Context) :-
    sumo(F, believes),
    dsubj(F, Agent),
    dobj(F, Fact),
    context_(Agent, Context).
given_context(Fact, believes(Agent)) :-
    sumo(F, believes),
    dsubj(F, Agent),
    dobj(F, Fact).

% poss -> possesses
```

```
possesses(Agent,Object,mainlevel) :-
    poss(Object,Agent),
    not(pred(Agent, a_number)).  % "A number of.."

% "of" -> possesses
possesses(Agent, Object, mainlevel) :-
   adjunct(Object,Of),
   ptype(Of,semantic),
   pred(Of,of),
   obj(Of,Agent).
```

A.4 Negation of the Hypothesis

This Java-program, `Mdl2tme.java`, translates the model of the hypothesis to a DLP containing the negated hypothesis.

```
/**        class Mdl2tme
   Reads a KRHyper-model-file (*.mdl) and produces
   a negated hypothesis-file for KRHyper (*neg.tme)
*/

import java.io.*;
import java.util.*;

public final class Mdl2tme {

  /** Contains the In-File */
  private static BufferedReader in;
  /** Contains the Out-File */
  private static BufferedWriter out;
  /** Counts lines in in-file */
  private static int lines=0;
  /** Holds predicates to be kept*/
  private static Collection keep=new TreeSet();
  static {
    keep.add("instance");
    keep.add("negated");
    keep.add("agent");
```

```
  keep.add("origin");
  keep.add("destination");
  keep.add("patient");
  keep.add("instrument");
  keep.add("possesses");
  keep.add("attribute");
  keep.add("located");
  keep.add("believes");
  keep.add("desires");
}

/*****************************************************************
 *                      Helper Routines                          *
 *****************************************************************/

/** Determines whether a predicate is worth to be kept*/
private static boolean keepIt(String s) {
  return(s.endsWith("fn") || keep.contains(s));
}

/** Seeks a substring in a StringBuffer. This is simply
    StringBuffer.indexOf, which is not supported by my
    Java-Version */
private static int indexOf(StringBuffer b, String s1, int pos) {
  int s1pos=0;
  while(pos<b.length()) {
    if(b.charAt(pos)==s1.charAt(s1pos)) s1pos++;
    else { pos=pos-s1pos; s1pos=0; }
    if(s1pos==s1.length()) return(pos-s1pos+1);
    pos++;
  }
  return(-1);
}

/** Replaces a string by a substring*/
private static StringBuffer replaceAll(StringBuffer b,
                                       String s1, String s2) {
```

```
    int pos=-1;
    while((pos=indexOf(b,s1,pos+1))!=-1)
                            b.replace(pos, pos+s1.length(), s2);
    return(b);
  }

  /*****************************************************************
   *                       Parsing                                 *
   *****************************************************************/

  /** Turns all f(nn) into Fnn */
  private static StringBuffer upcaseF(StringBuffer b) {
    int pos=-1;
    while((pos=indexOf(b,"f(",pos+1))!=-1) {
      b.replace(pos,pos+2,"F");
      while(b.charAt(pos)!=')') pos++;
      b.delete(pos,pos+1);
    }
    return(b);
  }

  /** Reads the KRHyper-file and writes it */
  private static void doHyp() throws IOException {
    boolean polarity=true;
    int preds=0;
    out.write("false :-\n");
    while(true) {
      String s=in.readLine();
      if(s==null) break;
      if(s.length()==0 || s.charAt(0)=='%') continue;
      if(s.equals("negated(context)")) polarity=false;
      if(s.startsWith("pred")) preds++;
      String functor=s.substring(0,s.indexOf('('));
      if(!keepIt(functor)) continue;
      StringBuffer b=new StringBuffer(s);
      b.setLength(b.length()-1);
      upcaseF(b);
```

```
    replaceAll(b,", mainlevel",", Mainlevel");
    out.write("   ");
    out.write(b.toString());
    out.write(",\n");
  }
  if(polarity==true) out.write("   not(negated(Mainlevel)),\n");
  // If there are less than 3 preds, this indicates a buggy
  // f-structure: Make the hypothesis unprovable
  if(preds<3)   out.write("   elvis_is_a_hound_dog,\n");
  out.write("   holdsduring(now, Mainlevel, mainlevel).\n");
}

/*****************************************************************
 *                      Main                                     *
 *****************************************************************/

/** Reads File and transforms it */
public static void main(String[] argv) throws IOException {
  if(argv!=null && (argv.length==0 || argv[0].endsWith("?") ||
     argv[0].startsWith("-"))) {
    System.err.println(
      "\n                              Mdl2tme\n\n"+
      "       Reads a KRHyper-model-file (*.mdl) and produces\n"+
      "       a negated hypothesis-file for KRHyper (*neg.tme)."+
      "\n\nCall: Mdl2tme <KRHyper-model-file>[.mdl]\n");
    return;
  }
  if(argv[0].indexOf('.')==-1) argv[0]=argv[0]+".mdl";
  String outfile=argv[0].substring(0,argv[0].lastIndexOf("."))
                +"neg.tme";
  System.err.print("Creating "+outfile+"... ");
  out=new BufferedWriter(new FileWriter(outfile));
  System.err.print("Done\nParsing "+argv[0]+"...");
  in=new BufferedReader(new FileReader(argv[0]));
  doHyp();
  out.close();
  System.err.println("Done");
```

```
  }

}
```

References

[BDK+02] M. Butt, H. Dyvik, T. H. King, H. Masuichi, and C. Rohrer. The Parallel Grammar Project. In *Proceedings of COLING-2002 Workshop on Grammar Engineering and Evaluation*, pages 1–7, 2002.

[BFL98] C. F. Baker, C. J. Fillmore, and J. B. Lowe. The Berkeley FrameNet Project. In *Proceedings of the COLING-ACL*, 1998.

[BFN96] P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In J. J. Alferes, L. M. Pereira, and E. Orlowska, editors, *Proceedings of the European Workshop on Logics in Artificial Intelligence (JELIA 1996)*, pages 1–17. Springer, 1996.

[BG98] L. Bachmair and H. Ganzinger. Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction. A Basis for Applications*, volume I: Foundations. Calculi and Refinements, pages 353–398. Kluwer Academic Publishers, 1998.

[BGV97] Leo Bachmair, Harald Ganzinger, and Andrej Voronkov. Elimination of Equality via Transformation with Ordering Constraints. Research Report MPI-I-97-2-012, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, December 1997.

[BHS93] W. Bibel, S. Hölldobler, and T. Schaub. *Wissensrepräsentation und Inferenz: eine grundlegende Einführung.* Vieweg, 1993.

[BP03] S. Branerjee and T. Petersen. Extended Gloss Overlaps as a Measure of Semantic Relatedness. In *Proceedings of IJCAI 2003*, 2003.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4:412–430, 1975.

[Bre82] J. Bresnan. *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts, 1982.

[BS05] P. Baumgartner and F. M. Suchanek. Automated Reasoning Support for SUMO/KIF. Submitted to IJCAI 2005, available at http://www.mpi-sb.mpg.de/~baumgart/publications/, 2005.

[Cro05] R. Crouch. Packed Rewriting for Mapping Semantics to KR. In *Proceedings of the Sixth International Workshop on Computational Semantics, IWCS-06*, 2005.

[Fel98] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[GBZ+04] J. Graupmann, M. Biwer, C. Zimmer, C. Zimmer, M. Bender, M. Theobald, and G. Weikum. The COMPASS Search Engine for Unified Ranked Retrieval of Heterogenous XML and Web Data. In *Proceedings of the 30th VLDB Conference, Toronto, Canada*, 2004.

[Gen91] M. Genesereth. Knowledge Interchange Format. In J. Allen et al., editor, *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR-91)*, pages 238–249. Morgan Kaufman Publishers, 1991.

[Gua98] N. Guarino. Formal Ontology and Information systems. *Formal Ontology in Information Systems*, 1998.

[MCHM03] D. Moldovan, C. Clark, S. Harabagiu, and S. Maiorano. COGEX: A Logic Prover for Question Answering. In *Proceedings of HLT-NAACL 2003*, pages 87–93, 2003.

[NP01] I. Niles and A. Pease. Towards a Standard Upper Ontology. In Chris Welty and Barry Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.

[NP03] I. Niles and A. Pease. Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology. In *Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE 2003)*, 2003.

[Pea04] A. Pease. Standard Upper Ontology Knowledge Interchange Format. http://cvs.sourceforge.net/viewcvs.py/*checkout*/sigmakee/sigma/suo-kif.pdf, 2004.

[Sak90] C. Sakama. Possible Model Semantics for Disjunctive Databases. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proceedings First*

International Conference on Deductive and Object-Oriented Databases (DOOD-89), pages 337–351. Elsevier Science Publishers B.V., 1990.

[Sow00] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, 2000.

[Suc03] F. M. Suchanek. Representing Ontological Structures in CLOS, Java and FAST. http://www-lehre.informatik.uni-osnabrueck.de/~fsuchane/ba.doc, 2003.

[Wer03] C. Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, 2003.

Erklärung

Hiermit erkläre ich, Fabian M. Suchanek, die vorliegende Arbeit "Ontological Reasoning for Natural Language Understanding" selbstständig verfasst zu haben und keine anderen Quellen oder Hilfmittel als die angegebenen verwendet zu haben.

Saarbrücken, den 15.3.2005

Fabian M. Suchanek
Mat.Nr.: 2503721