Automated Reasoning Support for First-Order Ontologies

Peter Baumgartner[1] and Fabian M. Suchanek[2]

[1] National ICT Australia (NICTA)
Peter.Baumgartner@nicta.com.au
[2] Max-Planck Institute for Computer Science, Germany
suchanek@mpi-sb.mpg.de

Abstract. Formal ontologies play an increasingly important role in demanding knowledge representation applications like the Semantic Web. Regarding automated reasoning support, the mainstream of research focusses on ontology languages that are also Description Logics, such as OWL-DL. However, many existing ontologies go beyond Description Logics and use full first-order logic. We propose a novel transformation technique that allows to apply existing model computation systems in such situations. We describe the transformation and some variants, its properties and intended applications to ontological reasoning.

1 Introduction

1.1 Motivation

Recent years have seen an increasing interest in formal knowledge bases (KBs). Demanding application areas – notably the *Semantic Web* – will have to remain a vision without powerful automated reasoning support.

The mainstream of research on automated reasoning focuses on ontology languages that are also Description Logics (DLs), such as OWL-DL. Yet, there are good reasons to also consider larger fragments of first-order logic as ontology languages. One reason is the ability to add "rules" to the ontology, as in languages like SWRL [HB+04]. An example for a rule is the statement [GHVD03]: individuals who live and work at the same location are home workers. This can be expressed as a Horn rule (clause) $\mathsf{homeWorker}(x) \leftarrow \mathsf{work}(x,y) \wedge \mathsf{live}(x,z) \wedge \mathsf{loc}(y,w) \wedge \mathsf{loc}(z,w)$, but is not expressible in current DL systems.

Another reason for considering even full first-order logic is the existence of numerous KBs that go beyond Description Logics. One example is the largest formal public ontology available today, the Suggested Upper Merged Ontology SUMO [NP01]. SUMO is written in KIF, the Knowledge Interchange Format [KIF], which is basically first-order logic with equality and some higher-order features. Together with its domain-specific extensions, SUMO contains more than 20'000 terms and 60'000 axioms. Unfortunately, only limited automated reasoning is available today for first-order KBs. For instance, to our knowledge, the only theorem prover applied to SUMO so far is Vampire [RV01].

This situation seems somewhat surprising, given the demonstrated usefulness of description logic systems for KBs written in $\mathcal{ALC}$-like languages [BCM+02]. Why has

J.J. Alferes et al. (Eds.): PPSWR 2006, LNCS 4187, pp. 18–32, 2006.

this success story not been repeated for KBs in first-order logic? The answer from a technical point of view might be that DL systems are so successful because they usually *decide* the satisfiability problem of their input language. This is an important feature, as it allows, for instance, to prove that a speculated subsumption relation between concepts does *not* hold. Furthermore, it allows the "debugging" of KBs.

Although such decision procedures cannot exist for first-order KBs, reasoning support by automated theorem provers may be attempted nevertheless. Indeed, within the Semantic Web framework a number of off-the-shelf first-order theorem provers have been tested on various KBs, unsatisfiable ones and satisfiable ones[1]. The provers generally performed well in solving the unsatisfiable test cases. However, they often could not solve the satisfiable ones, i.e., they did not terminate.

1.2 Contribution

To address the problem of non-termination of the prover, we propose a novel transformation technique on first-order logic KBs that allows to compute models more often. Our transformation is rather general regarding the underlying system to be used. We target at model computation systems as developed within the logic programming community or at bottom-up clausal theorem provers as long as they support a (weak) default negation principle. The rationale is to capitalize on these well-investigated techniques and lift them to a more general language, viz., first-order logic, and strengthen the model-building capabilities of such systems. Among the systems that are suitable are dlv [CEF+97], smodels [NS96] and KRHyper [Wer03]. In our experiments we have chosen KRHyper, simply because we know it best.

Our transformation is applicable to any first-order logic KB, but it is geared towards application to first-order logic ontologies. It differs from the textbook transformation to clause logic in several ways:

1. It transforms away equality, so that model generation systems can be used, even though they usually do not include built-in equality handling.
2. Optionally, it respects a certain form of the Unique Name Assumption (UNA). This is useful in the context of ontologies, when different constants are best considered to denote different objects.
3. It allows to avoid unnecessary Skolem terms, if an existentially quantified role is already filled. This keeps the resulting models slim and meaningful.
4. It allows for a non-standard reading of existentially quantified formulas, namely as *integrity constraints*. That is, the model building process can be instructed to fail if an existentially quantified formula is not already fulfilled by the KB.
5. Finally, it allows for a "loop check", by which infinite models can be avoided in some cases by detecting finite ones.

1.3 Related Work

From a methodological point of view, we were helped to achieve our results by considering insights and combining results and techniques from automated theorem proving,

[1] http://www.w3.org/2003/08/owl-systems/test-results-out

description logic and logic programming. For instance, we employ default negation, as available in logic programming systems, as a tool to realize the mentioned "loop check", which is modeled after the "blocking" technique commonly found in Description Logic systems.

Because of the use of default negation, we cannot use a theorem prover for classical (first-order) logic. Since the ontology may contain "disjunctive" formulas like $\forall x\ (\mathsf{man}(x) \vee \mathsf{woman}(x) \leftarrow \mathsf{person}(x))$, Horn-logic is not expressive enough and we need a system that accepts *disjunctive* programs. Thus, we cannot use, e.g., the widely available Prolog-like logic programming systems based on SLDNF-resolution (see e.g. [Llo87]), which support default negation but do no not support disjunctive programs.

An approach closely related to ours in methodology is the translation approach in [GHVD03]. It allows to translate certain DL fragments to a certain class of logic programs. However, this approach is restricted to definite programs, i.e. it cannot treat disjunctions as in the example above. This limitation could easily be overcome by translating to positive disjunctive logic programs (DLPs) instead. Yet, the method has a more severe limitation, which essentially forbids existential quantification to introduce new individuals. For example, consider the expression "every person has a father", expressed as a DL axiom

$$\mathsf{person} \sqsubseteq \exists \mathsf{father}\,.\,\top$$

or as a first-order logic formula

$$\forall x \exists y\ (\mathsf{father}(x,y) \leftarrow \mathsf{person}(x))\ .$$

Such formulas cannot be treated by the method in [GHVD03] and thus are not part of their input language. The technical difficulty with formulas of this kind is that they introduce Skolem terms (e.g. $\mathsf{f}(x)$, intended to denote the "father" of an object x), which in general lead to non-termination of model computation systems. From that point of view, the purpose of our approach is to address this very problem: to achieve termination even in presence of existential quantifiers.

Our approach is somewhat related to model construction by hyper resolution e.g. [FL93, GHS02, GHS03]. One difference is our use of default negation, which is not available in hyper resolution systems. The perhaps closest related work is the translation scheme in [BB04]. However, that work is concerned with one specific ontology, FrameNet, and it is shown how to translate it to a logic program. The approach in this paper is thus much more general.

The rest of this paper is structured as follows. Section 2 contains preliminaries. Section 3 is the main part, it contains the transformations. In Section 4 we turn to the treatment of equality. In Section 5 we report on first experiments carried out on the SUMO ontology. Finally, in Section 6 we draw some conclusions.

2 Preliminaries

We use standard terminology from first-order logic and automated reasoning (see e.g. [Llo87]). Our formulas, and specifically clauses, are built over a signature Σ, usually

left implicit in the following. We assume that Σ contains a distinguished nullary predicate symbol false and a 2-ary predicate symbol $\approx$, equality, used infix. We deviate from the standard definitions by distinguishing between constants and nullary function symbols. This allows us to take the Unique Name Assumption (UNA) into account: constants are subject to the UNA, i.e. no model shall assign true to $c_1 \approx c_2$ for any two different constants c_1 and c_2. Nullary function symbols, by contrast, are not affected by the UNA, so that our definitions are compatible with the standard semantics.

A *(program) rule* is an expression of the form $H_1 \vee \cdots \vee H_m \leftarrow B_1, \ldots, B_k, \mathsf{not}\ B_{k+1}, \ldots, \mathsf{not}\ B_n$, where $m \geq 1$, $n \geq k \geq 0$ and H_i, for $i = 1, \ldots m$, and B_j, for $j = 1, \ldots, n$ are (possibly non-ground) atoms (over Σ). Each H_i is called a *head literal*, and each B_j is called a *body literal*. The *negative* body literals are those that include the default negation operator not, the other body literals are the *positive* ones. We write $H \vee \mathcal{H} \leftarrow B, \mathcal{B}, \mathsf{not}\ B', \mathcal{B}_{\mathsf{not}}$ to mean a program rule containing the head literal H, the positive body literal B and the negative body literal not B'. In a *positive* rule it holds $k = n$. We treat the terms "positive rule" and "clause" as synonyms.

A *disjunctive logic program (DLP)*, also just *program*, is a finite set of rules. A *positive DLP* consists of positive rules only; it is thus the same as a *clause set*. In a *normal* program each rule has exactly one head literal. We consider only *domain restricted* programs, where every variable occurring in a rule must also occur in some positive body atom $B_1, \ldots, B_k$. This is a common assumption and is present in systems like KRHyper [Wer03] and smodels [NS96]. As an example, consider the following (propositional) program:

$$\mathsf{whiskey} \vee \mathsf{water} \leftarrow \mathsf{thirsty}, \mathsf{not\ hungry} \quad (1)$$

$$\mathsf{water} \leftarrow \mathsf{whiskey} \quad (2)$$

$$\mathsf{thirsty} \leftarrow \quad (3)$$

Program rules can be read operationally in a top-down or in a bottom-up fashion. The top-down paradigm (of normal programs) became popular with Prolog and its underlying SLDNF Resolution (see [Llo87]). The bottom-up paradigm became popular with the observation that it often better realizes the idea of purely *declarative* programming. The purely declarative nature renders these approaches suitable in particular for knowledge representation applications, which is our interest here.

A bottom-up evaluation of the above sample program assigns *true* to thirsty, because the (empty) body of rule (3) is (trivially) satisfied, and so its head thirsty must be satisfied. But then, as hungry is *false* (by default), the body of rule (1) is satisfied, and so must be its head. For that, there is a choice of satisfying whiskey or water (or both). Notice, in the first case rule (2) becomes applicable and water must become *true*, too. In sum, we have the two models, $\{\mathsf{thirsty}, \mathsf{whiskey}, \mathsf{water}\}$ and $\{\mathsf{thirsty}, \mathsf{water}\}$. Indeed, the literature discusses various alternatives to assign semantics to DLPs. For instance, the *stable model semantics* would reject the first model, because it is not a minimal one. The *possible model semantics* admits both. None of them admits the classical model that assigns *true* to hungry and thirsty but nothing else (the intuition is that there is no rule to justify the truth of hungry). Either semantics is usable in our case.

Furthermore, the programs constructed below will be *stratified*,[2] which guarantees that they will have a stable/possible model if and only if the original ontology has a classical first-order model (which is its intended semantics.) Without going into details, we only note that the KRHyper system [Wer03], which we used for our experiments, computes possible models of domain-restricted stratified DLPs, and thus is suitable in the sense just mentioned. We further note that the above notions concerning semantics of logic programs lift to first-order logic by letting a rule stand for the set of all its ground instances, i.e. by the set of variable-free rules obtainable by replacing each variable in the rule by some variable-free term, in all possible ways. A good overview on DLPs can be found in [Nie99] (although on normal programs only). A more comprehensive textbook is [Bar03].

For space reasons, we omit here various technical details. The interested reader is referred to the long version of this paper, which can be obtained from `http://rsise.anu.edu.au/~baumgart/publications/`.

3 Translating First-Order Formulae to DLPs

We assume as given some ontology, e.g. an OWL ontology. The ontology may contain facts as well as non-taxonomic axioms and it could contain "rules" (cf. the introduction). We assume it to be written as a sentence in first-order logic. This section describes how to transform the first-order sentence to a DLP. The first steps of the transformation are concerned with flattening the possibly deeply structured sentence towards the flat form of DLP rules. An important, non-standard aspect hereby is to isolate and name subformulas containing existentially quantified variables. Once we described how to isolate these subformulas, Section 3.1 proposes four different ways of translating them to a DLP.

We first fix some notation. If $\boldsymbol{x}$ is a sequence of variables $x_1,\ldots,x_k$, for some $k \geq 0$, then $\forall \boldsymbol{x}$ denotes the sequence $\forall x_1 \cdots \forall x_k$. The expression $\exists \boldsymbol{x}$ is defined analogously, and $\mathbf{Q}\boldsymbol{x}$ stands for any sequence $Q_1 x_1 \cdots Q_k x_k$, where $Q_i \in \{\forall, \exists\}$, for all $i = 1,\ldots,k$, $k \geq 0$. When ψ is a formula, the notation $\psi(\boldsymbol{x})$ means that ψ contains no more free variables than those in the sequence of variables $\boldsymbol{x}$. We assume, without loss of generality, that the first-order logic sentence ϕ is given in prenex negation normal form. Thus, it is of the form $\phi = \mathbf{Q}z\ \psi(z)$, where $\mathbf{Q}z$ is the quantifier prefix and $\psi(z)$ is a quantifier-free formula, built with logical operators $\wedge$, $\vee$ and $\neg$, where $\neg$ occurs only in front of atoms.

We define our transformation $\tau(\phi)$ as follows. The quantifier prefix $\mathbf{Q}z$ may contain an existential quantifier, or not. If it does not, set $\tau(\phi) = \{\phi\}$. Otherwise ϕ can be written as

$$\phi = \mathbf{Q}z\ \psi(z) = \forall \boldsymbol{x} \exists \boldsymbol{y} \mathbf{Q}'z'\ (\Delta(\boldsymbol{x}) \vee \psi'(\boldsymbol{x}\boldsymbol{y}z'))\ , \tag{1}$$

where $\mathbf{Q}'$ is either empty or starts with a universal quantifier. The intention is to separate Ψ into two parts, the part Δ containing universally quantified variables only, and the remainder Ψ' containing at least one existentially quantified variable. We may assume

[2] Stratification means that the call-graph of a program does not contain circles containing negative body atoms.

that $\psi'(\boldsymbol{xyz}')$ is not a disjunction such that one of its immediate subformulas contains at most the variables $\boldsymbol{x}$, because then this subformula could be part of Δ. Notice that by replacing $\psi(z)$ in ϕ by $\mathsf{false} \vee \psi(z)$, the form (1) is indeed a general form ($\Delta(\boldsymbol{x})$ could be the atom false).

Suppose ϕ is of the form (1) and consider the following sentences derived from ϕ:

$$\phi_1 = \forall \boldsymbol{x} \exists \boldsymbol{y} (\Delta(\boldsymbol{x}) \vee def_{\psi'}(\boldsymbol{x},\boldsymbol{y}))$$
$$\phi_2 = \forall \boldsymbol{x} \forall \boldsymbol{y} \mathbf{Q}'\boldsymbol{z}' (\neg def_{\psi'}(\boldsymbol{x},\boldsymbol{y}) \vee \psi'(\boldsymbol{xyz}'))$$
$$\phi_3 = \forall \boldsymbol{x} \forall \boldsymbol{y} \overline{\mathbf{Q}'}\boldsymbol{z}' (\mathrm{NNF}(sat_{\psi'}(\boldsymbol{x},\boldsymbol{y}) \vee \neg \psi'(\boldsymbol{xyz}'))) \ ,$$

where $def_{\psi'}$ and $sat_{\psi'}$ are fresh predicate symbols of appropriate arity. The intention is to introduce in ϕ_2 a name $def_{\psi'}$ for the subformula Ψ', which allows to replace Ψ' in ϕ by $def_{\psi'}$. Regarding the formula ϕ_3, $\overline{\mathbf{Q}'}\boldsymbol{z}'$ denotes the quantifier prefix obtained from $\mathbf{Q}'\boldsymbol{z}'$ by replacing every universal quantifier by an existential one and vice versa, and NNF converts its argument to negation normal form. The formula ϕ_3 will play a role only later, in Section 3.1. Roughly, the purpose of the new name $sat_{\psi'}$ is to identify situations when Ψ' holds true. One can prove that these transformations preserve satisfiability. More precisely, ϕ is satisfiable if and only if $\phi_1 \wedge \phi_2 \wedge \phi_3$ is satisfiable.

For illustration, consider the formula

$$\forall x (\mathsf{p}(x) \rightarrow \exists y \, \mathsf{q}(x,y) \vee \mathsf{r}(x)) \ . \tag{1}$$

We rewrite it as

$$\phi = \forall x \exists y (\neg \mathsf{p}(x) \vee \mathsf{r}(x) \vee \mathsf{q}(x,y))$$

so that it is of the form (1) with $\Delta(x) = \neg \mathsf{p}(x) \vee \mathsf{r}(x)$ and $\psi'(x,y) = \mathsf{q}(x,y)$. We derive the following sentences:

$$\phi_1 = \forall x \exists y (\neg \mathsf{p}(x) \vee \mathsf{r}(x) \vee \mathsf{def}_{\psi}(x,y))$$
$$\phi_2 = \forall x \forall y (\neg \mathsf{def}_{\psi}(x,y) \vee \mathsf{q}(x,y))$$
$$\phi_3 = \forall x \forall y (\mathsf{sat}_{\psi}(x,y) \vee \neg \mathsf{q}(x,y))$$

It is not too difficult to see that already ϕ_1 and ϕ_2 together are equisatisfiable with ϕ. Regarding ϕ_3, suppose that, say, $\mathsf{q}(\mathsf{a},\mathsf{b})$ holds true in some interpretation. By ϕ_3, $\mathsf{sat}_{\Psi}(\mathsf{a},\mathsf{b})$ must be true as well, which can be exploited to conclude that the formula $\exists y \, \mathsf{q}(\mathsf{a},y)$ holds true. (As said, ϕ_3 can be ignored for now, but it will be crucial for the improvement in Section 3.1 below.)

Recall that Δ is a part of ϕ that contains universally quantified variables only. Now, Δ can be written as[3]

$$\Delta = \neg B_1(\boldsymbol{x}) \vee \cdots \vee \neg B_m(\boldsymbol{x}) \vee \Delta'(\boldsymbol{x}),$$

for some formula Δ', negative literals $\neg B_i$, for all $i = 1, \ldots, m$, $m \geq 0$, where m is chosen as large as possible. Notice we allow $m = 0$. Hence Δ can indeed be written this way.

[3] Similarly to above, we allow $\Delta'(\boldsymbol{x})$ to be false.

We write Δ this way with the intention to turn it into a flat formula, basically an implication between atoms. While its literals $\neg B_1(\boldsymbol{x}), \ldots, \neg B_m(\boldsymbol{x})$ pose no problems, its subformula $\Delta'(\boldsymbol{x})$ need not be a disjunction of atoms. To overcome this problem, we introduce a fresh name for $\Delta'(\boldsymbol{x})$. More precisely, from Δ derive the formulas

$$\Delta_1 = \neg B_1(\boldsymbol{x}) \vee \cdots \vee \neg B_m(\boldsymbol{x}) \vee def_{\Delta'}(\boldsymbol{x})$$
$$\Delta_2 = \forall \boldsymbol{x}(\neg def_{\Delta'}(\boldsymbol{x}) \vee \Delta'(\boldsymbol{x})) \ ,$$

where again $def_{\Delta'}$ is a fresh predicate symbol of appropriate arity. In the example, this yields:

$$\Delta_1 = \neg \mathsf{p}(x) \vee \mathsf{def}_\Delta(x)$$
$$\Delta_2 = \forall x \ (\neg \mathsf{def}_\Delta(x) \vee \mathsf{r}(x))$$

The next step is to replace Δ in ϕ_1 by Δ_1, which yields

$$\phi_1^{\Delta_1} = \forall \boldsymbol{x} \exists \boldsymbol{y}(\neg B_1(\boldsymbol{x}) \vee \cdots \vee \neg B_m(\boldsymbol{x}) \vee def_{\Delta'}(\boldsymbol{x}) \vee def_{\psi'}(\boldsymbol{x},\boldsymbol{y})) \ .$$

In our example,

$$\phi_1^{\Delta_1} = \forall x \exists y \ (\neg \mathsf{p}(x) \vee \mathsf{def}_\Delta(x) \vee \mathsf{def}_\psi(x,y)) \ .$$

Above we already defined $\tau(\phi) = \{\phi\}$ for the case that $\mathbf{Q}z$ does not contain an existential quantifier. We are now ready to define $\tau(\phi)$ if $\mathbf{Q}z$ does contain an existential quantifier:

$$\tau(\phi) = \{\phi_1^{\Delta_1}, \Delta_2\} \cup \tau(\phi_2) \cup \tau(\phi_3) \ .$$

In our example, this boils down to

$$\begin{aligned}\tau(\phi) = \{&\forall x \exists y \ (\neg \mathsf{p}(x) \vee \mathsf{def}_\Delta(x) \vee \mathsf{def}_\psi(x,y)),\\ &\forall x \ (\neg \mathsf{def}_\Delta(x) \vee \mathsf{r}(x)),\\ &\forall x \forall y \ (\neg \mathsf{def}_\psi(x,y) \vee \mathsf{q}(x,y)),\\ &\forall x \forall y \ (\mathsf{sat}_\psi(x,y) \vee \neg \mathsf{q}(x,y))\} \ .\end{aligned}$$

It might be instructive to compare this result, in particular the first formula, to the formula (1) we started with.

To see the termination of the transformation τ, observe that both ϕ_2 and ϕ_3 are strictly smaller than ϕ in the (well-founded) ordering on formulas with quantifier prefixes of same length induced by the lexicographic ordering on quantifier sequences, where $\exists$ is greater than $\forall$.

Introducing names (like $def_{\psi'}(\boldsymbol{x},\boldsymbol{y})$ above) for subformulas and adding definitions for them, like our transformation does, is a standard technique used in clause normal form transformations. It is well-know that such transformations preserve satisfiability[4].

[4] Because existential quantifiers are not eliminated, τ even preserves models, in both ways (in the sense of conservative extensions for the newly introduced symbols).

Notice that *all* sentences in $\tau(\phi)$ containing an existential quantifier are of the (simple) syntactic form as obtained in $\phi_1^{\Delta_1}$. These are "almost" rules, except for the circumstance that the variables $\boldsymbol{y}$ are existentially quantified (in a rule all variables are implicitly universally quantified). All other sentences in $\tau(\phi)$ are of the form $\forall \boldsymbol{x}\, \Delta(\boldsymbol{x})$ and can be converted to clausal form (i.e. a positive DLP) easily by means of well-known techniques.

3.1 Treating Existentially Quantified Subformulas

At this point, we assume that all universally quantified formulas in $\tau(\phi)$ have been transformed to clausal form. The remaining formulas contain existential quantifiers, which are all of the form as denoted by $\phi_1^{\Delta_1}$ above. Let Φ be a formula of this kind. We propose four different options to translate Φ to a DLP, each designed for a specific purpose: The *Skolemization Option* translates Φ by the use of Skolem terms, resulting in a traditional Skolemized DLP. The *Recycling Option* allows to introduce Skolem terms only if they are necessary, resulting in slimmer and more meaningful models. The *Model Checking Option* treats the existential quantification as an *integrity constraint*. With this option, the model building process is instructed to fail if there is no role filler in the model for the existential role. Last, the *Loop Check Option* allows to re-use existing Skolem terms in such a way that preference is given to a finite model.

Skolemization Option. With this option, a Skolem term is chosen as a default value to satisfy – in Description Logic terminology – an existentially quantified role. Technically, the formula Φ is translated to the following (domain-restricted) DLP:

$$def_{\Delta'}(\boldsymbol{x}) \vee def_{\psi'}(\boldsymbol{x}, \boldsymbol{sk}_{\Phi}(\boldsymbol{x})) \leftarrow B_1(\boldsymbol{x}), \ldots, B_m(\boldsymbol{x}) \tag{2}$$

Here, $\boldsymbol{sk}_{\Phi}(\boldsymbol{x})$ is a list of Skolem terms made from the variables $\boldsymbol{x}$. Intuitively speaking, the premise of Φ implies that either the universally quantified part of Φ or the existentially quantified part of Φ must be satisfied. The existentially quantified part is given a Skolem filler for the existential variable. Thereby, our transformation includes the usual Skolemization as its simplest option.

Recycling Option. This option allows to avoid the introduction of a Skolem term if there is already a role filler present in the model. This can be achieved by translating Φ as follows:

$$def_{\Delta'}(\boldsymbol{x}) \vee \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}) \vee def_{\psi'}(\boldsymbol{x}, \boldsymbol{sk}_{\Phi}(\boldsymbol{x})) \leftarrow B_1(\boldsymbol{x}), \ldots, B_m(\boldsymbol{x}) \tag{3}$$

$$\mathsf{false} \leftarrow def_{\Delta'}(\boldsymbol{x}), \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}) \tag{4}$$

$$\mathsf{false} \leftarrow \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}), def_{\psi'}(\boldsymbol{x}, \boldsymbol{y}) \tag{5}$$

$$\mathsf{false} \leftarrow \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}), \mathsf{not}\ sat1_{\psi'}(\boldsymbol{x}) \tag{6}$$

$$sat1_{\psi'}(\boldsymbol{x}) \leftarrow sat_{\psi'}(\boldsymbol{x}, \boldsymbol{y}) \tag{7}$$

$$\mathsf{false} \leftarrow def_{\psi'}(\boldsymbol{x}, \boldsymbol{y}), sat_{\psi'}(\boldsymbol{x}, \boldsymbol{z}), \mathsf{not}\ \mathsf{equal}_{|\boldsymbol{y}|}(\boldsymbol{y}, \boldsymbol{z}) \tag{8}$$

$$\mathsf{equal}_{|\boldsymbol{y}|}(\mathrm{x}_1, \ldots, \mathrm{x}_n, \mathrm{y}_1, \ldots, \mathrm{y}_n) \leftarrow \mathrm{x}_1 \approx \mathrm{y}_1, \ldots, \mathrm{x}_n \approx \mathrm{y}_n \tag{9}$$

Rule (3) contains one more head literal than (2), which is $\mathsf{check\_}sat_{\psi'}(\boldsymbol{x})$. This literal signals that there *exists* already a role-filler in the model for the existentially quantified role. The other rules realize certain exclusivity tests among the alternatives.
For illustration, consider again the formula

$$\phi = \forall x(\mathsf{p}(x) \rightarrow \exists y\, \mathsf{q}(x,y) \vee \mathsf{r}(x)) \quad .$$

The translation τ with the recycling option applied to ϕ yields for the rule scheme (3) the DLP

$$\mathsf{def}_r(\mathrm{x}) \vee \mathsf{check\_sat}_q(\mathrm{x}) \vee \mathsf{def}_q(\mathrm{x}, \mathsf{sk}(\mathrm{x})) \leftarrow \mathsf{p}(\mathrm{x}) \quad .$$

Suppose additionally the fact $\mathsf{p}(\mathsf{a})$ as given. Then, the model must satisfy the formula $\exists y\, \mathsf{q}(\mathsf{a},y) \vee \mathsf{r}(\mathsf{a})$. This can be achieved in three different ways:

1. The atom $\mathsf{r}(\mathsf{a})$ is added to the model, i.e. the part of ϕ that is outside the scope of the $\exists$-quantifier is assigned true. This is achieved by the first alternative in rule (3), which, together with the rule $\mathsf{r}(\mathrm{x}) \leftarrow \mathsf{def}_r(\mathrm{x})$ in $\tau(\phi)$, derives $\mathsf{r}(\mathsf{a})$.
2. The model already contains an atom $\mathsf{q}(\mathsf{a},\mathsf{t})$, for some term t. This is tested by the alternative $\mathsf{check\_sat}_q(\mathrm{x})$. If the model does not already contain some such atom $\mathsf{q}(\mathsf{a},\mathsf{t})$, *false* is derived and the third alternative is chosen.
3. The atom $\mathsf{q}(\mathsf{a},\mathsf{sk}(\mathsf{a}))$ with the Skolem term $\mathsf{sk}(\mathsf{a})$ is added to the model. This is achieved by the third alternative in rule (3), which, together with the rule $\mathsf{q}(\mathrm{x},\mathrm{y}) \leftarrow \mathsf{def}_q(\mathrm{x},\mathrm{y})$ in $\tau(\phi)$, derives $\mathsf{q}(\mathsf{a},\mathsf{sk}(\mathsf{a}))$. In this case, rule (8) makes sure that no other filler will or has been inserted that is equal to $\mathsf{sk}(\mathsf{a})$. The test for (non-)equality is necessary, because later, $\mathsf{sk}(\mathsf{a})$ could be equated to some other term. For instance, if $\mathsf{q}(\mathsf{a},\mathsf{b})$ is also present and $\mathsf{sk}(\mathsf{a}) \approx \mathsf{b}$ is not present, this model candidate will be rejected and the alternative $\mathsf{check\_sat}_q(\mathsf{a})$ will be chosen.

The formula $\exists y\, \mathsf{q}(\mathsf{a},y)$ will thus be satisfied in one way or the other, with a preference to a filler different from the Skolem term[5].

Model Checking Option. Sometimes, it is useful to regard existential formulae as integrity constraints for a KB – for instance, to check if the objects mentioned in a given database suffice to extend it to a model for a given KB. Instead of creating fillers by means of Skolem terms, the model construction process must check that fillers are already present. This can be achieved by translating Φ according to the "recycling option", where (3) is replaced by the following scheme:

$$def_{\Delta'}(\boldsymbol{x}) \vee \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}) \leftarrow B_1(\boldsymbol{x}), \ldots, B_m(\boldsymbol{x}) \tag{10}$$

This transformation ensures that no Skolem terms can be inserted by the model computation. The only way to satisfy the existentially quantified part then is by proving that it is already satisfied.

[5] Assuming that the model generation system processes the alternatives in the order given by (3)

Loop Check Option. The introduction of Skolem terms leads easily to nontermination of model-generation systems. Instead of creating new Skolem terms, we propose to "reuse" existing Skolem terms, if they qualify as role fillers – similarly to the blocking techniques found in description logic systems (although more general). This can be achieved by translating Φ according to the "recycling option", where the rule (3) is replaced by the following rules:

$$def_{\Delta'}(\boldsymbol{x}) \vee \mathsf{check\_}sat_{\psi'}(\boldsymbol{x}) \vee choose\_default\_filler_{\Phi}(\boldsymbol{x}) \vee def_{\psi'}(\boldsymbol{x}, \boldsymbol{sk}_{\Phi}(\boldsymbol{x})) \leftarrow B_1(\boldsymbol{x}), \ldots, B_m(\boldsymbol{x}) \tag{11}$$

$$other\_filler_{\Phi}(\boldsymbol{x}, \boldsymbol{sk}_{\Phi}(\boldsymbol{y})) \vee def_{\psi'}(\boldsymbol{x}, \boldsymbol{sk}_{\Phi}(\boldsymbol{y})) \leftarrow choose\_default\_filler_{\Phi}(\boldsymbol{x}), sat_{\psi'}(\boldsymbol{x_1}, \boldsymbol{sk}_{\Phi}(\boldsymbol{y})) \tag{12}$$

$$\mathsf{false} \leftarrow def_{\psi'}(\boldsymbol{x}, \boldsymbol{y}), other\_filler_{\Phi}(\boldsymbol{x}, z) \tag{13}$$

$$\mathsf{false} \leftarrow def_{\psi'}(\boldsymbol{x}, \boldsymbol{y}), def_{\psi'}(\boldsymbol{x}, z), \mathsf{not\ equal}_{|\boldsymbol{y}|}(\boldsymbol{y}, z) \tag{14}$$

$$\mathsf{false} \leftarrow choose\_default\_filler_{\Phi}(\boldsymbol{x}), \mathsf{not}\ some\_default\_filler_{\Phi}(\boldsymbol{x}) \tag{15}$$

$$some\_default\_filler_{\Phi}(\boldsymbol{x}) \leftarrow def_{\psi'}(\boldsymbol{x}, \boldsymbol{y}) \tag{16}$$

Compared to rule (3), rule (11) contains again an additional head literal, which is $choose\_default\_filler_{\Phi}(\boldsymbol{x})$. Together with rule (12) this has the effect of nondeterministically selecting a default filler among all Skolem terms previously introduced to satisfy the existential quantification of (another instance of) the formula. The nondeterministic selection process is realized by the $other\_filler_{\Phi}$-alternative in the head, which allows to choose a default filler – or not. The remaining rules achieve that exactly one default filler will be generated.

For illustration, consider the following example from the Tambis Ontology [SPB+04]:

$$\forall x\ (\mathsf{chapter}(x) \rightarrow \exists y\ (\mathsf{in\_book}(x, y) \wedge \mathsf{book}(y))) \tag{17}$$

$$\forall x\ (\mathsf{book}(x) \rightarrow \exists y\ (\mathsf{has\_chapter}(x, y) \wedge \mathsf{chapter}(y))) \tag{18}$$

$$\forall x\ \neg(\mathsf{book}(x) \wedge \mathsf{chapter}(x)) \tag{19}$$

Notice the terminological cycle. To get the model computation started, suppose an additional fact $\mathsf{chapter}(a)$. Leaving away many uninteresting facts, the model generation process will first satisfy (17) by deriving

$$\mathsf{book}(f_1(a)) \tag{20}$$

$$\mathsf{in\_book}(a, f_1(a))\ . \tag{21}$$

Next, it will satisfy (18) by deriving

$$\mathsf{chapter}(f_2(f_1(a))) \tag{22}$$

$$\mathsf{has\_chapter}(f_1(a), f_2(f_1(a)))\ . \tag{23}$$

Now, (17) requires the existence of a book for the newly created chapter $f_2(f_1(a))$. Instead of creating a new Skolem term, rule (12) will find that $f_1(a)$ can be used as a default filler. Thus, the model generation process terminates by deriving

$$\mathsf{in\_book}(f_2(f_1(a)), f_1(a))\ . \tag{24}$$

In summary, the natural infinite model will be avoided by the loop check option. Thus, the loop check option can allow for a finite model in cases where a naive translation to clauses may only have an infinite model.

3.2 The Loop Check Option in Practice

Up to now, the loop check option has been introduced in a purely declarative way. More considerations are necessary to make it effective in practice. First of all, the loop check is not designed to prove the *unsatisfiability* of a set of formulae. Unsatisfiability can be proven more easily without the loop check option, because the search space is much smaller without the additional rules. Instead, the loop check aims at the more difficult problem of proving the *satisfiability* of a set of formulae.

If the loop check-transformation of a set of formulae has a finite model, then the original set of formulae also has a finite model. Unfortunately, model generation systems may have difficulties in finding this finite model, even if it exists.

The issue is to realize a *fair* search for a model. This is not trivial, as, in general, the Herband universe of the programs obtained by the translation is infinite. For instance, the search strategy of KRHyper is fair in the sense that it guarantees *refutational* completeness (in particular when the Herbrand universe is infinite). In contrast, even for very simple satisfiable programs obtained with the loop check option of Section 3.1, KRHyper will not terminate – the search strategy is just not fair for (finite) model building. The iterative deepening scheme KRHyper uses may lead into an infinite branch in the search tree and may thereby miss an alternative branch leading to a model. Other systems, like smodels, require full grounding-out of their input clause set, which is obviously, in general, not possible in presence of function symbols.

A solution to these problems is to generate (finite) interpretations as candidates, check them explicitly for being a model of the program and stop this search as soon as a model has been found. A systematic way to do so is to run the systems with a bound on the resources allowed, checking if a model has been found and increasing these resources in a fair way on failure (iterative deepening). For KRHyper, for instance, this can be achieved by running it with a limit on the term depth on the generated terms. Regarding smodels, one could work with growing approximations of the infinite set of all ground instances.

For the check for modelship the following rules are added to the loop check translation of a formula Φ:

$$unsatisfied_{\Phi}(\boldsymbol{x}) \leftarrow B_1(\boldsymbol{x}), \ldots, B_m(\boldsymbol{x}), \mathsf{not}\ sat1_{\Psi'}(\boldsymbol{x}) \tag{25}$$

$$\mathsf{unsatisfied\_some} \leftarrow unsatisfied_{\Phi}(\boldsymbol{x}) \tag{26}$$

Last, one adds the rule

$$\mathsf{satisfiable} \leftarrow \mathsf{not\ unsatisfied\_some}\ . \tag{27}$$

Now, the idea is to conclude if a model contains the atom satisfiable then the set of formula is indeed satisfiable (in a finite model) and no further deepening is necessary.

However, this conclusion is not true if the given formula, and hence the obtained translated program, contains function symbols other than Skolem functions and constants. The test not $sat1_{\Psi'}(\boldsymbol{x})$ in the body of the first clause is too weak then.

In practice, the situation is perhaps not as bad as it might seem. Many interesting ontologies can be formulated without function symbols at all (as is witnessed already by the existence of numerous interesting DL ontologies, which do not contain function symbols). We conjecture that our transformation will find a finite model whenever one exists, provided function symbols as mentioned are not present. For future work we intend to improve the transformation to cope better with function symbols.

4 Equality

Ontologies typically make use of equality. For example, equality is used in function definitions or in *integrity constraints* to state that certain objects are different. Another common use of equality is to state that two objects must be equal under certain circumstances. For example, the "age" of twins must be "equal".

At this point of the paper, we may assume that the ontology has been converted to a DLP. As a running example, consider the following DLP, which contains one equation:[6]

$$\mathsf{p}(\mathsf{c}, \mathsf{h}()) \leftarrow \tag{28}$$

$$\mathsf{x} \approx \mathsf{f}(\mathsf{g}(\mathsf{d})) \leftarrow \mathsf{p}(\mathsf{x}, \mathsf{h}()) \tag{29}$$

The model of this DLP will contain the fact $\mathsf{p}(\mathsf{c}, \mathsf{h}())$. Rule (29) will derive $\mathsf{c} \approx \mathsf{f}(\mathsf{g}(\mathsf{d}))$. However, equational consequences like $\mathsf{f}(\mathsf{g}(\mathsf{d})) \approx \mathsf{c}$ (by symmetry of $\approx$) are not derived. Hence, the $\approx$-predicate requires special treatment. The most advanced techniques to *efficiently* treat equality have been developed in the field of *automated theorem proving* for refutational theorem provers (see [BG98]). Unfortunately, none of these techniques has been implemented in the model computation systems we target at.

One generic option to treat equality is by means of adding the equality axioms. However, the search space induced by the resulting clause set is prohibitively high and achieving termination is practically impossible. The most problematic axioms in this regard are substitution axioms, like $\mathsf{f}(\mathrm{x}) \approx \mathsf{f}(\mathrm{y}) \leftarrow \mathrm{x} \approx \mathrm{y}$. As soon as the model contains one equation, say $\mathsf{a} \approx \mathsf{b}$, and one unary function symbol f, the substitution axioms generate infinitely many facts of the form $f(f(f(\mathsf{a}))) \approx f(f(f(\mathsf{b})))$. An alternative option is to "compile away" equality. The probably most well-known method in this direction is the "modification method" in [Bra75], which was later improved in [BGV97]. We follow this direction and propose an *equality transformation for DLPs*.

We say that a rule is *flat* if (1) the only proper subterms of terms in equations are either variables or constants, and (2) all arguments to predicate symbols are either variables or constants. Every rule can be turned into a flat one by recursively replacing an offending subterm t by a fresh variable x and adding the equation $t \approx x$ to the rule body(see again [BGV97]). For example, a flat version of the above DLP is

[6] Remember that we distinguish constants (like c, subject to the UNA) and nullary functions (like h(), not subject to the UNA).

$$\mathsf{p}(\mathsf{c}, v_1) \leftarrow v_1 \approx \mathsf{h}()$$
$$\mathsf{x} \approx \mathsf{f}(v_1) \leftarrow \mathsf{p}(\mathsf{x}, v_2), v_2 \approx \mathsf{h}(), v_1 \approx \mathsf{g}(\mathsf{d}) \ .$$

The purpose of flattening is to achieve the effect of the substitution axioms. To axiomatize the Unique Name Assumption (wrt. constants), one adds the rules $\mathsf{false} \leftarrow c \approx d$, for each pair c, d of different constants. Next, $\approx$ has to be confined to an equivalence relation by means of the rules[7]

$$\mathsf{x} \approx \mathsf{x} \leftarrow$$
$$\mathsf{x} \approx \mathsf{y} \leftarrow \mathsf{y} \approx \mathsf{x}$$
$$\mathsf{x} \approx \mathsf{z} \leftarrow \mathsf{x} \approx \mathsf{y}, \mathsf{y} \approx \mathsf{z} \ .$$

The addition of these axioms completes the equality transformation.

For the simple example above, any reasonable bottom-up model computation system will terminate on its equality transformation and report as the result $\{\mathsf{c} \approx \mathsf{f}(\mathsf{g}(\mathsf{d})), \mathsf{f}(\mathsf{g}(\mathsf{d})) \approx \mathsf{c}, \mathsf{p}(\mathsf{c}, \mathsf{h}()), \mathsf{x} \approx \mathsf{x}\}$, which describes the expected model of the original program. Note that any such system would not have terminated on the original program when equipped with the equality axioms. Our transformation is correct, i.e., the transformed clause set is satisfiable if and only if the given one is satisfiable wrt. interpretations where $\approx$ is interpreted as the equality relation. See the long version of this paper for a proof (`http://rsise.anu.edu.au/~baumgart/publications/`).

5 Preliminary Experiments

We applied our transformation to the core of the Suggested Upper Merged Ontology SUMO [NP01]. SUMO contains *meta-predicates*, i.e. predicates that define the properties of other predicates. We translated these predicates appropriately to first-order logic. For example, we translated the (higher-order) sentence $\mathsf{disjoint\_classes}(\mathsf{Man}, \mathsf{Woman})$ to the rule

$$\mathsf{false} \leftarrow \mathsf{instance}(x, \mathsf{Man}), \mathsf{instance}(x, \mathsf{Woman}) \ . \tag{30}$$

SUMO occasionally uses other higher order formulae, which we had to filter out. The resulting first-order KB contains about 1800 formulae.

Running KRHyper on the DLP transformation revealed numerous inconsistencies in SUMO. These included misspelled and hence unbound variables as well as semantic inconsistencies in connection with the Mid-level-ontology extensions. For example, one can derive that planetEarth is a geographicArea. Since each geographicArea is a geographicSubregion of planetEarth, it follows that planetEarth is a geographicSubregion of itself. This contradicts the irreflexivity of geographicSubregion. We reported the errors to the developers of SUMO and removed them. Then, KRHyper can calculate a model for our DLP translation within a few seconds. The model consists of roughly 2000 facts.

[7] Strictly speaking, the reflexivity rule $\mathsf{x} \approx \mathsf{x} \leftarrow$ is not domain-restricted. But this case is harmless and usually poses no problems.

To test the equality transformation, we added the following facts to SUMO: France lies west of Germany and Germany's biggest trading partner lies east of Germany.

$$\mathsf{orientation}(\mathsf{germany}, \mathsf{france}, \mathsf{west})$$
$$\mathsf{orientation}(\mathsf{germany}, \mathsf{biggestTradingPartner}(\mathsf{germany}), \mathsf{east}) \quad .$$

By help of the axioms in SUMO, KRHyper deduces (among others) the following facts:

$$\mathsf{orientation}(\mathsf{france}, \mathsf{germany}, \mathsf{east})$$
$$\mathsf{orientation}(\mathsf{biggestTradingPartner}(\mathsf{germany}), \mathsf{germany}, \mathsf{west})$$
$$\mathsf{between}(\mathsf{germany}, \mathsf{france}, \mathsf{biggestTradingPartner}(\mathsf{germany}))$$

Now, we add the fact $\mathsf{biggestTradingPartner}(\mathsf{germany}) \approx \mathsf{france}$. As a result, KRHyper derives a contradiction, as expected, because France cannot lie both east and west of Germany. To test our default value transformation, we added the following facts to SUMO:

$$\mathsf{instance}(\mathsf{p}, \mathsf{judicialProcess})$$
$$\mathsf{agent}(\mathsf{p}, \mathsf{a})$$

In SUMO, each judicial process is a political process. Furthermore, each political process requires an agent. Hence the model generation produces the fact $\mathsf{agent}(\mathsf{p}, \mathsf{f}(\mathsf{p}))$. However, if the recycling option is chosen, *a* qualifies as a default filler for the agent role. Consequently, the above fact is *not* derived with the recycling option.

SUMO contains numerous axioms that lead to infinite models. Unfortunately, in many cases they cannot be detected (finitely) by the current version of our loop check option. In these cases, the prover does not terminate.

6 Conclusions

We presented a transformation from first-order logic formulae to disjunctive logic programs. The programs resulting from the transformation can be fed into many existing logic programming model generation systems. As special features, our transformation allows the efficient treatment of equality, and it includes a certain form of the unique name assumption. Using Description Logic terminology, it allows a flexible handling of existentially quantified roles, including the avoidance of unnecessary Skolem terms, or the re-use of existing Skolem terms. By re-using existing Skolem terms, our transformation allows to generate finite models in certain cases, so that termination of the theorem prover can be achieved more often. (Of course, the general problem is undecidable, which puts natural limits on what can be achieved.)

Our main results are of a theoretical nature, namely soundness and completeness results. We carried out preliminary experiments with the SUMO ontology. Unfortunately our transformation did not prove strong enough to compute a finite model for the whole SUMO. The equality treatment and the flexible handling of existential roles, however, proved already applicable and useful, e.g. to subsets of SUMO. For future work, we intend to strengthen the transformation, so that finite models can be detected more often.

Acknowledgements. We wish to thank the reviewers for their helpful suggestions.

References

[Bar03] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[BB04] P. Baumgartner and A. Burchardt. Logic Programming Infrastructure for Inferences on FrameNet. In J. Alferes and J. Leite, eds., *JELIA'04*, LNAI 3229, Springer, 2004.

[BCM+02] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, eds. *Description Logic Handbook*. Cambridge University Press, 2002.

[BG98] L. Bachmair and H. Ganzinger. Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P. H. Schmitt, eds., *Automated Deduction. A Basis for Applications*, Volume I. Kluwer, 1998.

[BGV97] L. Bachmair, H. Ganzinger, and A. Voronkov. Elimination of Equality via Transformation with Ordering Constraints. In Proc. CADE 15, LNAI 1421, Springer 1998.

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

[CEF+97] S. Citrigno, Th. Eiter, W. Faber, G. Gottlob, Chr. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *Workshop Logische Programmierung*, 1997.

[FL93] C. Fermüller and A. Leitsch. Model Building by Resolution. In *Computer Science Logic: CSL'92*, *LNCS 702*, Springer, 1993.

[GHS02] L. Georgieva, U. Hustadt, and R. A. Schmidt. A new Clausal Class Decidable by Hyperresolution. In *CADE-18*, *LNAI* 2392. Springer, 2002.

[GHS03] L. Georgieva, U. Hustadt, and R. A. Schmidt. Hyperresolution for Guarded Formulae. *J. Symbolic Computat.*, 36(1–2):163–192, 2003.

[GHVD03] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *WWW 2003*, ACM, 2003.

[HB+04] I. Horrocks, H. Boley, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RulMl. http://www.w3.org/Submission/SWRL/, May 2004.

[KIF] Kif - knowledge interchange format. `http://www.csee.umbc.edu/kse/kif/`.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[Nie99] I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, 25(3-4):241–273, 1999.

[NP01] I. Niles and A. Pease. Towards a standard upper ontology. In C. Welty and B. Smith, eds., *In Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, 2001.

[NS96] I. Niemelä and P. Simons. Efficient Implementation of the Well-Founded and Stable Model Semantics. In *Proceedings of JICSLP*, The MIT Press, 1996.

[RV01] A. Riazonov and A. Voronkov. Vampire 1.1 (system description). In *Proc. IJCAR*, LNCS 2083. Springer, 2001.

[SPB+04] R. D. Stevens, N. W. Paton, S. K. Bechhofer, G. K. Ng, M. Peim, P. G. Baker, C. A. Goble, and A. M. Brass. Tambis: Transparent Access to Multiple Bioinformatics Services. *Genetics, Genomics, Proteomics, and Bioinformatics*, January 2004.

[Wer03] Christoph Wernhard. System Description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.