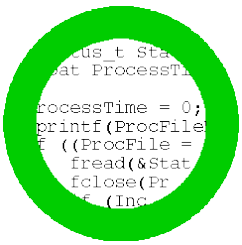
ESARLT

Proceedings of the CADE-21 Workshop on

Empirically Successful Automated Reasoning in Large Theories

Bremen, German, 15th July 2007

Editors: Geoff Sutcliffe, Josef Urban, Stephan Schulz



ESARLT

Empirically Successful Automated Reasoning in Large Theories

Geoff Sutcliffe
University of Miami

Josef Urban
Charles University in Prague

Stephan Schulz
Technische Universität München

The CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories (ESARLT) brings together practioners and researchers who are concerned with the development and application of automated reasoning in large theories. These are theories in which:

- There are many axioms
- There are many predicates and functors
- There are many theorems to be proved from the axioms
- There are many theorems that are provable from a subset of the axioms

In large theories it is useful to develop and apply intelligent reasoning techniques that go beyond "black box" reasoning, to include techniques for selecting axioms, using proved theorems as lemmas, etc. Reasoning in all forms (automated, interactive, etc) and all logics (classical, non-classical, all orders, etc) is of interest to the workshop. The workshop discusses only "really running" systems and applications, and not theoretical ideas that have not yet been translated into working software.

The workshop has two main topic areas:

Systems

- Implementation techniques and comparisons
- Data structures and formats for the representation of proof tasks and derivations, proof and lemma storage, etc.
- Algorithms and techniques for selecting and using axioms and lemmas
- Implemented and evaluated heuristics
- Implemented and evaluated user interfaces

Applications

- Descriptions of automated reasoning solutions in large theory domains
- Experiences with practical applications
- Encoding of domain problems into logic, and decoding of logic solutions into the domain
- User interfaces
- System integration

Additionally, the workshop includes **system and application demonstrations**.

Participants come from several sources:

- Researchers who have developed and implemented successful automated reasoning techniques and systems for large theories.
- Practioners who have deployed automated reasoning systems in large theories.
- Users who have already attempted to apply automated reasoning in their large theory domains, and now wish to learn more.
- Potential users who are interested in learning how automated reasoning may be used in their large theory domains.

The concrete application areas include the fields of verification, deductive databases, mathematics, knowledge representation, semantic web, etc. The workshop provides a forum for discussion of the techniques necessary to take automated reasoning from the lab and into the "real world" of large theories. The workshop enables the attendees to learn from each others' practical experiences, and documents their state-of-the-art techniques.

Developing Efficient SMT Solvers

Leonardo de Moura
One Microsoft Way
Redmond, WA
98052 USA

Abstract

Decision procedures for checking satisfiability of logical formulas are crucial for many verification applications. Of particular recent interest are solvers for Satisfiability Modulo Theories (SMT). SMT solvers decide logical satisfiability (or dually, validity) of formulas in classical multi-sorted first-order logic with equality, with respect to a background theory. The success of SMT for verification applications is largely due to the suitability of supported background theories for expressing verification conditions.

In this talk I will discuss how modern SMT solvers work, and the main implementation techniques used. I will also describe how SMT solvers are used in industry and Microsoft in particular.

Extensional Reasoning

Timothy L. Hinrichs and Michael R. Genesereth
Stanford University
{thinrich,genesereth}@cs.stanford.edu

Abstract

Relational databases have had great industrial success in computer science, their power evidenced by theoretical analysis and widespread adoption. Often, automated theorem provers do not take advantage of database query engines and therefore do not routinely leverage that source of power. Extensional Reasoning is a novel approach to automated theorem proving where the system automatically translates a logical entailment query into a query about a database system so that the answers to the two queries are the same. This paper discusses the framework for Extensional Reasoning, describes algorithms that enable a theorem prover to leverage the power of the database in the case of axiomatically complete theories, and discusses theory resolution for handling incomplete theories.

1 Introduction

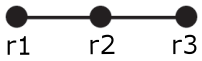
Relational database systems have been built to answer questions about enormous amounts of data, yet it is rare today for a theorem prover (Cyc's inference engine [MG03] a notable exception) to exploit those systems when confronted with large theories. Extensional Reasoning (ER) is an approach to automated theorem proving where the system transforms a logical entailment query into a database (a set of extensionally defined tables), a set of view definitions (a set of intensionally defined tables), and a database query.

For example, the map coloring problem is often stated as follows: given a map and a set of colors, paint each region of the map so that no two adjacent regions are the same color. One very natural way to encode this problem (according to the introductory logic students at Stanford who offer up this formulation year after year) centers around the following constraints.

$$color(x, y) \Rightarrow region(x) \wedge hue(y)$$
$$color(x, y) \wedge adj(x, z) \Rightarrow \neg color(z, y)$$

These constraints are the same regardless the map and the colors, i.e. regardless the definitions for *region*, *hue*, and *adj*; moreover, the definitions for these three predicates are often complete. For example, for the graph illustrated in Fig. 1, we would have the following definitions.

$$region(x) \Leftrightarrow (x = r_1 \vee x = r_2 \vee x = r_3)$$
$$hue(x) \Leftrightarrow (x = red \vee x = blue)$$
$$adj(x, y) \Leftrightarrow ((x = r_1 \wedge y = r_2) \vee (x = r_2 \wedge y = r_3))$$



Query: $next(x, y) \wedge next(y, z)$

Database:

next	
red	blue
blue	red

Figure 1: A 3-node graph and its corresponding database query

It so happens that the database formulation of this problem studied in, for example, [McC82] is quite different from the one above and is shown in Fig. 1. Given a table, *next*, that contains all the possible pairs of adjacent colors, the query for the three region graph would be

$$next(x, y) \wedge next(y, z)$$

where x represents the hue for r_1, y the hue for r_2, and z the hue for r_3.

The key difference between the formulations is that the database version entails all the valid colorings, whereas the classical version is only consistent with each valid coloring individually. In Extensional Reasoning, the transformation from one formulation into the other happens automatically.

A relational database corresponds to a special kind of logical theory—one that is axiomatically complete. Such theories are rare, and even when the theory is complete, recognizing that fact and constructing the appropriate database is nontrivial. Since the logical formulation of map coloring comprises an incomplete theory, to transform it into the database formulation we must first complete it. Because theory-completion in the context of Extensional Reasoning is performed solely for the sake of efficiency, a theory must be completed so that any entailment query about the original theory can be transformed into an entailment query about the completed theory where the answers to the queries are the same, another nontrivial problem.

Sometimes the work is worth the effort. A database can sometimes solve the database version of the query orders of magnitude faster than traditional techniques solve the classical version; moreover, the cost of a database query is bounded (polynomial in the size of the data and exponential in the size of the query), making the performance more predictable, sometimes an important feature from the users' perspective. These benefits appear to have several causes. First, the portion of the logical theory that is represented extensionally can be indexed very efficiently both for lookup and retrieval. Second, negation is treated with Negation as Failure, which can cause large reductions in the search space. Finally, if there is a solution to a problem, the candidates can be checked one at a time, which is in contrast to the theorem proving environment where disjunctive answers would require checking each subset of possible answers.

While we hope Extensional Reasoning will eventually be applied to a wide variety of logics, for the time being we have elected to focus on theories in a decidable logic, placing the issue of efficiency front and center. The particular logic we are studying is a fragment of first-order logic that is a perennial problem in the theorem proving community: it includes the domain closure axiom, which guarantees decidability while allowing arbitrary quantification. This logic, to which the example above belongs, allows

us to avoid issues of undecidability at this early stage in the development of Extensional Reasoning, while at the same time giving us the opportunity to make progress on an important class of problems.

Orthogonal to the choice of logic, Extensional Reasoning could be applied in a variety of settings that differ in the way efficiency is measured. Our work thus far measures efficiency the same way the theorem proving community does. Once the machine has been given a premise set and a query the clock starts; the clock stops once the machine returns an answer. We do not amortize reformulation costs over multiple queries, and we do not assume the premises or query have ever been seen before or will ever be seen again.

The bulk of this paper examines Extensional Reasoning in the case of complete theories and introduces algorithms for recognizing completeness and transforming a complete theory into a database system. For incomplete theories, some results on theory resolution are introduced. Further work on the incomplete case can be found in a companion paper [HG07]. Our technical contributions occur in the sections on Complete theories (3) and Incomplete theories (4). The necessary background is given in Sect. 2.

2 Background

In our investigations of Extensional Reasoning thus far, the logic we have considered is function-free first-order logic with equality, unique names axioms (UNA), and a domain closure axiom (DCA). The UNA state that every pair of distinct object constants in the vocabulary is unequal. The DCA states that every object in the universe must be one of the object constants in the vocabulary. Together, the UNA and DCA ensure that the only models that satisfy a given set of sentences are the Herbrand models of those sentences, and because the language is function-free, every such model has a finite universe. We call this logic Finite Herbrand Logic (FHL). It is noteworthy that entailment in FHL is decidable.

Besides the existence of UNA and a DCA, the definitions for FHL are the same as those for function-free first-order logic with equality. Terms and sentences are defined as usual. We say a sentence is *closed* whenever it has no free variables, and *open* whenever it has at least one free variable. The definition for a model is the usual one, but because all the models of FHL are isomorphic to finite Herbrand models, it is often convenient to treat a model as a set of ground atoms. When we do that, satisfaction is defined as follows.

Definition 1 (FHL Satisfaction). *The definition for the satisfaction of closed sentences, where the model M is represented as a set of ground atoms, is as follows.*

$\models_M s = t$ *if and only if* s *and* t *are syntactically identical.*

$\models_M p(t_1, \ldots, t_n)$ *if and only if* $p(t_1, \ldots, t_n) \in M$

$\models_M \neg\psi$ *if and only if* $\not\models_M \psi$

$\models_M \psi_1 \wedge \psi_2$ *if and only if* $\models_M \psi_1$ *and* $\models_M \psi_2$

$\models_M \forall x.\psi(x)$ *if and only if* $\models_M \psi(a)$ *for every object constant* a.

An open sentence $\psi(x_1, \ldots, x_n)$ with free variables $x_1, \ldots, x_n$ is satisfied by M if and only if $\forall x_1 \ldots x_n.\psi(x_1, \ldots, x_n)$ is satisfied by M according to the above definition.

A set of sentences in FHL constitutes an incomplete theory whenever there is more than one Herbrand model that satisfies it. A set of sentences in FHL constitutes a complete theory whenever there is at most one Herbrand model that satisfies it. Logical entailment is defined as usual: $\Delta \models \phi$ in FHL if and only if every model that satisfies Δ also satisfies ϕ.

The language used in this paper for representing database systems is nonrecursive datalog with negation, denoted *datalog*$^\neg$, which is equivalent in expressive power to relational algebra and SQL. This particular language has been chosen because it is well-understood and because some of the algorithms for processing it are very similar to algorithms for processing classical logic, which allows for relatively clean comparisons.

A *datalog*$^\neg$ system consists of (1) a set of tables, named using the Extensional Database predicates (EDB predicates) and (2) a set of datalog rules whose heads use the Intensional Database predicates (IDB predicates). The EDB predicates and IDB predicates are disjoint. A datalog rule is an implication, where $:-$ is used in place of $\Leftarrow$.

$$h :- b_1 \wedge \cdots \wedge b_n$$

h, the head of the rule, is an atomic sentence. Each b_i is a literal, and the conjunction of b_is is called the body of the rule. Every rule must be safe: every variable that occurs in the head or in a negative literal must occur in a positive literal in the body. Collectively, the rule set must be non-recursive, which can be defined in terms of the rules' dependency graph. The dependency graph for a rule set consists of one node per predicate and an edge from u to v exactly when there is a rule with u in the head and v in the body, labelled with $\neg$ if v occurs in a negative literal. To be nonrecursive, the dependency graph for a rule set must be acyclic.

A model for a *datalog*$^\neg$ system is the same as that for FHL: a set of ground atoms. We use the standard stratified semantics. A consequence of these definitions is that every *datalog*$^\neg$ system is satisfied by exactly one Herbrand model, or more precisely, a database system is a representation of a single model. A database query is true exactly when the query is true in that model, written $\Gamma \cup \Lambda \models \phi$, where Γ consists of the extensional tables and Λ the view definitions. When confusion may arise, we will subscript $\models$ with FHL and D to indicate the semantics for FHL and *datalog*$^\neg$, respectively.

Because every *datalog*$^\neg$ system represents a single model, the logical theory corresponding to a database system is one that is axiomatically complete.

Definition 2 (Axiomatic Completeness). *A satisfiable set of sentences Δ is axiomatically complete with respect to language L if and only if for every closed sentence s in L, either $\Delta \models s$ or $\Delta \models \neg s$. Δ is complete with respect to vocabulary V if and only if it is complete with respect to the set of all first-order sentences that can be formed from V.*

3 Complete Theories

A satisfiable set of sentences in Finite Herbrand Logic can always be transformed into $datalog^{\neg}$ while preserving logical equivalence if those sentences constitute a complete theory. In this section, we discuss algorithms for recognizing that a set of sentences is complete and algorithms for transforming such a sentence set into a $datalog^{\neg}$ system.

3.1 Recognizing Complete Theories

In Finite Herbrand Logic, a satisfiable, complete theory is satisfied by exactly one Herbrand model—by exactly one set of ground literals. Entailment in FHL is decidable because there is a fixed, finite bound on the size of the universe. Consequently, checking whether a satisfiable FHL theory is complete is decidable. For each ground atom a in the language (of which there are finitely many), check whether Δ entails a or Δ entails $\neg a$. If for some a, neither is entailed, the theory is incomplete; otherwise, the theory is complete.

This decision algorithm relies on entailment queries to determine whether the theory is complete, but entailment is the very problem Extensional Reasoning is meant to confront. For this reason we developed an alternate algorithm that has more of a syntactic flavor—one that performs some inexpensive tests that are sufficient for ensuring completeness.

Complete theories have been studied in the nonmonotonic reasoning literature. Predicate completion, for example, was one of the early techniques used to define the semantics of Negation as Failure in Logic Programming [Llo84]. When applied to a set of nonrecursive rules, predicate completion produces a set of nonrecursive biconditional sentences, e.g.

$$\begin{aligned} p(x) &\Leftrightarrow (q(x) \wedge r(x)) \\ q(x) &\Leftrightarrow (x = a \vee x = b) \\ r(x) &\Leftrightarrow (x = e \vee x = f) \end{aligned} \tag{1}$$

With some small caveats, a nonrecursive set of biconditional definitions is guaranteed to comprise a complete theory.

Definition 3 (Biconditional Definition). *A biconditional definition is a sentence of the form $r(\overline{x}) \Leftrightarrow \phi(\overline{x})$, where r is a relation constant, $\overline{x}$ is a sequence of non-repeating variables, and $\phi(\overline{x})$ is a sentence with free variables $\overline{x}$.*

Definition 4 (Nonrecursive biconditional definitions). *A set of biconditional definitions Δ is nonrecursive if and only if the dependency graph $\langle V, E \rangle$ is acyclic.*

V : the set of predicates in Δ

E : $\langle u, v \rangle$ is a directed edge if and only if there is a biconditional in Δ with u in the head and v in the body.

Theorem 1 (Biconditional Completeness). *Suppose Δ is a finite set of nonrecursive, biconditional definitions in FHL, where there is exactly one definition for each predicate in Δ and no definition for $=$. Δ is complete.*

In this paper we examine algorithms that attempt to reformulate a sentence set into a logically-equivalent set of nonrecursive biconditional definitions. Though incomplete, these algorithms run in low-order polynomial time, making them practical for certain classes of theories. These algorithms perform the recognition task by partitioning the sentence set and examining each partition individually. If each partition can be written as a single biconditional then by the way the partitions are chosen, the theory is guaranteed to be complete.

To partition the sentences, we ask the following question. Suppose the sentence set were originally written as nonrecursive, biconditional definitions. Further suppose that each definition were then rewritten independently of all the others without introducing any additional predicates while preserving logical equivalence. For each predicate p, how do we find all those sentences that were produced from the biconditional definition for p?

For example, the following set of clauses were produced by converting the nonrecursive, biconditional definitions for p, q, and r in Formula 1 into clausal form.

$$
\begin{array}{l}
p(x) \vee \neg q(x) \vee \neg r(x) \\
\neg p(x) \vee q(x) \\
\neg p(x) \vee r(x) \\
q(x) \vee x \neq a \\
q(x) \vee x \neq b \\
\neg q(x) \vee x = a \vee x = b \\
r(x) \vee x \neq e \\
r(x) \vee x \neq f \\
\neg r(x) \vee x = e \vee x = f
\end{array}
$$

How does one determine which clause came from which definition?

Because the original biconditionals were nonrecursive, there must be at least one of them whose body is expressed entirely in terms of equality; call each such biconditional a base table definition. All those clauses that contain one predicate besides equality must have originated in base table definitions. Within that set of clauses, all those that constrain the same predicate must have come from the definition for that predicate.

In the example above, applying this observation selects two sets of sentences.

$$
\begin{array}{l}
q(x) \vee x \neq a \\
q(x) \vee x \neq b \\
\neg q(x) \vee x = a \vee x = b
\end{array}
$$

and

$$
\begin{array}{l}
r(x) \vee x \neq e \\
r(x) \vee x \neq f \\
\neg r(x) \vee x = e \vee x = f
\end{array}
$$

The first is the set of sentences that came from the definition for q, and the second is the set of sentences that came from the definition for r.

The partitioning algorithm recurses on the remaining sentences, this time looking for sentences with a single predicate besides $=$ and the base table predicates. At iteration i,

the sentences that originated from definitions for predicates $p_1, \ldots, p_j$ have been found and the algorithm looks for sentences with a single predicate besides $\{p_1, \ldots, p_j, =\}$. This is a variation on the context-free grammar (CFG) marking algorithm: when the partition is found for predicate p, all occurrences of p in the remaining sentences are marked, and when a sentence has all but one of its predicates marked, it is added to the partition corresponding to the remaining predicate.

In the example, the remaining sentences are

$$p(x) \vee \neg q(x) \vee \neg r(x)$$
$$\neg p(x) \vee q(x)$$
$$\neg p(x) \vee r(x)$$

and since each contains one predicate, p, besides the predicates $\{q, r, =\}$, all of these sentences are grouped into the partition corresponding to the biconditional for p.

Alg. 1 embodies the approach outlined here. Each recursive call selects all those sentences with the same unmarked predicate p and calls REFORMULATE-TO-BICOND on those sentences. If they entail a biconditional for p, that biconditional is deemed to be the definition for p, and the algorithm recurses. If no such biconditional is entailed, the algorithm finds a different unmarked predicate and tries again. If there is no biconditional entailed for any of the unmarked predicates, the theory is incomplete.

Algorithm 1 TO-BICONDS(Δ, *basepreds*)

```
1: sents := {⟨p, d⟩|d ∈ Δ and p ∉ basepreds and Preds[d] is p plus a subset of basepreds}
2: preds := {p|⟨p, d⟩ ∈ sents}
3: bicond := NIL
4: for all p ∈ preds do
5:    partition := {d|⟨p, d⟩ ∈ sents}
6:    bicond := REFORMULATE-TO-BICOND(partition, p)
7:    pred := p
8:    when bicond then exit for all
9: end for
10: when not(bicond) then return NIL
11: remaining := Δ − partition
12: when remaining = ∅ then return cons(bicond, NIL)
13: rest := TO-BICONDS(remaining, basepreds ∪ {pred})
14: when rest then return cons(bicond, rest)
15: return NIL
```

TO-BICONDS runs in $O(n^2(m + r))$, where n is the number of predicates in Δ, m is the size of Δ, and r is the cost of REFORMULATE-TO-BICOND. Under the conditions mentioned above, TO-BICONDS is sound and complete as long as REFORMULATE-TO-BICOND is sound and complete.

Theorem 2 (Soundness of TO-BICONDS). *Under the conditions listed below, if* TO-BICONDS($\Delta, \{=\}$) *returns the non-NIL* Γ*, then* Γ *is a set of nonrecursive biconditionals with one definition per predicate in* Δ *besides equality and* Δ *is logically equivalent to* Γ*.*

- Δ *is a satisfiable sentence set in FHL*
- REFORMULATE-TO-BICOND *is sound, i.e. if* REFORMULATE-TO-BICOND(Σ, p) *returns* $p(\overline{x}) \Leftrightarrow \phi(\overline{x})$ *then the result is a biconditional definition for* p *entailed by* Σ.

Theorem 3 (Completeness of TO-BICONDS). *Under the conditions listed below, if* Δ *is a complete theory in FHL then* TO-BICONDS$(\Delta, \{=\})$ *does not return NIL.*

- Δ *is a satisfiable, nonempty sentence set that can be partitioned into sets so that there is one set per relation constant in* Δ *besides equality:* $S_{r_1}, \ldots, S_{r_n}$.
- Δ *is logically equivalent to a set of nonrecursive biconditionals with one definition per relation constant besides equality:* $\{\beta_{r_1}, \ldots, \beta_{r_n}\}$
- β_{r_i} *is logically equivalent to* S_{r_i} *and* $Preds[\beta_{r_i}] = Preds[S_{r_i}]$*, for every* i
- REFORMULATE-TO-BICOND *is complete, i.e. if* Σ *entails some nonrecursive biconditional definition* $p(\overline{x}) \Leftrightarrow \phi(\overline{x})$ *then* REFORMULATE-TO-BICOND(Σ, p) *returns an equivalent biconditional definition for* p*; otherwise, it returns NIL.*

TO-BICONDS relies on REFORMULATE-TO-BICOND, an algorithm for checking whether a given set of sentences entails a biconditional definition. There are many such algorithms, which form a spectrum of inexpensive syntactic checks to expensive semantic checks. Our approach lies closer to the inexpensive end of the spectrum: it attempts to rewrite the sentences of a partition into sufficient conditions for p, $p(\overline{x}) \Leftarrow \phi(\overline{x})$, and necessary conditions for p, $p(\overline{x}) \Rightarrow \psi(\overline{x})$. It then tests whether $\phi(\overline{x})$ and $\psi(\overline{x})$ are logically equivalent, and again there is a spectrum of algorithms for checking logical equivalence.

To finish the example, the sentences of the last partition can be written as

$$p(x) \Leftarrow (q(x) \wedge r(x))$$
$$p(x) \Rightarrow (q(x) \wedge r(x))$$

Clearly there is a biconditional definition entailed by these two impications; likewise for the other partitions in the example. Notice that because of the way the partitions were constructed, the resulting set of biconditionals must be nonrecursive.

$$p(x) \Leftrightarrow (q(x) \wedge r(x))$$
$$q(x) \Leftrightarrow (x = a \vee x = b)$$
$$r(x) \Leftrightarrow (x = e \vee x = f)$$

3.2 Translating Complete Theories into Datalog

One of the benefits of the recognition algorithm described in the last section is that when successful, the algorithm produces a formulation of the theory, a set of nonrecursive biconditionals, that is amenable to translation into *datalog*$^\neg$; that transformation is the subject of this section.

The translation takes place in two steps. First, the set of nonrecursive biconditional definitions is partitioned into the portion that is turned into data (extensional tables)

and the portion that is turned into views (intensional tables). Second, the extensional portion is converted into explicit tables and the intensional portion is converted into *datalog*$^\neg$ view definitions.

The approach reported in this paper for partitioning the set of biconditionals is a simple one. As mentioned in the last section, every set of biconditional definitions must include at least one definition whose body is expressed entirely in terms of equality. Our partitioning algorithm assigns all such definitions to the extensional portion of the theory and all the remaining definitions to the intensional portion. More sophisticated partitioning algorithms are the subject of future work.

Regardless of what algorithm is used to split the set of biconditional definitions into the extensional and the intensional portions, the algorithms for transforming those biconditionals into *datalog*$^\neg$ are fairly straightforward.

Turning a biconditional definition into a *datalog*$^\neg$ view definition can be performed using a typical recursive walk of the sentence and a post-processing step at the end. The walk turns $\forall$ into $\neg\exists\neg$; it introduces new predicates when negation and $\exists$ are encountered; boolean connectives are treated as usual. We demonstrate by showing a step-by-step transformation of $r(x) \Leftrightarrow \forall y.(\neg p(y) \vee q(x,y))$.

$r(x) \Leftrightarrow \forall y.(\neg p(y) \vee q(x,y))$
$r(x) :- \forall y.(\neg p(y) \vee q(x,y))$ ($\Leftrightarrow$ turned into $:-$)
$r(x) :- \neg\exists y.\neg(\neg p(y) \vee q(x,y))$ ($\forall$ turned into $\neg\exists\neg$)
$r(x) :- \neg\exists y.(p(y) \wedge \neg q(x,y))$ ($\neg$ pushed through)
(1) $r(x) :- not(newreln(x))$ (new predicate invented)
where $newreln(x) \Leftrightarrow \exists y.(p(y) \wedge \neg q(x,y))$
$newreln(x) :- \exists y.p(y) \wedge \neg q(x,y)$ ($\Leftrightarrow$ turned into $:-$)
(2) $newreln(x) :- p(y) \wedge \neg q(x,y)$ (drop $\exists$)

The result is two rules (1) and (2), one for r and one for $newreln$. Notice that neither rule is safe—there are variables in negative literals that do not occur in positive literals. To be valid *datalog*$^\neg$, safety must be ensured. To achieve this, we introduce a new predicate $univ$ that is true of all the object constants in the vocabulary, i.e. all the objects in the universe. Then we add $univ(x)$ for every variable x that is not safe in some rule. The result is shown below.

$$r(x) :- not(newreln(x)) \wedge univ(x)$$
$$newreln(x) :- p(y) \wedge not(q(x,y)) \wedge univ(x)$$

Because the biconditional definitions are nonrecursive, the algorithm illustrated above, which we will refer to as VIEWS-TO-DATALOG, is guaranteed to produce a set of safe, stratified *datalog*$^\neg$ rules. The algorithm used for converting the body of the rule will convert any sentence into *datalog*$^\neg$; it will be called SENTENCE-TO-DATALOG.

VIEWS-TO-DATALOG can also be used to convert a biconditional into a table of data. Suppose we convert all the biconditionals, both those destined to become data and those destined to become views, into *datalog*$^\neg$ views using VIEWS-TO-DATALOG, treating $=$ as an uninterpreted predicate. We can then add in a table for $=$, which has one row per element in the universe. Then we can materialize a table by constructing the appropriate *datalog*$^\neg$ query and running it through a standard deductive database engine. In the case of very large theories, here again, we are relying on database algorithms to do what

they do best—manipulate large amounts of data. We will refer to the algorithm for converting a biconditional into an extensional table with the name MATERIALIZE.

The partitioning algorithm, the algorithm for converting a sentence into *datalog*$^\neg$, and the algorithm for transforming biconditionals into views and data are bundled together into BICONDS-TO-DATALOG, an algorithm for converting an entailment query in FHL about a set of nonrecursive biconditionals into *datalog*$^\neg$.

Algorithm 2 BICONDS-TO-DATALOG(Δ, ϕ)

Assumes: Δ is a set of nonrecursive biconditionals and ϕ is in the language of Δ

1: $(D, V) :=$ PARTITION(Δ)
2: $\Gamma :=$ MATERIALIZE($Preds[D]$, Δ)
3: $\Lambda :=$ VIEWS-TO-DATALOG(V)
4: $\psi :=$ SENTENCE-TO-DATALOG(ϕ)
5: **return** (Γ, Λ, ψ)

BICONDS-TO-DATALOG ensures that Δ entails ϕ under FHL semantics if and only if $\Gamma \cup \Lambda$ entails ψ under Datalog semantics. BICONDS-TO-DATALOG ensures something much stronger as well: Δ and $\Gamma \cup \Lambda$ have all the same models (under their respective semantics). This ensures all the logical consequences are exactly the same; the transformation preserves not only entailment of the query in question but of all queries in the language.

Theorem 4 (Equivalence Preservation of BICONDS-TO-DATALOG). *Let $\Delta \models_{FHL} \phi$ be a logical entailment query where Δ is a set of nonrecursive biconditionals with one definition per predicate besides equality, and let ϕ be in the language of Δ. Suppose* BICONDS-TO-DATALOG *produces (Γ, Λ, ψ). $\Delta \models_{FHL} \phi$ if and only if $\Gamma \cup \Lambda \models_D \psi$, under the following assumptions.*

- $\psi =$ SENTENCE-TO-DATALOG(ϕ)
- *If Δ is a set of nonrecursive biconditionals, then $(D, V) = Partition(\Delta)$ is a partitioning of Δ.*
- *If D is a set of nonrecursive biconditionals for predicates $\{r_1, \ldots, r_n\}$, then $\Gamma =$* MATERIALIZE*$(Preds[D], \Delta)$ consists of a table for each r_i, i.e. $\overline{a}$ is a tuple in the r_i table in Γ if and only if $\Delta \models r_i(\overline{a})$.*
- *If V is a set of nonrecursive biconditionals, then $\Lambda =$* VIEWS-TO-DATALOG*(V) is a nonrecursive, stratified set of datalog$^\neg$ rules such that for every set E of nonrecursive biconditional definitions for Preds[D],*

$$E \cup V \models_{FHL} \phi \text{ if and only if } \text{MATERIALIZE}(Preds[D], E) \cup \Lambda \models_D \psi$$

3.3 Extensional Reasoning

When given a logical entailment query $\Delta \models \phi$, the recognition algorithm TO-BICONDS determines whether Δ is complete and if so uses the transformation algorithm BICONDS-TO-DATALOG to turn the query into *datalog*$^\neg$. Then an algorithm for processing *datalog*$^\neg$ is

employed to answer the query. If Δ is not complete, traditional algorithms are employed to determine whether entailment holds. The following algorithm, ER-ENTAILEDP, relies on DB-ENTAILEDP to answer the *datalog*$^\neg$ query and FHL-ENTAILEDP to answer an arbitrary entailment query in FHL in the case of incomplete theories.

Algorithm 3 ER-ENTAILEDP(Δ, ϕ)

Returns: T if and only if $\Delta \models_{FHL} \phi$

```
1: Σ := TO-BICONDS(Δ)
2: if Σ then
3:    (Γ, Λ, ψ) := BICONDS-TO-DATALOG(Σ,φ)
4:    return DB-ENTAILEDP(Γ, Λ, ψ)
5: else
6:    return FHL-ENTAILEDP(Δ, φ)
7: end if
```

The theorems from previous sections ensure ER-ENTAILEDP is sound and complete.

Theorem 5 (Soundness and Completeness). *Suppose ϕ is a sentence in the language of Δ.* ER-ENTAILEDP(Δ, ϕ) *returns T if and only if $\Delta \models_{FHL} \phi$.*

One of the keys to the proof of this theorem bridges the gap between how the semantics of logic and the semantics of *datalog*$^\neg$ treat negation. Logic uses classical negation whereas *datalog*$^\neg$ uses negation as failure. While NAF is often thought of as a nonmonotonic and therefore unsound inference rule, in the case of complete theories, NAF is sound.

Theorem 6 (Soundness of NAF). *Negation as failure is a sound rule of inference when it is applied to a closed sentence in the language of an axiomatically complete theory while using a complete proof procedure.*

Proof. NAF is a meta inference rule based on a proof system $\vdash$. NAF infers $\neg\phi$ whenever $\Delta \nvdash \phi$. Suppose that ϕ is closed, Δ is axiomatically complete (entails ψ, $\neg\psi$, or both for every closed sentence ψ), and $\vdash$ is complete (finds a proof whenever entailment holds). Then, if $\Delta \nvdash \phi$, by the completeness of $\vdash$, $\Delta \not\models \phi$. By the completeness of Δ, we have $\Delta \models \neg\phi$. Thus, when $\Delta \nvdash \phi$, $\Delta \models \neg\phi$, which is the conclusion NAF produces; thus, it is sound. □

3.4 A Comparison of ER and Traditional Techniques

The central claim of this paper is that sometimes answering an entailment query using algorithms for processing *datalog*$^\neg$ is more efficient than using the typical algorithms for processing FHL. One of the main benefits of *datalog*$^\neg$ is its use of Negation as Failure, which happens to be a sound rule of inference in the case of complete theories. This section compares NAF to traditional treatments of negation from both the theoretical and the empirical perspective.

Theoretical Comparison

To demonstrate the power of NAF, we start by comparing SLDNF resolution and model elimination on an example, a fair comparison since the two proof procedures differ mainly in how they treat negation. SLDNF uses NAF, whereas model elimination uses classical negation. Consider the following biconditional definition for p.

$$p(x) \Leftrightarrow \left(\begin{array}{l} (p_1(x) \wedge p_2(x) \wedge p_3(x)) \vee \\ (p_4(x) \wedge p_5(x) \wedge p_6(x)) \vee \\ (p_7(x) \wedge p_8(x) \wedge p_9(x))) \end{array} \right)$$

Converting the biconditional above to view definitions using the algorithms of the last section basically amounts to dropping the $\Rightarrow$.

$$\begin{array}{lll} p(x) & :- & p_1(x) \wedge p_2(x) \wedge p_3(x) \\ p(x) & :- & p_4(x) \wedge p_5(x) \wedge p_6(x) \\ p(x) & :- & p_7(x) \wedge p_8(x) \wedge p_9(x) \end{array}$$

In contrast, converting the biconditional to clausal form produces the implications above along with implications for $\neg p(x)$. In this example, there are 27 possible ways to prove $\neg p(t)$ for a particular t, which is exponential in the size of the biconditional. The naive clausal form conversion will construct one clause for each one. (Structure-preserving clausal form conversion, which can be found in chapters 5 and 6 of [RV01], ensures the number of clauses is polynomial in the size of the original sentence set, but as we will see the size of the search space can still be exponential.)

$$\begin{array}{c} \neg p(x) \Leftarrow \neg p_1(x) \wedge \neg p_4(x) \wedge \neg p_7(x) \\ \neg p(x) \Leftarrow \neg p_1(x) \wedge \neg p_4(x) \wedge \neg p_8(x) \\ \neg p(x) \Leftarrow \neg p_1(x) \wedge \neg p_4(x) \wedge \neg p_9(x) \\ \vdots \\ \neg p(x) \Leftarrow \neg p_3(x) \wedge \neg p_6(x) \wedge \neg p_9(x) \end{array}$$

We compare SLDNF resolution on the $datalog^{\neg}$ with model elimination on the clauses. For the entailment query $\exists x.p(x)$, the two perform identically (assuming we turn off the reduction operation [AS92] for model elimination); however, for the query $\exists x.\neg p(x)$, the two techniques differ significantly. SLDNF resolution uses the rules with $p(x)$ in the head to look for a t such that the proof for $p(t)$ fails. Model elimination uses the rules with $\neg p(x)$ in the head to look for a t such that $\neg p(t)$ has a proof. Depending on the relative sizes of the universe, the search space for $p(x)$, and the search space for $\neg p(x)$, proving $\neg p(t)$ by exhausting the search space for $p(t)$ can be far less expensive than finding a proof in the search space for $\neg p(t)$.

Information theoretically, it is not surprising that for complete theories, NAF sometimes answers queries more quickly than methods for classical negation. NAF can be viewed as a mechanism for reasoning about complete theories, whereas methods for classical negation are mechanisms for reasoning about incomplete theories. NAF implicitly leverages the fact that the theory is complete, but methods for classical negation do not.

More tangibly, the tradeoff between NAF and classical negation can be seen as a tradeoff of search spaces. Often the space for $\neg p$ is larger than the space for p, and it is this case that NAF takes advantage of. As opposed to classical negation, NAF

avoids searching the large space and instead searches just the small space. (We might additionally consider Truth as Failure to handle the case where the space for p is very large but the space for $\neg p$ is very small.)

As mentioned earlier, structure-preserving clausal-form conversion will avoid enumerating the exponential number of rules with $\neg p$ in the head, but that does not necessarily prevent resolution from enumerating an exponential search space. Moreover, with certain assumptions placed on what it means to convert to clausal form, one can show that regardless of which clausal form conversion is used, there is an infinite class of biconditionals such that resolution has the potential to generate exponentially many clauses in the size of the biconditional, assuming that the implementation of resolution is generatively complete.

Lemma 1. *Consider the following biconditional β.*

$$p(x) \Leftrightarrow \begin{pmatrix} (p_{11}(x) \wedge \cdots \wedge p_{1n_1}(x)) \vee \\ \vdots \\ (p_{m1}(x) \wedge \cdots \wedge p_{mn_m}(x)) \end{pmatrix}$$

When applying the naive clausal form conversion to β, the number of clauses with a negative p literal is the product of the lengths of the disjunctions: $\prod_i n_i$.

$$\begin{gathered} \neg p(x) \vee p_{11}(x) \vee p_{21}(x) \vee \cdots \vee p_{m1}(x) \\ \neg p(x) \vee p_{11}(x) \vee p_{21}(x) \vee \cdots \vee p_{m2}(x) \\ \vdots \\ \neg p(x) \vee p_{1n_1}(x) \vee p_{2n_2}(x) \vee \cdots \vee p_{mn_m}(x) \end{gathered}$$

Suppose a clausal form conversion algorithm converts β into Γ. Further suppose Γ it is logically equivalent to β with respect to the predicates in β, i.e. for every sentence σ whose predicates are a subset of those in β, $\Gamma \models \sigma$ iff $\beta \models \sigma$. Suppose Res is an implementation of resolution that is generatively complete.

$$Res[\Gamma \cup \{p(t)\}] \text{ will produce at least } \prod_i n_i \text{ clauses.}$$

Proof. Here we argue somewhat informally. Consider an arbitrary clause with a negative p literal.

$$\neg p(x) \vee p_{1j_1}(x) \vee p_{2j_2}(x) \vee \cdots \vee p_{mj_m}(x)$$

This clause, which is entailed by β, entails

$$p(t) \Rightarrow p_{1j_1}(t) \vee p_{2j_2}(t) \vee \cdots \vee p_{mj_m}(t)$$

Consequently Γ entails it as well. Thus, $\Gamma \cup \{p(t)\}$ entails

$$p_{1j_1}(t) \vee p_{2j_2}(t) \vee \cdots \vee p_{mj_m}(t)$$

Since *Res* is generatively complete up to subsumption, $Res[\Gamma \cup \{p(t)\}]$ must include either this disjunction, or some clause that subsumes it. No clause that subsumes this one is entailed by $\Gamma \cup \{p(t)\}$; hence, this clause must appear in the closure. Since the clause was chosen arbitrarily, the same holds for all clauses. Since there are $\prod_i n_i$ of these clauses, the resolution closure must contain at least that many clauses. □

The above lemma guarantees a local property about resolution—that given a single biconditional and a query, the resolution closure is exponentially large in the number of disjunctions. When other sentences are included, the lemma makes no guarantees. For example, if resolution were given a biconditional for $p(x)$ along with the sentence $\neg p(x)$, resolution could find a proof without enumerating the exponential number of consequences described above.

To truly understand a technique it is important to find its limitations. Extensional Reasoning is not always superior to traditional techniques. Consider a satisfiable, complete premise set that consists of a single sentence $\forall xy.p(x,y)$, which when converted to clausal form is just $p(x,y)$. The query $\forall xy.p(x,y)$ when negated and converted to clausal form is $\neg p(k_1,k_2)$. Resolution finds a proof in a single step, regardless of how large the universe is.

In Extensional Reasoning, if the decision is made to materialize p, the database includes n^2 tuples for p, where n is the size of the universe. Using SLDNF resolution, the proof attempt $\forall xy.p(x,y)$ is performed by attempting to find elements k_1 and k_2 such that $\neg p(k_1,k_2)$ is true. Since every attempt fails, even assuming perfect indexing, the proof costs n^2.

Thus, in some cases Extensional Reasoning can find proofs in complete theories more efficiently than traditional techniques, but this last example demonstrates that this is not always the case. An important next step is to investigate algorithms that predict which approach will find a proof (or fail to find a proof) more quickly; such algorithms are the subject of future work.

Empirical Comparison

Next we report on experiments designed to compare Extensional Reasoning techniques to traditional techniques in the case of complete theories. Since the algorithms for detecting completeness presented in this paper produce a set of nonrecursive biconditional definitions, all the tests we performed were on such sentence sets. The results demonstrate what is predicted above. When negation occurs in the sentences, the *datalog*$^\neg$ implementation, which uses Negation as Failure, is significantly faster than techniques that do not employ NAF.

Each sentence set represents the layout of a two-dimensional grid, like the one shown in Fig. 2. Every sentence set has exactly six biconditionals representing the following information.

- $west(x,y)$: cell x is the cell immediately west of cell y.
- $north(x,y)$: cell x is the cell located immediately north of cell y.
- $duewest(x,y)$: cell x is in the same row as cell y and located to the west.
- $duenorth(x,y)$: cell x is in the same column as cell y and located to the north.
- $vert(x,y)$: cell x is $duenorth$ of y or vice versa.
- $westof(x,y)$: cell x is located to the west of y, i.e. x and y can be in different rows but x's column is to the west of y's column.

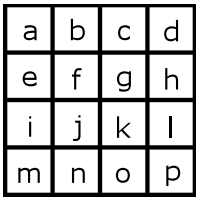


Figure 2: A 4×4 grid, corresponding to the theory in the appendix.

An axiomatization for the 4×4 case can be found in the appendix.

The tests varied the dimensions of the grid, starting at 4×4 and ending at 20×20. The queries were of the form $westof(t, u)$ and $\neg westof(t, u)$, where t and u were always ground. Every query tested was entailed. The results of the positive queries are reported separately from the results of the negative queries because negation plays a prevalent role in the negative queries but not the positive. The positive queries provide a baseline for comparing techniques on how well they handle nonrecursive biconditional definitions. The negative queries illustrate how NAF can affect performance.

We compared our implementation of Extensional Reasoning for the case of complete theories, DBD (Datalog Based Deduction), with Vampire 8.1, Darwin, and Epilog (the Stanford Logic Group's model-elimination implementation[1]). Vampire and Darwin were run using SystemOnTPTP, and Epilog and DBD were run in Macintosh Common Lisp on a 1.5 GHz G4 Powerbook with 1.25 GB of RAM.

As expected, DBD, the system built to reason about towers of biconditionals, outperformed the three general-purpose systems on both the positive and the negative queries. Each system had 1000 seconds to find a proof. Most of the systems performed fairly similarly until the last or second-to-last grid size they solved under the time limit.

For the positive queries (Fig. 3), Epilog and Darwin performed similarly, both failing to find a proof in 1000 seconds for the 16×16 grid. Vampire failed to find a proof at 20×20; DBD solved the 20×20 in 126 seconds. These tests were run without the DCA or the UNA since entailment did not require them; thus, these tests primarily illustrate how the various systems cope with the potentially large number of clauses generated by converting biconditionals to clausal form.

[1]For completeness, model elimination and therefore Epilog require all contrapositives of the clauseset to be explored. For these experiments, only those clauses produced by a typical clausal form conversion were used.

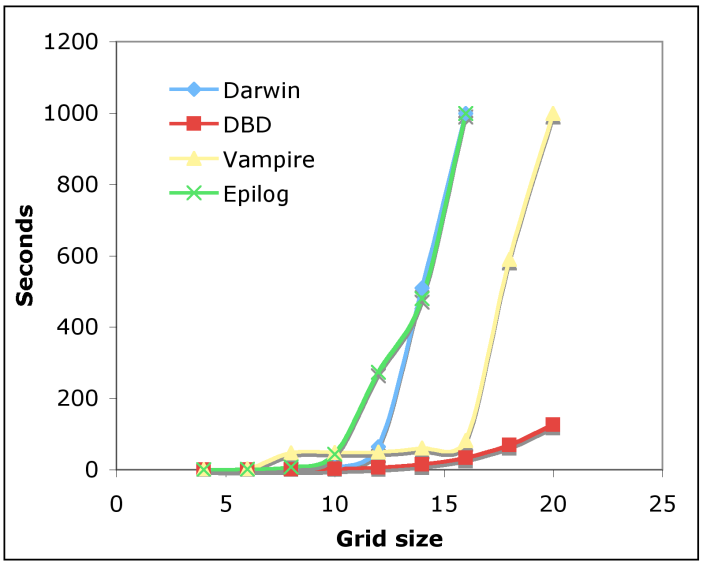


Figure 3: Results for queries of the form $westof(t, u)$.

$n \times n$	Darwin	Vampire	Epilog	DBD
4	0	0	0	0
6	0	3	1	0
8	1	47	7	1
10	7	48	43	2
12	65	50	152	6
14	509	61	480	15
16	> 999	82	> 999	33
18		589		69
20		> 999		126

For the negative queries (Fig. 4), the difference between DBD and the others is larger. Vampire failed at size 8×8; Epilog failed at size 12×12; Darwin failed at 16×16. DBD solved 20×20 in 187 seconds. This difference appears to be due mainly to the fact that DBD uses NAF, and the negative search space for *westof* (as induced by the clauses with negative *westof* literals) is substantially larger than the positive search space. These tests required the UNA, but not the DCA.

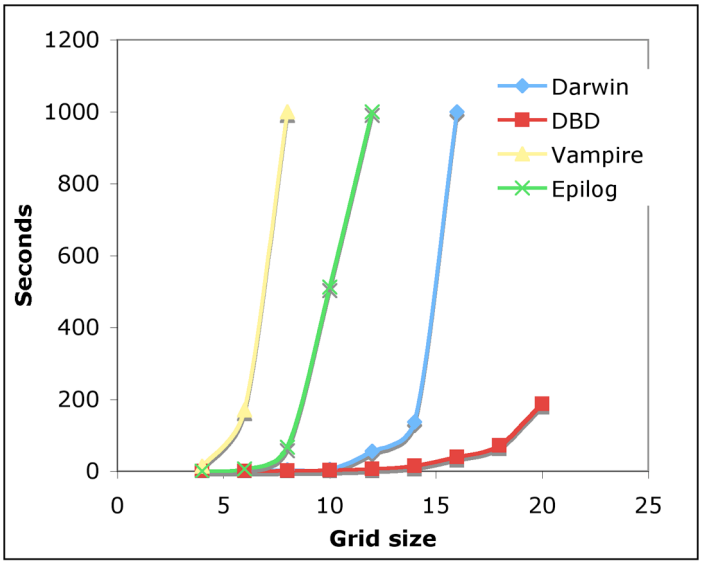


Figure 4: Results for queries of the form $\neg westof(t, u)$.

$n \times n$	Darwin	Vampire	Epilog	DBD
4	0	14	0	0
6	0	170	6	0
8	4	> 999	67	1
10	5		512	2
12	55		> 999	6
14	137			15
16	> 999			40
18				72
20				187

We also worked on answering entailment queries using boolean SAT solvers, since every FHL theory can always be converted into propositional logic. Conversion to clausal form after grounding proved to be prohibitively expensive, even using the structure-preserving version of clausal form conversion. Comparing DBD to a non-clausal SAT solver is the subject of future work.

These experiments compare how quickly various sytems can prove theorems when the search space is shallow but has a high branching factor. Typical database applications have this character: the majority of the cost comes from manipulating large amounts of data, and the tower of view definitions built on top of the extensional tables is relatively short. As expected, in such situations Negation as Failure can produce significant savings over treating negation classically.

4 Incomplete Theories

Complete theories have powerful properties, but incomplete theories are the norm. The last section detailed techniques for reasoning efficiently about complete theories using database techniques. However, if one were to add even just a small amount of incompleteness into a complete theory by including new predicates, those techniques could no longer be applied. This is unfortunate since the speedups in the complete case seem to be large enough to absorb some extra overhead for dealing with theories that have a small amount of incompleteness.

For example, suppose we take any complete theory and add in the following sentences, where p and q are new predicates.

$$p(a) \vee p(b) \vee p(c)$$
$$q(d) \vee q(e) \vee q(f)$$

Supposing the complete theory had a definition for the binary predicate r, a reasonable entailment query might be

$$\forall xy.(p(x) \wedge q(y) \Rightarrow r(x, y)). \tag{2}$$

When the definitions for r are large enough to warrant using a database system, we probably still want to use that database system in spite of the fact that we now have an incomplete theory.

Theory resolution [Sti85] is one approach to dealing with such situations, where part of the theory can be effectively represented and reasoned about with a specialized procedure. If a theory were partitioned into the complete portion C and the incomplete portion I, theory resolution would allow us to use a database system to represent C while answering queries about $C \cup I$ using resolution.

One of the benefits of the TO-BICONDS algorithm shown in Alg. 1 is that with a two-line change, it can be used to find the portion of the theory that is complete, partitioning the theory as required for theory resolution. This change consists of replacing lines (14) and (15) so that once the algorithm fails to find a new biconditional definition, it returns all the biconditionals it has found so far. Alg. 4 gives this new algorithm, TO-BICONDS-MAX. It is noteworthy that TO-BICONDS-MAX can be turned into an anytime algorithm, allowing the user or system designer to determine how much time to spend trying to find a complete subtheory.

With the partitioning algorithm in place, we can now focus on theory resolution. Because C is complete, we can think of it as being a set of ground literals such that every ground atom or its negation is included. For the case where the incomplete portion I is in $\forall^*$, i.e. where when I is written in prenex normal form the quantifiers are all universal, theory resolution takes a particularly simple form. Suppose p is a predicate that belongs to the complete portion, i.e. every ground p literal or its negation belongs to C. Then if there were some clause

$$\{p(\bar{t})\} \cup \Phi,$$

resolution would produce a resolvent for every literal $\neg p(\bar{a})$ where $p(\bar{a})$ unifies with $p(\bar{t})$:

$$\Phi\sigma, \text{ where } \sigma \text{ is the mgu of } p(\bar{a}) \text{ and } p(\bar{t})$$

Algorithm 4 TO-BICONDS-MAX(Δ, *basepreds*)

1: $sents := \{\langle p, d\rangle | d \in \Delta$ and $p \notin basepreds$ and $Preds[d]$ is p and some subset of $basepreds\}$
2: $preds := \{p | \langle p, d\rangle \in sents\}$
3: $bicond :=$ NIL
4: **for all** $p \in preds$ **do**
5: $\quad partition := \{d | \langle p, d\rangle \in sents\}$
6: $\quad bicond :=$ REFORMULATE-TO-BICOND($partition$, p)
7: $\quad pred := p$
8: $\quad$ **when** $bicond$ **then** exit **for all**
9: **end for**
10: **when** $not(bicond)$ **then return** NIL
11: $remaining := \Delta - partition$
12: **when** $remaining = \emptyset$ **then return** $cons(bicond, \text{NIL})$
13: $rest :=$ TO-BICONDS-MAX($remaining$, $basepreds \cup \{pred\}$)
14: **return** $cons(bicond, rest)$

For completeness, theory resolution must produce all such clauses.

Definition 5 (Complete Theory Resolution). *Complete Theory Resolution (CTR) is the following rule of inference. When it is added to the usual resolution inference rules, we denote the closure of clauses S using all those rules by $CTRRes(C, S)$. Suppose C is a complete theory with a definition for predicate p.*

$$\frac{\begin{array}{l}\pm p(\bar{t}) \cup \Phi \\ \textit{Let } \{\sigma_1, \ldots, \sigma_n\} \textit{ be the set of all mgus } \sigma \textit{ such that } \mp p(\bar{t})\sigma \textit{ is entailed by } C.\end{array}}{\begin{array}{l}\Phi\sigma_1 \\ \vdots \\ \Phi\sigma_n\end{array}}$$

$\mp p(\bar{t})\sigma$ has the opposite sign of $\pm p(\bar{t})$.

In general, for FHL the DCA, UNA, and $x = x$ must be added to a set of clauses for standard implementations of resolution equipped with paramodulation to be sound and complete; however, in the case where the clauses are in $\forall^*$, it has been shown [Rei80] that neither the DCA nor paramodulation are necessary for completeness. This fact simplifies the completeness proof below.

Theorem 7 (CTRRes Soundness and Completeness for $\forall^*$). *Suppose $\Delta = C \cup I$ is a finite set of FHL sentences, where C is a satisfiable, complete theory with definitions for predicates P, and I is in $\forall^*$. Δ is unsatisfiable if and only if $CTRRes(C, I)$ contains the empty clause.*

Proof. (Soundness) Every resolution inference rule is sound, which means we need only show CTR is sound. But this is immediate because CTR is simply n applications of resolution, using literals from C.

(Completeness) A database system compactly represents C, which is semantically a finite set of ground literals. We show that CTRRes is complete by showing that

every inference step that could occur using resolution between C, represented as a set of ground literals, and I will also occur in $CTRRes(C, I)$.

Because C is a set of ground literals, it is in $\forall^*$, and I is in $\forall^*$ by assumption; thus, $C \cup I$ is in $\forall^*$, which means neither the DCA nor paramodulation are necessary for resolution to be complete. Thus, the only necessary rules of inference are binary resolution and factoring; moreover, only those inferences that use as a premise some literal from C could cause incompleteness.

If Δ is unsatisfiable then there is a resolution proof of the empty clause from $C \cup I$. Consider any step in which one of the literals from C is resolved with a non-unary clause.

$$\frac{\begin{array}{c}\pm p(\bar{t}) \cup \Phi \\ \mp p(\bar{a}) \text{ (from } C)\end{array}}{\Phi\sigma, \text{ where } \sigma \text{ is the mgu of } p(\bar{t}) \text{ and } p(\bar{a})}$$

In $CTRRes(C, I)$, this resolvent is one of many produced when CTR is applied to the first clause above. CTR finds all variable assignments $\rho_1, \ldots, \rho_m$ so that $\mp p(\bar{t})\rho_i$ belongs to C. Because every literal in C is ground, the unifier σ is a variable assignment such that $p(\bar{t})\sigma$ belongs to C, i.e. σ is one of the ρ_i. Since CTR produces all $\Phi\rho$ and σ is some ρ_i, CTR certainly produces $\Phi\sigma$.

For every resolution between some literal in C and some other literal, we know that the other literal could not have come from C because that would make C unsatisfiable; the above argument applies to this case as well, which guarantees no binary resolution inferences are lost by using CTRRes.

Hiding C does not result in the loss of any factoring step since factoring does not apply to unit clauses. Altogether, every inference rule application using resolution can be mirrored using CTRRes. □

In effect, this result is a practical approach to enlarging Reiter's result [Rei80] that eliminates the need for paramodulation in the case of $\forall^*$. Here we have shown that regardless what prefix class the complete portion of the theory belongs to, we can avoid paramodulation as long as the remaining sentences are in $\forall^*$.

We have speculated on inference rules for performing theory resolution when the incomplete sentences do not belong to $\forall^*$. The difficulty is that skolems exist, which must be dealt with by the database; moreover, the proof of completeness is complicated by the fact that paramodulation is necessary. Here we simply illustrate the issues and point toward an avenue with promise.

Consider an example where the incomplete sentences are in $\exists^*\forall^*$, which causes the clausal form conversion to introduce new skolem constants but no skolem functions.

$$\begin{array}{l} p(x) \Leftrightarrow (x = a \vee x = b) \\ q(x) \Leftrightarrow (x = c) \\ \exists x.(\neg p(x) \wedge \neg q(x)) \end{array}$$

The first two sentences comprise complete definitions for p and q, and the last sentence belongs to the incomplete portion. These sentences are inconsistent because every element is either in p or in q, but the last sentence says that there is some element in neither p nor q. The definitions for p and q are stored in the database, which leaves the existential to be manipulated by resolution. The clauses we start with (UNA left out for brevity) are as follow.

1. $x = a \vee x = b \vee x = c$
2. $\neg p(k)$
3. $\neg q(k)$

Notice here that not only must the database be used to answer queries with skolems, but the result of such queries must include information about skolems. Concentrating on the $\neg p(k)$ clause, we see that if k is equal to any one of the values true of p in C, then we have an inconsistency; or equivalently, k cannot be equal to any one of those values if the sentences are consistent. This line of reasoning produces the following two resolvents.

4. $k \neq a$
5. $k \neq b$

Applying the same rule of inference to $\neg q(k)$ produces the following resolvent.

6. $k \neq c$

Together these three resolvents are inconsistent with the DCA, which, by the completeness of resolution, ensures the production of the empty clause.

Further work needs to be done to determine whether an inference rule built around this idea would be complete. The next step would be to build an inference rule for handling skolem functions as well as skolem constants.

The downside to the theory resolution approach is that if large amounts of data are stored in the database, and even a small fraction of that data must be used for a proof, we still must address the problem of building theorem provers that can handle massive amounts of data.

Our approach is to avoid theory resolution altogether by completing the incomplete theory, and using the techniques described in the previous section to answer questions about the theory. This gives the database the opportunity to solve the entire problem itself, managing the massive amount of data as it sees fit. The tricky part is performing theory completion in a way that is not so expensive as to negate the benefits of using a database to reason about the completed theory.

Our approach to theory completion, outlined in [HG07], is to first partition the theory into the complete portion and the incomplete portion using TO-BICONDS-MAX. Then we use various techniques for completing the incomplete portion of the theory while ignoring the complete portion, to the extent possible. We now have a complete theory, which can be reasoned about using the algorithms presented in the last section.

We expect this approach to work well in the context of large theories when a large portion of the theory is complete. As the size of the complete portion increases, so does the cost of manipulating the data, which increases the utility of using a database. Moreover, if the incomplete portion of the theory is small enough, running what might normally be considered expensive algorithms to perform theory completion is affordable because of the relative cost of manipulating the data. Thus, in large theories that have a small amount of incompleteness, Extensional Reasoning has the potential for large computational savings over traditional techniques.

5 Conclusion and Future Work

This paper presents Extensional Reasoning for both complete theories and incomplete theories. In the complete case it introduces a quadratic-time partitioning algorithm for rewriting a class of complete theories into a set of nonrecursive biconditional definitions and discusses issues regarding the transformation of those definitions into $datalog^{\neg}$. For the incomplete case, it introduces an anytime algorithm for finding the portion of a theory that can be transformed into a set of nonrecursive biconditionals, and theory resolution techniques that allow the complete portion to be represented with a database system. The theory resolution techniques are sound and complete when the incomplete portion of the theory is in $\forall^*$. Empirically, Extensional Reasoning techniques perform better than traditional theorem proving techniques when the theory consists of nonrecursive biconditional definitions.

Besides enlarging the class of complete theories that we can detect, the first extension to the work presented here is a better algorithm for finding a biconditional that is entailed by a given set of sentences. This problem is unlike the traditional theorem proving problem because the entailment query is a metalevel query: do these sentences entail a sentence of the form $p(\overline{x}) \Leftrightarrow \phi(\overline{x})$? We have done some work on metalevel logic [HG05] and made preliminary investigations into meta-resolution, a variant of resolution that targets metalevel logic.

Second, the algorithm illustrated in section 3 for converting a set of nonrecursive biconditional definitions into $datalog^{\neg}$ is straightforward, but in those cases where the resulting rules are unsafe, we introduce the *univ* relation, which is true of every object constant in the language. Minimizing the cases where *univ* is used can have drastic effects on run time. Because such domain-dependent queries are often explicitly disallowed in the traditional database setting, standard database query optimizers will not take advantage of the semantics of *univ*. ER will reap large benefits from augmented query-optimization algorithms.

Third, the policy we currently use for determining which portion of the theory to turn into extensional tables and which portion to turn into intensional tables needs further study. The database community has studied view materialization [Hal01] and view construction [Chi02] in depth, and those results can surely inform if not entirely address this issue.

A Example of Test Theory

$$west(x,y) \Leftrightarrow \left(\begin{array}{l} (x=a \wedge y=b) \vee \\ (x=b \wedge y=c) \vee \\ (x=c \wedge y=d) \vee \\ (x=e \wedge y=f) \vee \\ (x=f \wedge y=g) \vee \\ (x=g \wedge y=h) \vee \\ (x=i \wedge y=j) \vee \\ (x=j \wedge y=k) \vee \\ (x=k \wedge y=l) \vee \\ (x=m \wedge y=n) \vee \\ (x=n \wedge y=o) \vee \\ (x=o \wedge y=p) \end{array} \right)$$

$$north(x,y) \Leftrightarrow \left(\begin{array}{l} (x=a \wedge y=e) \vee \\ (x=e \wedge y=i) \vee \\ (x=i \wedge y=m) \vee \\ (x=b \wedge y=f) \vee \\ (x=f \wedge y=j) \vee \\ (x=j \wedge y=n) \vee \\ (x=c \wedge y=g) \vee \\ (x=g \wedge y=k) \vee \\ (x=k \wedge y=o) \vee \\ (x=d \wedge y=h) \vee \\ (x=h \wedge y=l) \vee \\ (x=l \wedge y=p) \end{array} \right)$$

$$duewest(x,y) \Leftrightarrow \left(\begin{array}{l} west(x,y) \vee \\ \exists z.(west(x,z) \wedge west(z,y)) \vee \\ \exists zw.(west(x,z) \wedge west(z,w) \wedge west(w,y)) \end{array} \right)$$

$$duenorth(x,y) \Leftrightarrow \left(\begin{array}{l} north(x,y) \vee \\ \exists z.(north(x,z) \wedge north(z,y)) \vee \\ \exists zw.(north(x,z) \wedge north(z,w) \wedge north(w,y)) \end{array} \right)$$

$$vert(x,y) \Leftrightarrow (duenorth(x,y) \vee duenorth(y,x))$$

$$westof(x,y) \Leftrightarrow (duewest(x,y) \vee \exists z.(vert(x,z) \wedge duewest(z,y)))$$

References

[AS92] Owen Astrachan and Mark Stickel. Caching and lemmaizing in model elimination theorem provers. *CADE*, 1992.

[Chi02] Rada Chirkova. *Automated Database Restructuring.* PhD thesis, Stanford University, 2002.

[Hal01] Alon Halevy. Answering queries using views: A survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, 2001.

[HG05] Timothy L. Hinrichs and Michael R. Genesereth. Axiom schemata as metalevel axioms. *AAAI*, 2005.

[HG07] Timothy L. Hinrichs and Michael R. Genesereth. Reformulation for extensional reasoning. *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation*, 2007.

[Llo84] John Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[McC82] John McCarthy. Coloring maps and the Kowalski doctrine. *Stanford Technical Report*, 1982.

[MG03] James Masters and Zelai Gungordu. Structured knowledge source integration: A progress report. *Integration of Knowledge Intensive Multiagent Systems*, 2003.

[Rei80] Raymond Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249, 1980.

[RV01] Alan Robinson and Andrei Voronkov. *Handbook of Automated Reasoning.* MIT Press and Elsevier Science, 2001.

[Sti85] Mark Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1:333–356, 1985.

Semantic Selection of Premisses for Automated Theorem Proving

Petr Pudlák[1]
[1]Charles University, Prague
petr.pudlak@mff.cuni.cz

Abstract

We develop and implement a novel algorithm for discovering the optimal sets of premisses for proving and disproving conjectures in first-order logic. The algorithm uses interpretations to semantically analyze the conjectures and the set of premisses of the given theory to find the optimal subsets of the premisses. For each given conjecture the algorithm repeatedly constructs interpretations using an automated model finder, uses the interpretations to compute the optimal subset of premisses (based on the knowledge it has at the point) and tries to prove the conjecture using an automated theorem prover.

1 Importance of selecting appropriate premisses in automated theorem proving

A proper set of premisses[1] can be essential for proving a conjecture by an automated theorem prover. Clearly, the larger the number of the initial premisses the larger the number of the inferred formulae. And as for the most proving techniques the number of inferred formulae is in general super-exponential in the number of input formulae, the impact on the performance of an automated theorem prover is quite significant. Removing even a single superfluous premiss can make the difference between being or not being able to prove the conjecture.

This major problem is even more serious when reasoning within large mathematical theories, which can contain hundreds of premisses and thousands of conjectures. In such cases, selecting proper premisses for proving the conjectures becomes a necessity.

In our previous work [Pud06a, Pud06b] we described a method for compacting proofs of conjectures. By constructing lemmas from the proofs and by syntactically analyzing both the lemmas and the proofs, we constructed sets of premisses that produced shorter proofs of given conjectures, or allowed to construct proofs much faster. However, the assumption was that for each conjecture we already know some set of premisses from which we were able to prove the conjecture (although possibly redundant and inefficient for constructing a proof). If we were not able to prove the conjecture at all, the situation was more complicated.

[1]Using the notion *axioms* for the formulae the prover uses as assumptions sometimes causes confusion, therefore we shall prefer the notion *premisses* instead. See section 2.1.

One possibility is to syntactically analyze the formulae and/or use an AI algorithm for guessing the proper premisses. Successful examples of such procedures are Josef Urban's tools for the Mizar Project [Urb06a, Urb06b], reducing axiom sets in software verification [RS98] or filtering of axioms for machine-generated problems [MP06]. Although even simple syntactic heuristics can be very effective, the syntactic approach is generally restricted to the cases when the (syntactic) structure of formulae well reflects their semantics. Clearly, this is not always the case. Moreover, the syntactic analysis is usually only a heuristic procedure that tries to learn what premisses could be needed but is likely to fail on a new kind of problem. Syntactic filters are also often incomplete in the sense that they can eventually remove too many premisses.

The procedure we shall describe in this article uses *semantic* analysis. By observing which formulae are true in which interpretations, we can get a deeper insight into the nature of a conjecture and compute such a set of premisses that is proper for proving the conjecture. As far as we are aware, this is a novel approach, which has not been researched before.

We shall focus on two interconnected goals:

- Determine which sets of premisses are sufficient for proving the conjecture;
- among those sets, choose such a set that contains no redundant premisses and that is minimal with respect to some criterion.

The criterion can be just a simple one – to minimize the number of premisses, or a more complex one, for example, to avoid premisses of a certain kind that complicate the proving process.

2 Semantic analysis of problem using interpretations

2.1 Notation

Throughout the whole article we shall always work within the domain of first-order logic. We shall always assume that a language is given and all formulae we work with are formulated in the language and are closed.

We shall denote formulae by regular letters (e.g. A, F), interpretations by calligraphic letters (e.g. $\mathcal{M}$), sets of formulae by boldface letters (e.g. $\mathbf{B}$, $\mathbf{A}_C$) and sets of sets or sets of interpretations by script letters (e.g. $\mathscr{F}$, $\mathscr{N}$). We shall also use the following symbols:

$\mathcal{M} \models B$	$\mathcal{M}$ is a model of a formula B	(1)
$\mathcal{M} \models \mathbf{B}$	$\mathcal{M}$ is a model of all formulae in $\mathbf{B}$	
	(i.e. a model of the conjunction of all formulae in $\mathbf{B}$)	(2)
$\mathbf{B} \models G$	for all possible interpretations $\mathcal{M}$, if $\mathcal{M} \models \mathbf{B}$ then $\mathcal{M} \models G$	
	(i.e. G is a consequence of $\mathbf{B}$)	(3)
$\mathbf{B} \vdash G$	G can be proved from $\mathbf{B}$	(4)

For brevity we shall often say that "a formula B avoids an interpretation $\mathcal{M}$" (resp. B avoids a set of interpretations $\mathscr{M}$) if and only if $\mathcal{M} \not\models B$ (resp. $\mathcal{M} \not\models B$ for all $\mathcal{M} \in \mathscr{M}$).

Often formulae given to the prover as a basis for (dis)proving conjectures are simply called *axioms*. However, this sometimes causes confusion, because these given formulae actually do not have to be the axioms of a particular theory. They can be also propositions that have already been proved before, lemmas, etc. Therefore, we shall instead call such assumptions *premisses*.

2.2 Basic principle

Let a conjecture C and a set of premisses $\mathbf{A}$ (finite, but presumably large) be given. We would like to prove (or to refute) C from $\mathbf{A}$ using an automated theorem prover. But since $\mathbf{A}$ is large, the prover is overloaded by the high number of premisses and we are not able to prove C directly from $\mathbf{A}$. We expect that only a small subset of $\mathbf{A}$ is necessary to prove or to refute C and we would like to find such a subset.

The idea of the semantic analysis is based on two principal properties of first-order logic:

1. If $\mathbf{B} \vdash C$ then for any interpretation $\mathcal{M}$ such that $\mathcal{M} \not\models C$ also $\mathcal{M} \not\models \mathbf{B}$. That is, there is $B \in \mathbf{B}$ such that $\mathcal{M} \not\models B$.

2. Having a set of premisses $\mathbf{B}$, then

   - either C is provable from $\mathbf{B}$, hence $\mathbf{B} \vdash C$, which is equivalent to $\mathbf{B} \models C$ by the completeness theorem,
   - or there is an interpretation $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$.

Using these two properties we shall construct a sequence of interpretations $\mathcal{M}_i$ and a sequence of premisses F_i (for convenience let us set $\mathbf{B}_i = \{F_0, \ldots, F_{i-1}\}$). At each step we select F_i to be from the set $\mathbf{A}_i = \{F \in \mathbf{A} \,|\, \mathcal{M}_i \not\models F\}$ and eventually some set $\mathbf{B}_k$ will be sufficient for proving C.

Let us start with an empty set of premisses $\mathbf{B}_0 = \emptyset$. If C is a tautology, then $\mathbf{B}_0 \vdash C$ and we are finished. Otherwise, there is some interpretation $\mathcal{M}_0$ such that $\mathcal{M}_0 \models \neg C$ (see Fig. 1a). Let

$$\mathbf{A}_0 = \{F \in \mathbf{A} \,|\, \mathcal{M}_0 \not\models F\}$$

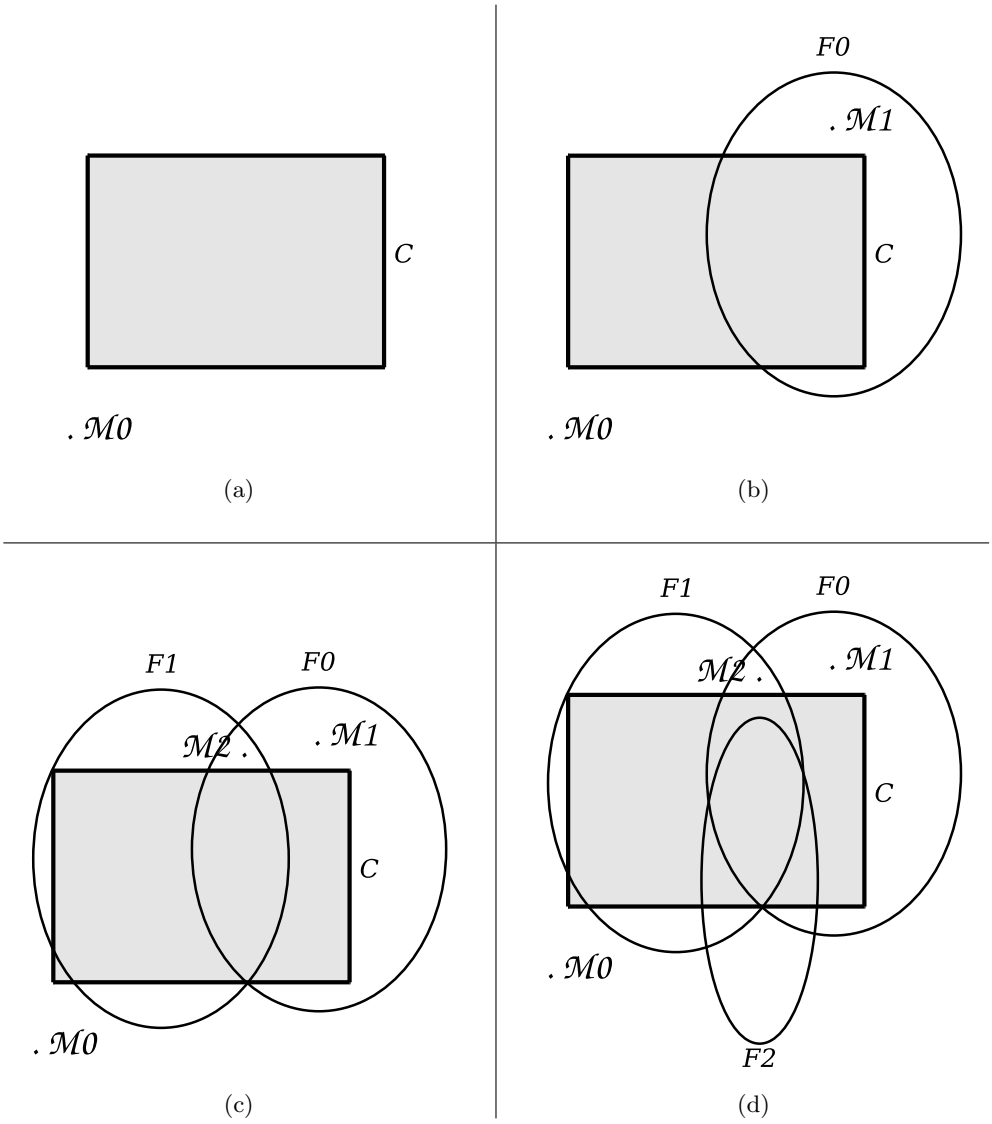
Let us select any $F_0 \in \mathbf{A}_0$ and set $\mathbf{B}_1 = \{F_0\}$. Because $\mathcal{M}_0 \not\models F_0$, also $\mathcal{M}_0 \not\models \mathbf{B}_1$. Now, either $\mathbf{B}_1 \models C$, thus $\mathbf{B}_1 \vdash C$, or there is an interpretation $\mathcal{M}_1$ such that $\mathcal{M}_1 \models \mathbf{B}_1 \cup \{\neg C\}$ (see Fig. 1b). We set

$$\mathbf{A}_1 = \{F \in \mathbf{A} \,|\, \mathcal{M}_1 \not\models F\}$$

and continue in similar fashion. In step i, either already $\mathbf{B}_i \models C$ (as in Fig. 1d, where $i = 3$) or there is an interpretation $\mathcal{M}_i$ such that $\mathcal{M}_i \models \mathbf{B}_i \cup \{\neg C\}$ (as in Fig. 1c, where $i = 2$). In such a case we set

$$\mathbf{A}_i = \{F \in \mathbf{A} \,|\, \mathcal{M}_i \not\models F\}$$

If $\mathbf{A}_i$ is empty, then $\mathcal{M}_i \models \mathbf{A} \cup \{\neg C\}$, hence C is not a theorem of $\mathbf{A}$. Otherwise, we set $\mathbf{B}_i = \mathbf{B}_{i-1} \cup \{F_{i-1}\}$, where F_i is an arbitrary formula chosen from $\mathbf{A}_i$.



Each point of the plane of the diagram represents a single possible interpretation of the language. A formula is represented as a shape that contains precisely the interpretations in which the formula is true. The conjecture C is illustrated by the gray rectangle. First, we construct a model $\mathcal{M}_0$ of $\neg C$ (a). We pick some premiss F_0 that avoids the model. Next, we construct a model $\mathcal{M}_1$ of $\{F_0, \neg C\}$ (b) and pick some other premiss F_1 which avoids this model. Then, because we still cannot prove C from $\{F_0, F_1\}$, we construct a model $\mathcal{M}_2$ of $\{F_0, F_1, \neg C\}$ (c) and pick yet another premiss F_2 that avoids it. Now, there is no model of $\{F_0, F_1, F_2, \neg C\}$ (d) and hence we can prove C from $\{F_0, F_1, F_2\}$.

Figure 1: Illustration of the semantic selection of premisses.

Observe that at each step $\mathbf{B}_i$ avoids all constructed interpretations $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$. Also note that $\mathbf{B}_i \cap \mathbf{A}_i = \emptyset$, since $\mathcal{M}_i \models \mathbf{B}_i$, but for each $F \in \mathbf{A}_i$ we know that $\mathcal{M}_i \not\models F$ by the definition of $\mathbf{A}_i$. Therefore, at each step, a different formula is moved from $\mathbf{A}_i$ into $\mathbf{B}_i$ (and possibly other formulae are removed from $\mathbf{A}_i$) and $|\mathbf{B}_i| = i$. And since $\mathbf{A}$ is finite, either $\mathbf{B}_k \models C$ for some k, or $\mathbf{A}_l = \emptyset$ for some l and thus $\mathbf{A} \not\models C$.

2.3 Adaptation for automated theorem provers

Automated theorem provers, model finders and other similar tools are always limited in available resources such as time or computer memory. Hence, if we are given a set of premisses $\mathbf{B}$ and a conjecture C, the result of a computation could be that we either

- find a proof of C from $\mathbf{B}$, or
- find an interpretation $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$, or
- run out of resources.

We have to take into account also the last possible outcome and modify the procedure described in the previous section to be able to deal with such a result.

The simplest possible solution is that if at step n we are neither able to prove C from $\mathbf{B}_n$ nor to find an interpretation $\mathcal{M}_n$ such that $\mathcal{M}_n \models \mathbf{B}_n \cup \{C\}$, we backtrack and at some previous step $k < n$ we try to pick a different formula F_i from $\mathbf{A}_k$.

Nevertheless, we shall now describe a more general solution that allows us to proceed in those cases when we are neither able to prove nor disprove C. The procedure remembers all failed attempts (sets of premisses) in a system of sets $\mathscr{F}$ to avoid trying the same attempt again and thereby getting into an infinite cycle. All interpretations that are constructed during the computation are stored in a set $\mathscr{M}$ and are then reused for selecting the premisses.

Let us give a schema of the procedure:

1. Initialize $\mathscr{F} = \emptyset$ and $\mathscr{M} = \emptyset$.
2. Repeat:
   (a) Construct a subset of premisses $\mathbf{B} \subseteq \mathbf{A}$ such that $\mathbf{B} \notin \mathscr{F}$ and such that for each $\mathcal{M} \in \mathscr{M} : \mathcal{M} \not\models \mathbf{B}$. If no such $\mathbf{B}$ can be found, report failure and exit.
   (b) Try to prove C from $\mathbf{B}$. If successful, print the proof and exit.
   (c) Otherwise, try to construct an interpretation $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$. In our case, this step is performed by an automated model finder like Paradox or Darwin.
   (d) If such a model $\mathcal{M}$ is found, then
      i. if $\mathcal{M} \models \mathbf{A}$ then report that $\mathbf{A} \not\models C$ and exit;
      ii. otherwise add $\mathcal{M}$ to $\mathscr{M}$, and
      iii. loop to 2.
   (e) If no such model is found, add $\mathbf{B}$ to $\mathscr{F}$ and loop to 2.

We only need to remember the failed attempts, because if we successfully construct a model $\mathcal{M} \models \mathbf{B}$, the condition $\mathcal{M} \not\models \mathbf{B}$ in 2a prevents choosing $\mathbf{B}$ again.

The procedure described in section 2.2 was a special case of this generalized procedure. As there were no failed attempts, the system $\mathscr{F}$ remained empty and in i-th cycle the set $\mathbf{B}$ constructed in 2a was the set $\mathbf{B}_i$.

As mentioned in step 2c, we use an external tool to construct the interpretations. There is no restriction on the constructed interpretations except that we are able to decide whether a formula is true in it or not. However, as far as we are aware, all current model finders are limited to constructing only finite interpretations.

The important part, which we have omitted until now, is the construction of the set of premisses $\mathbf{B}$ in 2a. We shall examine this problem in the next section and give a specification of a complete algorithm.

3 Specification of the algorithm

3.1 Converting the task to the weighted set cover problem

In the previous section we have outlined a schema of an algorithm for selecting premisses for proving a conjecture. In this section we shall construct a complete, generalized algorithm that tries to find a minimal set of premisses with respect to a given criterion. The criterion will be represented by a weight function on premisses and our goal will be to find a set of premisses for which the sum of the weights is minimized.

Definition 1 (Weight function)**.** *Let* $\mathbf{A}$ *be a set of premisses. A* weight function β *is a function that maps formulae from* $\mathbf{A}$ *into positive real numbers:*

$$\beta : \mathbf{A} \to \mathbb{R}^+$$

We shall now focus on the construction of the optimal set of premisses in step 2a of the procedure given in section 2.3. Our aim is to find such a subset of premisses $\mathbf{B} \subseteq \mathbf{A}$ that avoids all known interpretations of $\neg C$ and for which the sum $\sum_{B \in \mathbf{B}} \beta(B)$ is minimized.

For each premiss $B \in \mathbf{A}$ let $\mathscr{C}_B$ be the set of interpretations that B avoids:

$$\mathscr{C}_B = \{\mathcal{M} \in \mathscr{M} \mid \mathcal{M} \not\models B\} \tag{5}$$

Thus, $\mathbf{B}$ avoids all the interpretations in $\mathscr{M}$ if and only if $\bigcup_{B \in \mathbf{B}} \mathscr{C}_B = \mathscr{M}$. By this assignment we have converted our problem to the well-known *weighted set cover problem*:

Definition 2 (The weighted set cover problem)**.** *For the input*

- *ground elements* $\mathscr{M} = \mathcal{M}_1, \ldots, \mathcal{M}_n$
- *subsets* $\mathscr{C}_B \subseteq \mathscr{M}$ *where* $B \in \mathbf{A}$
- *weights* $\beta(B)$ *defined for* $B \in \mathbf{A}$

find a set $\mathbf{B} \subseteq \mathbf{A}$ *that minimizes* $\sum_{B \in \mathbf{B}} \beta(B)$ *such that* $\bigcup_{B \in \mathbf{B}} \mathscr{C}_B = \mathscr{M}$*. For the unweighted set cover problem we take* $\beta(B) = 1$ *for all* B*s.*

In our case, the elements being covered are the interpretations we have constructed so far and the covering sets are defined by (5).

There are many theoretical results regarding this problem in literature. For us the most important results concern the greedy algorithm, which approximates the problem:

Definition 3 (Greedy algorithm for the weighted set cover problem)**.** *The greedy algorithm for set cover selects sets according to the rule:*

> *At each stage, select the set that minimizes the cost (with respect to the weight function β) per additional element covered.*

More formally, if $\mathscr{N}$ is the set of uncovered ground elements, select B for which $\frac{\beta(B)}{|\mathscr{C}_B \cap \mathscr{N}|}$ is minimal.

For the unweighted set cover problem this means that we select the set for which $|\mathscr{C}_B \cap \mathscr{N}|$ is maximal, that is the set which contains the largest number of uncovered elements.

By the following results, the algorithm gives the best possible polynomial approximation of the weighted set cover problem:

Remark 1 (Properties of the greedy algorithm)**.** *The greedy algorithm for the weighted set cover problem has the following properties:*

- *The weighted set cover problem is NP-complete (see [Kar72]).*
- *The approximation ratio*[2] *of the greedy algorithm is $H(\max_{B \in \mathbf{B}} |\mathscr{C}_B|)$, where $H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq \ln(n) + 1$ (see [Chv79]).*
- *This is essentially the best possible approximation (see [Fei98]).*

3.2 Searching for sets of premisses using the greedy set cover algorithm

The greedy algorithm described in the previous section will find an approximation of the optimal subset of premisses. If we are able to prove C from $\mathbf{B}$, we are finished, and if we are able to find a model of $\mathbf{B} \cup \{\neg C\}$ we use the greedy algorithm to find a new optimal subset. The problem arises if we are neither able to prove C or find a new interpretation. Therefore, we slightly modify the greedy algorithm so that we are able to backtrack and find another next-to-optimal set of premisses that covers all the interpretations. Hereby we are able to construct a sequence of sets of premisses until we prove C or find a new interpretation.

The work of the modified algorithm can be viewed as traversing a search tree, where each node of the tree corresponds to a subset of premisses selected from $\mathbf{A}$. The original

[2] An algorithm Γ is an α-approximation algorithm for a minimization problem Π if

- Γ runs in polynomial time.
- Γ always produces a solution which is within a factor of α of the value of the solution of the optimal algorithm Π.

The least α that satisfies these conditions is called *the approximation ratio of* Γ.

greedy algorithm traverses only a single branch of the search tree until it finds a covering set. The modified algorithm remembers its position in the search tree and if it is neither able to prove the conjecture from the premisses nor to find a new interpretation, the algorithm continues traversing the tree according to the greedy rule. If it exhausts the whole tree, it exits with a failure.

Remark 2. *The theoretical results described in the previous section cover only the original greedy algorithm. The question whether the subsequent covering sets returned by this extended algorithm also well approximate the optimum is still to be researched. However, we believe that it a reasonable assumption.*

The algorithm in pseudo-code. Let us now look at the schema of the algorithm. The main function searchForProofOrModel recursively searches the sub-tree that corresponds to a given set of premisses $\mathbf{B}$. It sorts the remaining admissible premisses $B_1, \ldots, B_k$ by their cost per additional covered element (with respect to the weight function β) and recursively processes the sets $\mathbf{B} \cup \{B_i\}$ until either a model is found or the whole sub-tree is exhausted (the original greedy algorithm always picks $\mathbf{B} \cup \{B_1\}$). The function returns either a new model, or a **NIL** value if it exhausts all nodes of the sub-tree.

The traversed nodes are remembered using a global set variable $\mathscr{F}$. It is constructed so that the set $\{Y \subseteq \mathbf{A} \mid \textit{there is } X \in \mathscr{F} \textit{ such that } X \subseteq Y\}$ is the set of all visited nodes of the tree. Every time we visit a whole sub-tree, we replace all the visited sets of premisses occurring in the sub-tree by the set of premisses corresponding to the root of the sub-tree (line 40). This allows us to keep $\mathscr{F}$ small and thus to efficiently recognize the sets of premisses we have already inspected.

```
1  /* first, try to find any model of ¬C */
2  try to prove C from ∅; if successful, print ⊨ C and exit;
3  try to find a model M ⊨ ¬C;
4    if successful, set 𝒜 := {M};
5    otherwise exit "Failed to prove nor disprove C from A";
6  /* now we have at least one element to cover */
7  label ADD_MODEL:
8    for B ∈ A do
9      𝒞_B := {M ∈ 𝒜 : M ⊭ B}; /* initialize the covering sets 𝒞_B */
10   ℱ := ∅; /* initialize the set of failed attempts */
11   M := searchForProofOrModel(∅); /* search for a new model */
12   if M ≠ NIL then /* if a new model is found, search again */
13     if M ⊨ A then
14       exit "C not provable from A"
15     /* invariant: M ∉ 𝒜, see Corollary 1 that follows */
16     𝒜 := 𝒜 ∪ {M};
17     goto ADD_MODEL;
18   /* otherwise exit with failure */
19 exit "Failed to prove nor disprove C from A";

20 function searchForProofOrModel(set of premisses B)
21   returns new model or NIL
22 begin
23   𝒩 := 𝒜 \ ∪_{B∈B} 𝒞_B; /* the set of uncovered models */
```

24 let $B_1, \ldots, B_k$ be the premisses from $\mathbf{A}$ for which $(\mathscr{C}_{B_i} \cap \mathscr{N}) \neq \emptyset$
25 sorted by $\frac{\beta(B)}{|\mathscr{C}_{B_i} \cap \mathscr{N}|}$ in ascending order;
26 **for** $i := 1$ **to** k **do**
27 **begin**
28 $\mathbf{B}' := \mathbf{B} \cup \{B_i\}$;
29 **if** $\neg((\exists \mathbf{X} \in \mathscr{F}) : \mathbf{X} \subseteq \mathbf{B}')$ **then**
30 $\mathscr{N}' := \mathscr{N} \setminus \mathscr{C}_{B_i}$;
31 **if** $\mathscr{N}' = \emptyset$ **then**
32 try to prove C from $\mathbf{B}'$; if successful, **print** $\mathbf{B}' \models C$ and **exit**;
33 try to find a model $\mathcal{M} \models \mathbf{B}' \cup \{\neg C\}$; if successful, **return** $\mathcal{M}$;
34 **else** /* $\mathscr{N}' \neq \emptyset$, *i.e. there are still uncovered models* */
35 $\mathcal{M}$:= searchForProofOrModel($\mathbf{B}'$); /* *recursively try to find a model* */
36 **if** $\mathcal{M} \neq$ **NIL then**
37 **return** $\mathcal{M}$;
38 /* *whether* $\mathscr{N}' = \emptyset$ *or not, remember the visited sub−tree:* */
39 /* *replace all descendants of* $\mathbf{B}'$ *by* $\mathbf{B}'$ */
40 $\mathscr{F} := \{\mathbf{B}'\} \cup (\mathscr{F} \setminus \{\mathbf{X} \in \mathscr{F} \mid \mathbf{B}' \subseteq \mathbf{X}\})$;
41 **end**;
42 /* *exhausted all possibilities, neither proof nor new model was found* */
43 **return NIL**;
44 **end**;

3.2.1 Constructed interpretations and termination of the algorithm

Lemma 1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two interpretations constructed by the algorithm, $\mathcal{M}_2$ after $\mathcal{M}_1$. Let $\mathbf{B}_1$ and $\mathbf{B}_2$ be their corresponding sets of premisses $\mathbf{B}'$ from which the interpretations were constructed at line 33 ($\mathcal{M}_i \models \mathbf{B}_i \cup \{\neg C\}$). Then $\mathbf{B}_1 \neq \mathbf{B}_2$ and there is a formula F such that $\mathcal{M}_2 \models F$ but $\mathcal{M}_1 \not\models F$.*

Proof. $\mathbf{B}_2$ is selected so that for every $\mathcal{M} \in \mathscr{M}$ at that point $\mathcal{M} \not\models \mathbf{B}_2$. In particular $\mathcal{M}_1 \not\models \mathbf{B}_2$, hence there is $F \in \mathbf{B}_2$ such that $\mathcal{M}_1 \not\models F$. But $\mathcal{M}_2 \models F$ (because $\mathcal{M}_2 \models \mathbf{B}_2$), hence F has the desired properties. And because $\mathcal{M}_1 \models \mathbf{B}_1$, $F \notin \mathbf{B}_1$, thus $\mathbf{B}_1 \neq \mathbf{B}_2$. □

Corollary 1. *No two constructed interpretations are isomorphic.*

Theorem 1. *The algorithm terminates.*

Proof. By lemma 1, every time the program constructs an interpretation it uses a different subset of premisses. There only $2^{|\mathbf{A}|}$ possible subsets of $\mathbf{A}$, hence the algorithm can construct at most $2^{|\mathbf{A}|}$ interpretations and must eventually terminate. □

4 Implementation

We have designed an implementation of the algorithm that is aimed to prove or disprove several conjectures at once. The program is written in the Java programming language and is connected to E prover ver. 0.99 [Sch02, Sch07] and two model finders: Darwin ver. 1.3FM [FBT07, BFdNT06] and Paradox ver. 2.0 [CS07, CS03]. When constructing models the program runs both these model finders simultaneously and takes the result

of the first one that completes. As the model finders take quite different approaches, this arrangement leads to a better success rate, and on multi-processor machines also to shorter times.

4.1 Input of the program

The input of the program is a list of TPTP[3] files. The formulae in each file can be divided into these categories:

Conjecture (denoted by C). The formula which we want to prove or disprove from the premisses given in the same file. (TPTP name `conjecture`.)

Premisses (denoted by $\mathbf{A}_C$). The set of formulae from which the algorithm should select the optimal set of premisses for (dis)proving the conjecture C. (TPTP name `axiom`.)

Definitions (denoted by $\mathbf{D}_C$). The formulae which are always included as premisses for proving C. In some cases the user may surmise that some formulae are indispensable for the proof, for example definitions of essential functional or predicate symbols. (TPTP name `definition`.)

4.2 Execution of the program

Recall that we denote by $\mathbf{C}$ the set of all conjectures the program is asked to prove, by $\mathbf{A}_C$ the input set of premisses for proving each conjecture $C \in \mathbf{C}$, by $\mathbf{D}_C$ the definitions given for proving C and finally by $\mathscr{M}$ the set of constructed interpretations.

At the beginning $\mathscr{M} = \emptyset$. The program runs in a loop until either all conjectures are (dis)proved or until it runs out of time. During each pass of the loop the program performs the following step:

1. Decide which conjecture C to try to (dis)prove at this step (we shall describe this in more detail later).
2. Use the greedy algorithm to select the optimal set of premisses $\mathbf{B} \subseteq \mathbf{A}_C$ that covers all interpretations from $\{\mathcal{M} \in \mathscr{M} : \mathcal{M} \models \neg C\}$.
3. Run the model finder and the prover on $\mathbf{B} \cup \{\neg C\}$.
   - If a model $\mathcal{M}$ is found, set $\mathscr{M} := \mathscr{M} \cup \{\mathcal{M}\}$;
   - or if $\mathbf{B} \models C$, report the achievement and remove C from further processing ($\mathbf{C} := \mathbf{C} \setminus \{C\}$);
   - or if neither a model is found nor C proved (we shall call this outcome *a failed attempt*), save the state of the greedy algorithm to be able to restore it next time C is selected and to construct another possible covering set.
4. In all cases, loop again to 1.

[3] Thousands of Problems for Theorem Provers, see [SS98].

Selecting the conjecture. When designing the way how to select the next conjecture to be processed, we had two primary requirements:

1. The program should not spend too much time trying to prove a single, possibly too hard conjecture. Instead, it should alternate between the given conjectures.
2. The program should favour the easier conjectures because it is likely it will be able to prove them earlier.

Here too we can take an advantage of the information stored in the constructed interpretations. Both above requirements are satisfied by assigning the following weight w_C to each conjecture and selecting the one with the highest weight:

$$w_C = |\{\mathcal{M} \in \mathscr{M} \mid \mathcal{M} \models C\}| - |\mathbf{C}| \cdot u_C \quad (6)$$

$$u_C = \text{the number of previous failed attempts when } C \text{ was selected.} \quad (7)$$

The more interpretations C is valid in, the more general C is likely to be and thus supposedly easier to prove. And each time C is selected we construct an interpretation $\mathcal{M}$ such that $\mathcal{M} \models \neg C$, hence w_C does not increase. On the other hand, if another conjecture D is true in $\mathcal{M}$, w_D increases. This happens for example if $D \in \mathbf{A}_C$ and it is selected as a premiss for proving C. Subtracting the number of failed attempts (multiplied by the number of conjectures for better efficiency) prevents the program from sticking to a single conjecture in the case when it is unable to construct a new interpretation for a long time and therefore the weights do not change.

4.3 Optimizations

4.3.1 Large number of constructed interpretations

One of the main drawbacks of the algorithm is that when it constructs a new interpretation $\mathcal{M} \models \mathbf{B} \cup \{\neg C\}$, only a small subset $\mathbf{B}$ of all possible premisses $\mathbf{A}_C$ is true in $\mathcal{M}$. Most of the other premisses from $\mathbf{A} \setminus \mathbf{B}$ are usually false in $\mathcal{M}$ and therefore they cover $\mathcal{M}$. High number of possible coverings then leads to construction of a large number of interpretations. One possible solution is to add some of the remaining premisses and/or conjectures $(\mathbf{A} \setminus \mathbf{B}) \cup (\mathbf{C} \setminus \{C\})$ to $\mathbf{B}$ in such a way that they will not affect the possibility and the difficulty of constructing $\mathcal{M}$. In particular, if $A \in (\mathbf{A} \setminus \mathbf{B}) \cup (\mathbf{C} \setminus \{C\})$ has no common predicate symbol with any of the formulae in $\mathbf{B} \cup \{\neg C\}$, we can safely add A to $\mathbf{B}$. Depending on the structure of the problem, we can enlarge $\mathbf{B}$ by several formulae and thus reduce the number of possible coverings of $\mathcal{M}$, especially at the beginning of the process when the sets of selected premisses are small. On the other hand, in some cases this optimization can considerably slow down the model finder.

4.3.2 Formulae problematic for the model finder

Another question that highly affects the efficiency of the algorithm is the ability of the model finder to construct an interpretation. It is not surprising that some formulae can make the process of searching for models much more difficult. By assigning higher weights β to such problematic formulae we can discourage the algorithm from selecting such formulae.

For example, the model finder Darwin is not optimized for equality reasoning and therefore we primarily wanted to avoid formulae with equality. Secondarily, we wanted to assign higher weights to formulae with many variables, because such formulae also make the process of model construction more difficult (this was suggested by Koen Claessen, the author of Paradox). Therefore, we assigned the weight β of a formula A as

$$\begin{array}{lll} \beta(A) = & \theta(A) & \text{if } A \text{ does not contain '='} \\ \beta(A) = & \theta(A) + 10000 & \text{if } A \text{ contains '='} \end{array} \tag{8}$$

where $\theta(A)$ is defined recursively as

$$\begin{array}{rll} \theta(P) = & 0 & \text{if } P \text{ is an atom} \\ \theta((\forall x)F) = & \theta(F) + 1 & \\ \theta((\exists x)F) = & \theta(F) + 1 & \\ \theta(\neg F) = & \theta(F) & \\ \theta(F\Omega G) = & \max(\theta(F), \theta(G)) & \text{where } \Omega \text{ is any binary connective} \end{array} \tag{9}$$

$\theta(A)$ is simply the maximum number of quantifiers we can encounter when traversing from an atom of A to A's topmost connective. Such proper assignment of weights can indeed reduce the unwanted outcomes when the model finder is not able to find an interpretation.

We also tried to set the weight to $\beta(A) = 2^{\theta(A)}$ (resp. $\beta(A) = 2^{\theta(A)} + 1000$) as this number more closely corresponds to the number of possible variable assignments that must to be examined when deciding the validity of A. This weight also produced very good results.

There are of course many other possibilities how to assign the weights, depending on the nature of the particular problem. For example, when deciding relationships between modal logic systems in [Pud06b] we based the weights on the number of modal symbols occurring in the formula.

5 Empirical evaluation

For the first time we have used the algorithm when deciding relationships between modal systems in [Pud06b]. The first implementation of the algorithm allowed us to decide several cases that we were not able to solve with an ordinary theorem prover. The recent improved version of the program has not yet been evaluated on a large sets of problems. However, the present experiments already show that the method is quite promising.

5.1 Tests on the bushy division of The MPTP Challenge

We have tested the program on the bushy division The MPTP Challenge [US06]. The challenge is focused on automated theorem proving in environments with many axioms, predicates and functors. For the test we have selected 121 problems on which E prover spent more than 10s (or failed at all) in the MPTP referential tests.

All experiments were run on a Linux machine with 4 Dual Core AMD Opteron^{TM} 1.8 GHz Processors and 5 GB of memory. The program was able to take advantage of the multiple processors by running the prover and the model finders simultaneously.

article	thm. name	E proof CPU time [s]	proved after [s]	proof time [s]	# of interpre-tations	# of initial prems.	# of selected prems.	# of failed proof attempts
finset_1	t17		87	0.20	4860	87	32	392
funct_1	t21		485	0.93	412	47	21	322
funct_1	t57		90	0.03	135	46	20	40
funct_1	t62		40	0.04	118	46	16	13
funct_1	t145		391	0.15	171	40	15	253
lattice3	t1	14				143		
lattice3	t3	9				152		
orders_2	t25	147				84		
ordinal1	t23		8	0.19	56	44	12	3
pre_topc	t48	1114	320	0.24	2074	71	17	2
relat_1	t12		76	0.02	160	36	14	93
relat_1	t25	24	210	0.37	266	34	13	149
relat_1	t44		276	0.04	249	34	15	351
relat_1	t45		651	0.04	284	34	15	1247
relat_1	t74		1728	0.26	466	38	17	2140
relat_1	t86		134	0.60	186	38	18	55
relat_1	t88	14	47	0.12	153	35	16	19
relat_1	t115		66	0.62	228	38	18	22
relat_1	t117	15	39	0.10	145	35	15	15
relat_1	t143		58	0.06	253	38	18	24
relat_1	t166		69	0.08	207	38	16	33
relat_1	t167		759	0.04	508	39	18	890
relat_1	t174	30	101	0.66	170	39	16	36
relat_1	t178		41	0.11	199	35	12	11
relset_1	t12	729				36		
subset_1	t43	89	3	0.18	29	26	14	0
waybell_7	t8	2				195		
waybell_9	t29	31				249		
wellord1	l29		12	0.03	93	36	15	3
wellord1	t19		86	0.58	161	54	25	62
wellord1	t21		9	0.48	89	41	12	0
wellord1	t24	31				35		
wellord2	t3		15	0.10	131	55	19	0
wellord2	t5		45	0.06	153	55	21	0
yellow_0	t42		472	0.07	3237	89	27	0
yellow_0	t60		790	0.05	4119	91	27	0
yellow_0	t61		771	0.09	2385	96	34	180
yellow_1	t2	16				183		
zfmisc_1	t99		3	0.60	11	13	7	0
zfmisc_1	t136	96				8		

Table 1: Results for 121 "difficult" problems from the bushy division of MPTP.

The results of the test are summarized in Table 1. The table shows only the problems solved by E or by the program (or by both). The third column shows the CPU time of standalone E. The fourth column shows the wall-clock time[4] of the program. The fifth column shows the CPU time of E on the premisses selected by the program. The sixth column shows the number of interpretations constructed by the program during the search. The last column shows the number of failed attempts, that is the number of cases where the program was neither able to construct a new model nor to prove the conjecture. If the program (or E) were not able to solve a problem, the corresponding columns are left blank.

The presented experiments were conducted with the CPU limit set to 1.2s for both the prover and the model finders. When have noticed that the program performs better when the time limits for the prover and for the model finder are very low.[5] This is explained by the observation that in vast majority of cases if a model cannot be found within a few seconds it is not found at all. And if an optimal (or near-optimal) set of premisses is eventually found, the prover spends very little time on proving the conjecture. Thus, if the time limits are higher, the program wastes much of its time on unsuccessful attempts and is not able to construct as many interpretations.

The program was able to prove 31 problems of 121 compared to 15 of standalone E. 6 problems were solved by both E and the program. The total time is mostly affected by the number of failed attempts and by the total number of constructed interpretations. As expected, the proofs from the selected subsets of premisses (fifth column) indeed took very little time, just fractions of seconds. E exhibits the usual behavior of proving tools that with increasing time it becomes overwhelmed by the number of derived clauses. In our case, E successfully proved only two conjectures after time longer than 150s. On the other hand, when the program was given enough time, it was able to prove many more difficult conjectures.

5.2 SRASS

The idea was recently implemented by Geoff Sutcliffe and Yury Puzis in SRASS system [SP07]. The system uses more advanced techniques to further improve performance, including a syntactic relevance measure as an ordering heuristic to guide the selection process. SRASS is able to solve many problems that can not be solved by the underlying theorem prover alone and is among the most successful systems entering The MPTP challenge [US06].

6 Conclusion and further work

The main difference from the commonly used proving techniques is that the algorithm is capable of selecting only those premisses that are necessary for the proof and that its decisions are based not on a syntactic but on a semantic basis.

[4]The program runs in a platform-independent language and moreover it spawns multiple child processes (the prover and the model finders), therefore we were not able to gather the total CPU time of the program.

[5]Of course this observation applies only to this particular set of problems.

The program was eventually able to find a proof for some conjectures that were too difficult for the classical black-box prover. When an optimal set of premisses was found, the proof constructed from this set usually took just a fraction of second. Thus, even if it takes the program a long time to find a proof, the optimal set of premisses provides the information how to prove the conjectures efficiently.

6.1 Advantages

6.1.1 Reusing interpretations

The interpretations constructed during the process of proving one conjecture can be reused by the algorithm for proving other conjectures, even from different theories. It is only required that the theories share the same language.

By accumulating such interpretations, the program gains knowledge about relationships between theorems and premisses within the theory. As the process is fully automated, the algorithm could be used to gather a large amount of such interpretations without any user intervention and these interpretations could be then used to answer queries in the future.[6] Therefore, we surmise that the algorithm could prove its worth for large databases of mathematical knowledge, both for automated and interactive proving of theorems.

6.1.2 Saving the state of the algorithm

The state of the algorithm can be easily saved and restored. The state is fully described by the set of unsuccessful attempts $\mathscr{F}$ and by the set of constructed interpretations $\mathscr{M}$. Hence, the algorithm can be easily suspended, if there is a more important task to be solved, and resumed later in the future. Or, like in our implementation, the program can alternate the conjecture it tries to prove according to some criterion.

6.1.3 More efficient on long runs than conventional provers

As we have noticed in our example, classical black-box provers usually become ineffective (and tend to consume a lot of memory) after a few minutes, when the number of inferred clauses becomes too large. The program does not suffer from this issue, as it only needs to remember the constructed interpretations[7] and the set of failed attempts.

6.2 Disadvantages

6.2.1 A high number of constructed interpretations

A major disadvantage of the algorithm is that the number of constructed interpretations can become quite high, especially if there are a lot of similar axioms and lemmas in the set of premisses. Although the program was able to handle even several tens of thousands

[6] This idea was first proposed by Jiří Vyskočil in a personal discussion.

[7] In fact, it is even not necessary to remember the interpretations. It is sufficient to store the information of what premisses from $\mathbf{A}_C$ are true in each interpretation, that is just $|\mathscr{M}| \cdot |\mathbf{A}_C|$ Boolean values.

of interpretations, we believe that reducing the number would make the method much more efficient. We suggest the following improvements:

1. Focus on constructing interpretations in which a high number of the premisses is true. The higher the number, the less is the number of the remaining premisses that can be used to avoid the interpretation. Thus, the algorithm would be led more directly towards finding the optimal set of premisses.

2. Try to construct interpretations in which several of the conjectures are false. A single interpretation can be used to guide the search for all conjectures which are false in the interpretation.

   We have already partially implemented these two suggestions as described in section 4.3.1.

3. If possible, analyze what interpretations became unnecessary and remove them from the pool. As we have seen, the algorithm never constructs two isomorphic interpretations (Corollary 1). Therefore, every constructed interpretation carries a unique piece of knowledge. It will be necessary to devise a more sophisticated technique for detecting which interpretations participate only negligibly on the selection of premisses and thus can be discarded.

6.2.2 Limitation to finite interpretations

In general, the problem of constructing a model of a set of formulae is algorithmically undecidable. Any algorithm we construct will be only able to construct some specific class of interpretations. In most cases, it will be the class of finite interpretations. And as many interesting theories (like arithmetic or set theory) have infinite interpretations, this may impose a strict restriction on the method. However, the situation needs not to be as hopeless as it seems to be. Recall that we only construct interpretations of subsets of the given axioms. If we force the algorithm to elude those axioms that cause the interpretation to be infinite, we may still be able to achieve good results. This can be easily arranged by assigning high weights to those problematic axioms. However, we are not aware how to automatically identify such axioms.

Another solution would be to construct and represent some class of infinite interpretations. For example, one could construct Herbrand interpretations, saturations or more generally represent the interpretations as complex functions or programs that compute the validity of formulae.

6.3 Comparison with syntactic techniques

As it turns out, the algorithm falls short of the original expectation that it could very effective when the number of premisses is very large (hundreds or more). The reason is the high number of interpretations that need to be constructed to determine the optimal set of premisses in such cases. Syntactic approaches as [Urb06a, Urb06b, RS98, MP06] seem to be much more efficient. However, the value of the algorithm emerges in the cases where even the number of premisses is moderate, the prover is overwhelmed by unnecessary premisses.

It seem quite possible that a combination of the semantic approach with syntactic techniques might be very efficient. This assumption is supported by the results of the SRASS system [SP07] in the MPTP Challenge [US06] (see also Section 5.2).

6.4 Further work

We have already suggested many possible areas of improvement in the previous section. We believe that considerable improvements could be attained by close cooperation with the designers of model finders and theorem provers to implement some of the following proposals:

1. Guide the model finder when constructing interpretations. We suggest that the model finder would be given two sets of formulae, **A** and **L**. It would be required to find a model $\mathcal{M} \models \mathbf{A}$ such that it is also a model of as much as possible formulae from **L** (for example within some given time limit, domain size, etc.).

2. Reveal which premisses are well suited for a particular theorem prover or a model finder. This information could be used to minimize the number of cases where neither a proof nor a new interpretation are found, and secondly to develop a better founded, automated process of assigning weights to premisses.

The program would also benefit from a graphical user interface that would facilitate visual and interactive analysis of the process.

Although the method is not mature yet and many aspects still need to be researched, we believe that it can bring significant benefit to the task of automated proving of theorems. Not only it can allow to prove conjectures that are hard to prove by conventional prover, but it also opens the possibility to further analyze the relationships between the conjectures, the premisses and the interpretations that it constructs.

References

[BFdNT06] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. Computing finite models by reduction to function-free clause logic. to appear, preliminary version, June 2006.

[Chv79] Vašek Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4:233–235, 1979.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[CS07] Koen Claessen and Niklas Sörensson. Paradox – a first-order logic model finder. WWW pages, 2007. `http://www.cs.chalmers.se/~koen/paradox/`.

[FBT07] Alexander Fuchs, Peter Baumgartner, and Cesare Tinelli. Darwin – a theorem prover for the model evolution calculus. WWW pages, 2007. `http://combination.cs.uiowa.edu/Darwin/`.

[Fei98] Uriel Feige. A threshold of $\ln(n)$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[Kar72] Richard Manning Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[MP06] Jia Meng and Lawrence Paulson. Lightweight relevance filtering for machine-generated resolution. volume 192 of *CEUR Workshop Proceedings*, 2006.

[Pud06a] Petr Pudlák. Search for faster and shorter proofs using machine generated lemmas. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 34–52, 2006.

[Pud06b] Petr Pudlák. *Verification of Mathematical Proofs*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2006. Available online at `http://lipa.ms.mff.cuni.cz/~pudlak/pp-thesis.pdf`.

[RS98] W. Reif and G. Schellhorn. Theorem proving in large theories. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III, 2. Kluwer Academic Publishers, Dordrecht, 1998.

[Sch02] S. Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[Sch07] Stephan Schulz. The E equational theorem prover. WWW pages, 2007. `http://www.eprover.org/`.

[SP07] Geoff Sutcliffe and Yury Puzis. SRASS – a semantic relevance axiom selection system, 2007. `http://www.cs.miami.edu/~tptp/ATPSystems/SRASS/`.

[SS98] G. Sutcliffe and C. B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Urb06a] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 4(1):414–427, 2006.

[Urb06b] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[US06] Josef Urban and Geoff Sutcliffe. The MPTP $100 Challenges, 2006. `http://www.cs.miami.edu/~tptp/MPTPChallenge/`.

MaLARea: a Metasystem for Automated Reasoning in Large Theories

Josef Urban
Dept. of Theoretical Computer Science
Charles University
Malostranske nam. 25, Praha, Czech Republic

Abstract

MaLARea (a Machine Learner for Automated Reasoning) is a simple metasystem iteratively combining deductive Automated Reasoning tools (now the E and the SPASS ATP systems) with a machine learning component (now the SNoW system used in the naive Bayesian learning mode). Its intended use is in large theories, i.e. on a large number of problems which in a consistent fashion use many axioms, lemmas, theorems, definitions and symbols. The system works in cycles of theorem proving followed by machine learning from successful proofs, using the learned information to prune the set of available axioms for the next theorem proving cycle. Although the metasystem is quite simple (ca. 1000 lines of Perl code), its design already now poses quite interesting questions about the nature of thinking, in particular, about how (and if and when) to combine learning from previous experience to attack difficult unsolved problems. The first version of MaLARea has been tested on the more difficult (chainy) division of the MPTP Challenge solving 142 problems out of 252, in comparison to E's 89 and SPASS' 81 solved problems. It also outperforms the SRASS metasystem, which also uses E and SPASS as components, and solves 126 problems.

1 Motivation and Introduction

In the recent years there has been a growing need for Automated Reasoning (AR) in Large Theories (ARLT). First, formal mathematical libraries created by proof assistants like Mizar, Isabelle, HOL, Coq, and others have been growing, often fed by important challenges in the field of formal mathematics (e.g. the Jordan Curve Theorem, the Kepler Conjecture, the Four Color Theorem, and the Prime Number Theorem). At least some of these proof assistants are using some automated deductive methods, and some of these libraries are (at least partially) translatable to first-order format [MP06b, Urb06] (like the TPTP language [SS98]) suitable for Automated Theorem Provers (ATPs). Also, when looking at the way how some of the hard mathematical problems like Fermat's last theorem and Poincare's conjecture were solved, it seems that difficult mathematical problems quite often necessitate the development of large (sometimes even seemingly unrelated) mathematical theories, which are eventually ingeniously combined to produce the required results. An attempt to create a set of notrivial large-theory mathematical problems as an initial benchmark for ARLT systems is the MPTP Challenge[1] (especially its chainy division), consisting of 252 problems (some of them with quite long human-written formal proofs) in a theory with 1234 formulas and 418 symbols which represents a part of the Mizar library.

[1] `http://www.cs.miami.edu/~tptp/MPTPChallenge/`

Today, there also seems to be a growing interest in using Automated Reasoning methods (and translation to TPTP) for sufficiently formal large "nonmathematical" knowledge bases, like SUMO [NP01] and CyC [MJWD06]. It seems that semantic ontologies and semantic tagging and processing e.g. for publications in natural and technical sciences are currently taking off, creating more and more applications for ARLT.

On the other hand, it seems to be quite a common thinking that the resolution-based ATP systems using usually some complete combination of paramodulation and resolution are easily overwhelmed when a large number of irrelevant axioms are added to the problems. Sometimes (actually surprisingly often in the AI and Formal Math communities) this is even used as an argument attempting to demonstrate the futility of using ATPs for anything "serious" in formal mathematics and large theories in general. There have been several answers to this problem (and to the general problem of the fast growing search space) from the ATP community so far. Better (i.e. less prolific while still complete) versions of the original calculi, and their combinations with even more efficient special-purpose decision procedures, have been a constant research topic in the ATP field. More recently, a number of heuristical approaches have appeared and been implemented. These methods include strategy scheduling (competitive or even cooperative, see e.g. [Sut01]), problem classification and learning of optimal strategies for classes of problems, lemmatization (i.e. restarting the problems only with a few most important lemmas found so far, see also [Pud06]), weakening [NW04] (solving a simpler problem, and using the solution to guide the original problem), etc.

The metasystem for ARLT which is described here falls into this second category of heuristical additions governing the basic ATP inference process. It is based on an assumption (often spelled-out by the critics of uniform ATP) that while working on problems in a particular domain, it is generally useful to have the knowledge of how previous problems were solved, and to be able to re-use that particular knowledge in a possibly nonuniform and even generally incomplete way. Again, this idea is not exactly new, and not only in the world of "very-AI-but-very-weak" experimental AR systems implemented in Prolog. At least the very efficient E prover [Sch02] has the optional capability to learn from previous proofs, and to re-use that knowledge for solving "similar" problems. In comparison to this advanced functionality of E, the first version of MaLARea is quite simple, and more suitable for easy experimenting with different systems. The machine learning is done by an external software, and one can imagine any reasonable learning system to take the place of the currently used SNoW [CCRR99] system. The features used for learning should be easy to extend, as well as the whole learning setting. It is quite easy to change the parameters of how and when particular subsystems are called, and to experiment with additional heuristics. In the next section we describe the structure of the metasystem in more detail, and then we show its current performance on the chainy division of the MPTP Challenge.

2 How MaLARea works

2.1 Basic idea

The basic idea of the metasystem is to interleave the ATP runs with learning on successful proofs, and to use the learned knowledge for limiting the set of axioms given to the ATPs in the following runs. In full generality, the goal of the learning could be stated as creating an association of some features (in the machine learning terminology) of the conjecture formulas (or even of the whole problems when speaking generally) with proving methods which turned out to be successful when those particular features were present. This general setting is at the moment mapped to reality in the following way: The features

characterizing formulas are just the symbols appearing in them. The "proving method" is just an ordering of all available axioms. So the goal of our learning is to have a function which when given a set of symbols produces an ordering of axioms, according to their expected relevancy with respect to that set of symbols. One might think of this as the particular set of symbols determining a particular sublanguage (and thus also a subtheory) of a large theory, and the corresponding ordering of all the available axioms as e.g. a frequency of their usage in a book written about that particular subtheory. There are certainly many ways how this setting can be generalized (more term structure and problem structure as features, considering ATP ordering and literal selection strategies as part of the "proving method", etc.). The pragmatic justification for doing things in the first version this way, is that while this setting is sufficiently simple to implement, it also turned out to be quite efficient in the first experiments with theorem proving over the whole translated Mizar library [Urb06, Urb04].

This central "deduce, learn from it, and loop" idea is now implemented using a simple (one might call it "learning-greedy") "growing axiom set" and "growing timelimit" policy. The main loop first tries to solve all the problems "cheaply", i.e., with the minimal allowed number of the most relevant axioms and in the lowest allowed timelimit. Each time there is a success (i.e. a new problem is solved), the learning is immediately performed on the newly available solution, and the axiom limit and timelimit drop to the minimal values (hoping that the learning has brought a new knowledge, which will make some other problem easily solvable). If there is no success with the minimal axiom and time limits, the system first tries with a doubled axiom limit until a certain maximal threshold (currently 128) on the number of axioms is reached. If there is still no success, the time limit is quadrupled, and the axiom limit drops to a small value again, etc. The system is now limited by the minimal and maximal values of the axiom and time limit, and also by a maximal number of iterations (currently 1000). That means that the system stops if either all the problems are solved (unlikely), or the maximal values of the limits are reached without any new solution, or the maximal number of iterations has been reached.

2.2 Detailed implementation

The first version of MaLARea is available at `http://lipa.ms.mff.cuni.cz/~urban/MaLARea/MaLARea0.1.tar.gz`. The distribution contains a main Perl script (TheoryLearner.pl), and a number of ATP and other tools used by the main script. These tools currently are:

- the `E` (version 0.99) prover, and the tools epclextract and eproof needed for getting a detailed TPTP proof from E
- the `SPASS` (version 2.2) prover [Wei01]
- the `SNoW` (version 3.0.3) learning system
- the `tptp4X` utility from the TPTPWorld distribution for transforming the TPTP format to other ATP formats (like DFG)
- the `GetSymbols` utility from the TPTPWorld distribution for extracting symbols and their arities from formulas written in the TPTP language

The set of theorem provers could be larger, the constraints that we use is that

- the provers should be reasonably efficient and reasonably orthogonal

- it has to be possible to run the prover in a "proof mode", getting from it the list of axioms which were actually used in the successful proof (this is necessary for the learning phase, and it is the main reason why Vampire [RV02] with its undocumented and hardly parsable proof output cannot be used)
- a future requirement might be that the provers should be easily parameterizable, making learning over their strategies possible (this is definitiely the case for E)

The input to the system is a set of files (problems in the FOF TPTP format, the system probably would not handle the CNF format now). The following two "large theory" assumptions are made about the input problems:

- that the names of formulas in the files are stable (i.e. one name always denotes the same formula in all files in which it appears)
- and that the symbols are stable (the same symbol in two files has the same intended meaning).

So far, the system has always worked with problems where each conjecture was assumed to be provable from its axioms (assumed SZS status [SZS04] Theorem), however it seems that having some possibly countersatisfiable problems would not hurt the system. For simplicity of the Perl processing, we require that each formula is written on just one line in the input file (this can be achieved by preprocessing with `tptp4X`), and that each file is named after its (exactly one) conjecture, by adding certain prefix and certain suffix (specified as parameters to the main script) to the conjecture's name. As an example (and the main target of the system so far), the 252 problems from the chainy division of the MPTP Challenge are included in the distribution.

Given a set of problems, the system starts by creating a main "specs" file, containing for every conjecture the names of the axioms which are available (in its problem file) for its proving. Similarly, the system creates (using the GetSymbols utility) a "refsyms" file, containing for each formula appearing in some problem the set of symbols appearing in it. These files are actually kept in memory (as hashes) during the whole processing. The formulas and the symbols are disjointly numbered. This later serves for communicating with the SNoW system, which assumes the features over which it learns to be represented as numbers.

One of the main data structures which the system maintains is a hash (`%gresults`) which for each conjecture (problem) keeps the list of all proof attempts conducted so far, and their results. The data which are exactly kept for one proof attempt are following:

```
[SZSStatus, NumberOfReferences, CPULimit,
 ListOfReferences, ListOfNeededReferences]
```

Where the `SZSStatus` tells the result of the proof attempt, `CPULimit` is the limit with which the proof attempt was run, `ListOfReferences` are the axioms used for that particular proof attempt, `NumberOfReferences` is their number, and in case of a successful proof (SZSStatus Theorem), the `ListOfNeededReferences` tells which of the axioms were actually used in the proof.

Before we start explaining how the ATP proof attempts and machine learnings are organized, we first note what is exactly meant by the term "running ATPs on a problem" in the rest of this paper, and what exactly is meant by machine learning in the current version of the system.

2.2.1 Running ATPs

All the available ATPs (now just E and SPASS) are run on a problem in order of their expected performance (that now means E first, SPASS second) with the given timelimit. They are run only in the fast "assurance mode", i.e. not with the slowdown caused by doing the additional bookkeeping necessary for printing the proof. As soon as one of the provers solves the given problem (this means that it determines that the problem's status is either Theorem or CounterSatisfiable), the "assurance mode" processing stops. If the result status is Theorem, the successful system is re-run in a "proof mode" with a timelimit of 300 seconds. The reason for raising the timelimit so much is that at this point we know that the system is capable of solving the problem, and we want to know the solution so that we can learn from it for solving the rest of the problems. This procedure could be in the future modified e.g. by quite standard strategy scheduling, i.e. running only those provers (with those strategies, and possibly appropriately modified timelimits) which seem (again from previous experience) to be most likely to succeed on the problem. Another useful extension (suggested by Geoff Sutcliffe) motivated by the experience with the SRASS system [SY07] would be addition of the Paradox [CS03] system to the chain of ATP systems, especially in the proof attempts when the number of axioms is decreased to minimum. That's because in these very incomplete specifications the frequency of the CounterSatisfiable result is quite high, and Paradox seems to be currently the strongest system for determining CounterSatisfiability. After the procedure stops (either successfully, or unsuccessfully, with all ATPs resulting in timeout), the result is recorded in the results hash (`%gresults`) in the format described above (that means in case of successful proof also recording the names of the axioms which were exactly needed for the proof). The format now does not keep the information about particular ATPs (and possibly particular strategies). This is not needed as long as some smarter machine learning for selection of ATPs and strategies is not done, however the information about which ATP solved which system can still be retrieved from the metasystem's standard output.

2.2.2 Machine Learning in MaLARea and its use for selection of axioms

As noted above, the SNoW system is used in the naive bayes mode for all learnings, mainly because of its speed and relatively good previous experience on the whole Mizar library (thousands of symbols and tens of thousands of formulas). After each ATP run on all (unsolved) problems, the information about all successful proofs found so far is collected in a format suitable for the SNoW system. As explained above, our goal is to learn the association of symbol sets to axiom orderings. This in practice means that one training example contains all the symbols of a solved conjecture, together with the names of axioms needed for its proof (the symbols are in the machine learning terminology the "input features", while the names of the axioms are the "output (or target) features", i.e. those features that a trained classifier will try to assign to a test example consisting of the input features). Then the classifier (a bayes network) is trained on this set of examples. This procedure is very fast with the SNoW system (seconds or less for the number of features and examples available in the MPTP Challenge problems). It is technically possible just to add the new examples to an existing bayes network trained on the results of the previous ATP runs, however until really large time-consuming learnings (this means really large theories like the whole Mizar library) are needed, this is not necessary. The trained classifier is then used to prune the axiom sets for the next runs. It means that we take all the unsolved conjectures, and create a testing example from each by taking all its symbols. The classifier is run on this set of test examples, printing for each example the

list of target features (axiom names) in order of their likelihood to be useful (as judged from the training examples, i.e. previous proofs). This order of axioms is then used to select the required number of axioms for the next run on the unsolved problems. I.e., provided that the next run will limit the number of axioms to n, we are looking for each problem for n axioms from the problem's specification whose ranking by the classifier is highest.

2.2.3 Initial proof attempts in MaLARea

Before the system enters the main loop it first does two special proving attempts, in order to generate the "initial knowledge". The motivation for both of them is quite heuristical, and both have been subject to experimenting.

The first proof attempt is quite expensive: it tries to solve all the problems in their original form (that means without restricting the axioms in any way) with the maximum timelimit (now 64 seconds). The motivation for these settings is to allow the ATPs to consider the full axiom space for each problem, giving a chance to ATPs internal "large problem" strategies. The heuristical justification for the high timelmit is that any proof found at this point is a proof considering all the available axioms, which is not true for the rest of the proof attempts done by the metasystem. Therefore the information obtained from this proof attempt is in a certain way fresh and unbiased by solutions already found. It can be compared to the mode of work of some mathematicians, who when entering a new field, first try to think about the field themselves, possibly creating fresh insights, and only after that consult other experts in the field and the available literature. It should be noted that this approach probably would not be feasible for an order of magnitude larger theories (e.g. for the ca. 40000 theorems and definitions in the whole Mizar library). On the other hand, there seems to have been progress in this capability of ATP systems in the recent years: several years ago one might have claimed that already 1000 axiom gives to any resolution-based ATP system no hope.

The second proof attempt is cheap: it uses the minimal timelimit (1 second), and a purely symbol based similarity measure to cut the number of axioms to the maximal axiom limit (now 128). The symbol based measure is now just for simplicity achieved again through the SNoW's bayesian classifier: For each formula (all axioms and conjectures) one training example for SNoW is created from the list of the formula's symbols, and from the formula's name (this can be explained by saying that each formula is useful for proving itself, or more precisely, for proving something with a similar set of symbols). The classifier is trained on these examples, and then evaluated on the symbols of each conjecture formula, providing for each conjecture the ordering of axioms according to their symbol overlap with the conjecture. A possible future extension could be to employ more elaborate similarity measures at this point, possibly also with a higher timelimit. One reason why we are not here as aggressive with the timelimit in comparison to the previous pass, is that the symbol based measure is taken into account in all the following passes, i.e. for all the learnings on successful proofs explained in Section 2.2.2, we also add the training examples saying that each formula can be proved by itself.

2.2.4 The main loop

After the two initial proving passes the system performs first learning (Section 2.2.2) on the successful proofs, and enters the main loop with the initial timelimit set to minimum and axiom limit set to maximum (this is quite arbitrary, it might as well be the minimal axiom limit). The main loop is now limited to a certain number (default 1000) of iterations (passes) (this is probably a bit redundant, but good for fast testing), and it will also stop

when the maximal time and axiom limits are reached without finding any new proof. Obviously, as is the case for the MPTP Challenge, it can also be stopped by a user or operating system after a certain overall timelimit (252 problems times 300 seconds for the Challenge, i.e. 21 hours). The loop starts by an ATP run (see Section 2.2.1) on all unsolved problems with the given axiom and time limits. Then the loop branches.

If no problem was solved by the ATPs, no new learning is done, the axiom limit is doubled (if it is smaller than the maximal axiom limit), and for each unsolved conjecture a new specification is created using the last learning results and the new axiom limit. In case the axiom limit is already maximal, we instead quadruple the timelimit, and set the axiom limit to the double of the minimal value (the minimal value seemed a bit too useless when running with higher timelimits). If both the axiom and time limit are maximal, we stop.

If a problem was solved during the last ATP run, learning immediately follows (in order to take advantage of the newly available knowledge). The time and axiom limits are reset to the minimal values (hoping that the new knowledge will allow us to solve some more problems quickly), and the loop continues.

2.2.5 Usage of previous results

As described above, the system keeps the data about all previous proof attempts. This is mainly used to avoid the proof attempts which do not make sense in the light of the previous results. The current implementation recognizes three such situations for a given unsolved problem:

- the suggested set of axioms is a subset of a previously tried set of axioms, whose result was CounterSatisfiable
- the suggested set of axioms is equal to a previously tried set of axioms, whose result was ResourceOut, and the suggested timelimit is less or equal to the timelimit of the previous attempt (note that this practically interprets ResourceOut as running out of time, which is the vast majority of cases especially with the low timelimits, however the implementation could be improved in this respect)
- all ATPs have previously unexpectedly failed (status Unknown) on the suggested set of axioms (regardless of timelimit)

Note that for checking the last two conditions it would be good to keep already now the detailed information about each ATP system's result, not just one summarized version for all ATPs as is being done right now. In the current implementation this is temporarily worked around by using the Unknown status only if it was the result of all (both) ATPs. If at least one system ended with the ResourceOut status, this is the status recorded in the result datastructure. Another practical problem is that the status Unknown is currently used also if an ATP does not obey the timelimit (specified to it as a parameter) with which it is run, and has to be killed by using the operating system's limit. Given the three policies described above, it seems better to use ResourceOut in such situations, which would make it possible to re-run the system with higher timelimit later. Fortunately, it happens only very rarely that both ATPs have to be killed by the operating system.

Especially the first condition is fulfilled quite often when the axiom limit is in its lower values (the minimum is four axioms). Since the goal of the system is to try as many reasonable axiom subsets as quickly as possible, we try to repair the "subsumed" axiom specifications by adding additional axioms (again according to their rating by the last learning) when the timelimit is minimal (1 second), and the additional check is thus

cheap. So in such cases, the advertised system's axiom limit is not observed (though the correct data are obviously kept in the results datastructure). It could be argued that this should also be done for the higher timelimits. This is quite hard to decide, and hopefully not much relevant to the overall performance of the metasystem. The current heuristical reason for not doing it, is that we are happy to "shake abundantly" the set of axioms when it is cheap (i.e. low timelimit), while we are trying to limit the higher timelimit runs only to the combinations of axioms which make most sense. On the other hand, since we grow the timelimit exponentially, it could be argued that the notion of cheapness applies to all but the highest timelimits.

2.2.6 More questions on MaLARea policies

The previous paragraph actually shows some of the hard (and interesting) heuristical choices which one has to make when experimenting even with such a simple kind of combined deductive/inductive reasoning system. Why do we (after the first two passes) "learn greedily", and always prefer learning and low timelimit to more ATP with higher timelimits? Would not the "tabula rasa mathematician" argument used for the initial expensive pass also justify a less greedy approach to learning (i.e. let the system learn something, but not all others' inventions at once, the ATPs might still come up with something relatively new)? Or wouldn't it pay to have even much more of the fast "reasonable axiom shaking" attempts instead of the later and expensive higher timelimit attempts? Why do we double the axiom size, and quadruple the timelimits, and why are their minima and maxima set to their current values? Some explanation of this is the experience with the (super)exponentially behaving ATPs that are used, however one might conjecture that the system should be relatively robust to small changes of these values and policies. Even more of such interesting questions are likely to appear if new components (lemmatization, weakening, conjecturing, defining, etc.) are added, and if the learning component becomes more sophisticated. Even now, simple as the whole setting is, it sometimes gives a strange impression of conducting a bit of exploratory Artificial Intelligence.

3 Results

As noted above, the system's main target so far has been the chainy division of the MPTP Challenge. This is a set of 252 related mathematical problems, translated by the MPTP system from the Mizar library. The conjectures of the problems are Mizar theorems, which were recursively needed for the Mizar proof of one half (one of two implications) of the general topological Bolzano-Weierstrass theorem. The whole problem set contains 1234 formulas and 418 symbols. Unlike in the "less AI" bushy division of the Challenge, where the goal is just to reprove the Mizar theorems from their explicit Mizar references (and some background formulas used implicitly by Mizar), the problems in the chainy division intentionally contain all the "previous knowledge" as axioms. This results in an average problem size of ca. 400 formulas. The Challenge allows an overall timelimit policy, i.e., instead of being forced to solve the problems one-at-a-time with a fixed timelimit of 300 seconds, it is allowed to use the overall timelimit of 21 hours in an arbitrary way for solving the problems.

The system was run on this set of problems in five differently parameterized instances, on a cluster of 3056MHz Pentium Xeons each with 1GB memory (the memory limit for all the ATP runs was always 800MB). Before these instances were run, E version 0.99 and SPASS version 2.2 were tested on the cluster in the standard 300 seconds timelimit setting. E has solved 89 problems, and SPASS has solved 81. This is quite similar to the MPTP

Challenge measurements[2] on Geoff Sutcliffe's cluster, which claim 36% (91) problems solved by E 0.99, and 31% (78) problems solved by SPASS 2.2 (the relative differences might be caused e.g. by different memory limits). The total number of problems solved by either E or SPASS is 104.

All the five instances of MaLARea shared the minimal timelimit set to 1 second, and the minimal axiom limit set to 4 axioms. The instances differed in the values for the maximal timelimit, and maximal axiom limit, which were as follows:

- 128_4s: maximal axiom limit set to 128, maximal timelimit to 4 seconds
- 128_16s: maximal axiom limit set to 128, maximal timelimit to 16 seconds
- 128_64s: maximal axiom limit set to 128, maximal timelimit to 64 seconds
- 64_4s: maximal axiom limit set to 64, maximal timelimit to 4 seconds
- 64_64s: maximal axiom limit set to 64, maximal timelimit to 64 seconds

The last instance unfortunately crashed (for unknown, probably cluster-related issues) after 18 hours. The 128_64s version was let to run even beyond the timelimit of 21 hours for a total of 30 hours (when it was stopped by the operating system), to see if there is any improvement in the later stages (which was not the case). The 4 second and 16 second instances have stoped themselves before the timelimit of 21 hours, because they reached their maximal axiom and time limits. The reason for running the very low (4 seconds) maximal timelimit instances was to find out how important is the long initial pass, and how the system performs in a "shallow thinking only" mode.

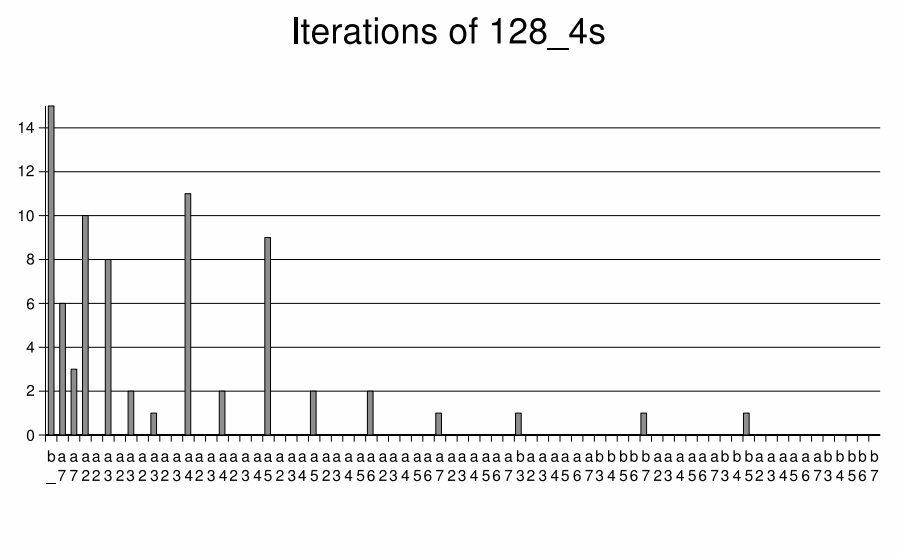
The following Table 1 summarizes the main results (the times are in minutes, last successful iteration is the last iteration in which a problem was solved). The Figures 1

description	solved	iterations	last successful iter.	time to stop	time to solve last
128_4s	131	73	62	300	270
128_16s	141	137	121	930	810
128_64s	142	127	108	1800	1160
64_4s	130	44	35	240	210
64_64s	136	77	62	1080	900

Table 1: Statistics for the five instances of MaLARea fighting the MPTP Challenge

and 2 show the iterations for all five instances and the gains in terms of solved problems. To make the scale readable on these figures, the timelimit is encoded as a letter (a,b,c,d), corresponding to the exponentially grown timelimits (1,4,16,64). The numbers (2,3,4,5,6,7) are the powers of 2 that should be used to get the axiom threshold (i.e. 4,8,16,32,64,128). The first pass in each figure uses an underscore instead of the threshold exponent, which means that the axioms were not limited in that pass. Instead of scaling the Y axis logarithmically, the value of the first most successful pass is cut on the figures, and given in their captions. Also note that the second and third passes are not the same, even though they have the same time and axiom limits. The second pass is the "symbol similarity only" pass, while the third one is the first in the main loop, i.e. the first which uses learning on previous successful proofs.

[2] `http://www.cs.miami.edu/~tptp/MPTPChallenge/Results/SVGResults.html`, `http://www.cs.miami.edu/~tptp/MPTPChallenge/Results/ChainyResults.data`



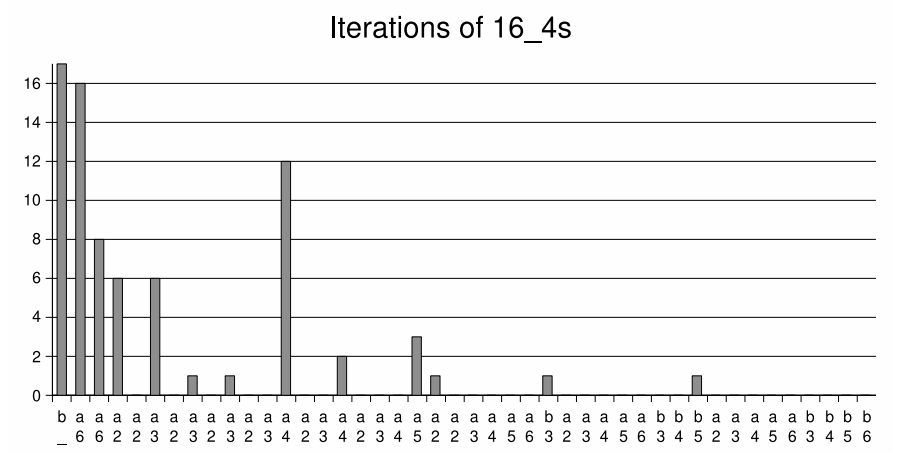


Figure 1: The first pass (cut) value for 128_4s is 71, and 72 for 64_4s

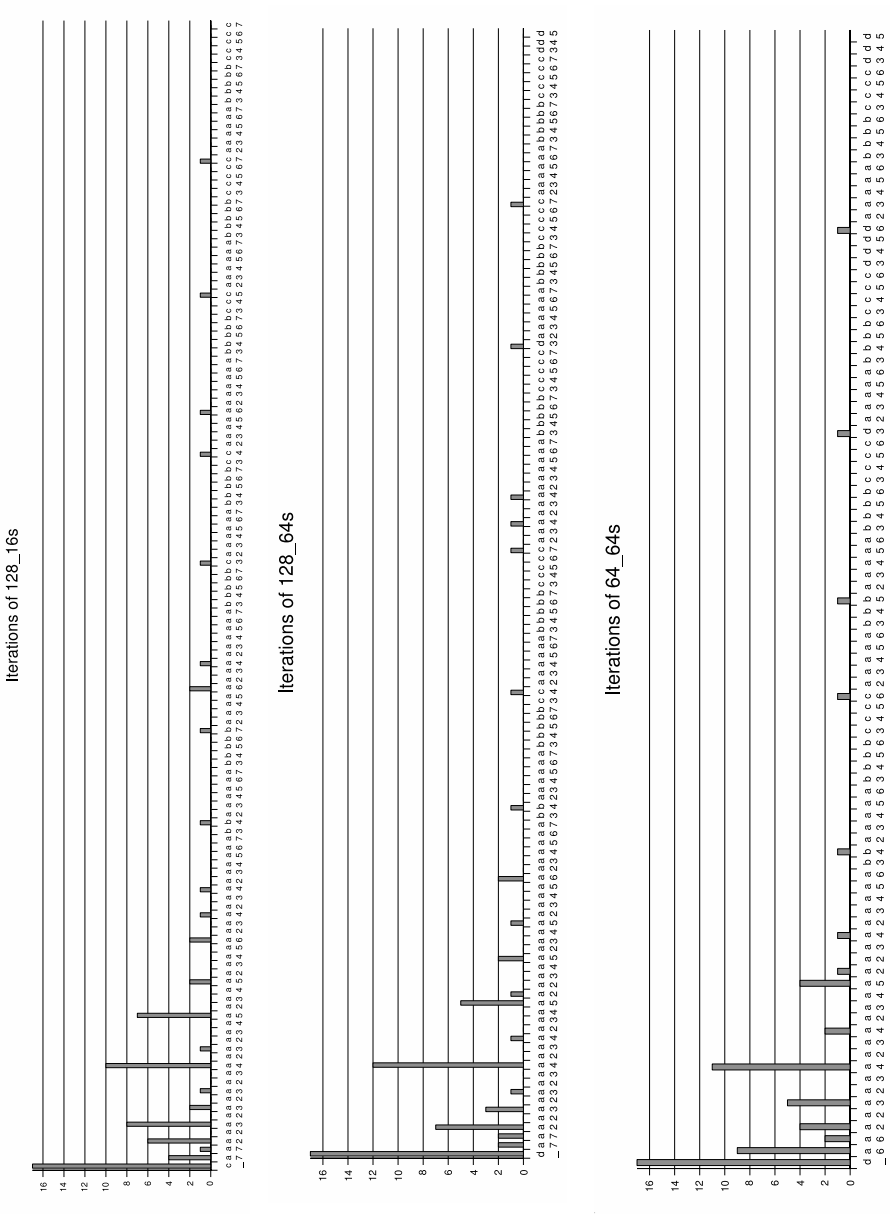


Figure 2: The first pass (cut) value for 128_16s is 85, 96 for 128_64s, and 93 for 64_64s

One easy observation made on the results, is that the combinations of time limit and axiom limit which have not been tried yet usually produce some new proofs. This could be explained by the fact that the relative gain from learning is in those situations much bigger (involving all the previously found solutions) than in the later runs, when only a few new solutions found in the meantime are used to modify the axiom relevancy. On the other hand, it is also interesting how sometimes a solution which was obtained in quite a difficult way and at quite late stage (e.g. the one b7 solution in 128_16s, and the one c7 solution in 128_64s) can make previously difficult problems quite easy to solve (the b7 solution is followed by two a6 and that in turn by one more a4 solutions, i.e. a series of solutions found in 1 second timelimit, similarly for the c7 solution in 128_64s). A bit closer analysis of some of the runs seems to suggest that using smarter learning could make this effect even more frequent. E.g. in some cases it seems that if the classifier knew more about the relationships between some symbols (e.g. that one is a predicate implying the other one, or that they are nearly equivalent predicates or functors), it could draw better analogies and give better advice.

4 Related Work

A very good overview of the field of "machine learning for automated reasoning" is given in the technical report[DFGS99]. The learning capability of E prover mentioned above is today probably the most sophisticated implementation existing in the field. There is quite a lot of related work on symbol-based and structure-based filtering of axioms, a recent one (done for the Isabelle system) is [MP06a], which also cites some more work.

Generally, it is a bit hard for the author to compare related (meta)systems with MaLARea. Quite often that would require further work on those systems, or their reimplementation, which could be criticized as "not being the original system". The point of creating the MPTP Challenge problems in the most standard FOL syntax available today (i.e. TPTP) is to allow everyone to test their system under very clear conditions, and report their results for comparison. It is currently also quite hard to test MaLARea on other than MPTP problems, since it is quite difficult to determine to what extent a given set of large theory problems satisfies the "large theory" criteria needed for MaLARea's machine learning, i.e., consistency of symbol and formula naming. This unfortunately seems to apply also to the set of Isabelle problems included in TPTP and used for evaluation in [MP06a].

5 Future Work and Conclusions

Although MaLARea's current performance is quite encouraging, it is still in a very early stage, and quite a lot of its possible extensions are mentioned above. The machine learning framework could be extended and improved, taking e.g. more relationships among the symbols (and other formula features) into account. Lemmatization could be also quite helpful, and while its addition should not be difficult, it would make the whole theory evolving, not static like so far. The same could be said about defining new useful notions, and possibly reformulating parts of the theory with them. Quite a strong method seems to be weakening, and its extreme version using completely instantiated models. A good database of models for a theory could be also used just as another simple way to classify formulas (adding more features to the learning). Shortly speaking, it seems that with rich theories the AI methods useful for Automated Reasoning can also get quite rich.

One thing that should be noted about the current version of the system is that it

does not use any Mizar-specific knowledge. There are two reasons for it. One is that the system is intended to be generally useful, not just Mizar-specific. The second reason is that re-using Mizar-specific knowledge requires some additional work on the system. But it is quite possible, that e.g having the standard MPTP algorithm for adding the background (e.g. Mizar type) formulas to the axiom set would sometimes be more useful than relying only on learning. Because particularly type hierarchies are quite likely to appear also in all kinds of non Mizar large theories, it would however be preferable to have a more general (quite likely heuristic, and possibly to some extent also governed by learned previous experience) methods for such "rounding-up" of axiom sets.

6 Acknowledgments

This work was supported by a Marie Curie International Fellowship within the 6^{th} European Community Framework Programme. The resources for the reproving experiments were provided by the Czech METACentrum supercomputing project.

References

[CCRR99] A. J. Carlson, C. M. Cumby, J. L. Rosen, and D. Roth. Snow user's guide. Technical Report UIUC-DCS-R-99-210, UIUC, 1999.

[CS03] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[DFGS99] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[MJWD06] C. Matuszek, Cabral J., M. Witbrock, and J. DeOliveira. An Introduction to the Syntax and Content of Cyc. In Baral C., editor, *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49, 2006.

[MP06a] Jia Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 53–69. CEUR, 2006.

[MP06b] Jia Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 70–80. CEUR, 2006.

[NP01] Ian Niles and Adam Pease. Towards a standard upper ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 2–9, New York, NY, USA, 2001. ACM Press.

[NW04] Monty Newborn and Zongyan Wang. Octopus: Combining learning and parallel search. *J. Autom. Reasoning*, 33(2):171–218, 2004.

[Pud06] Petr Pudlák. Search for faster and shorter proofs using machine generated lemmas. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 34–52. CEUR, 2006.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *Journal of AI Communications*, 15(2-3):91–110, 2002.

[Sch02] S. Schulz. E – a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Sut01] G. Sutcliffe. The Design and Implementation of a Compositional Competition-Cooperation Parallel ATP System. In H. de Nivelle and S. Schulz, editors, *Proceedings of the 2nd International Workshop on the Implementation of Logics*, number MPI-I-2001-2-006 in Max-Planck-Institut für Informatik, Research Report, pages 92–102, 2001.

[SY07] G. Sutcliffe and Puzis Y. SRASS - a semantic relevance axiom selection system. In Pfenning F., editor, *CADE 2007*, Lecture Notes in Artificial Intelligence. Springer, 2007. To appear.

[SZS04] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In V. Sorge and W. Zhang, editors, *Distributed and Multi-Agent Reasoning*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2004.

[Urb04] Josef Urban. MPTP - motivation, implementation, first experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

[Urb06] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

[Wei01] C. Weidenbach. *Handbook of Automated Reasoning*, volume II, chapter SPASS: Combining Superposition, Sorts and Splitting, pages 1965–2013. Elsevier and MIT Press, 2001.

Cyc Design Challenges and Solutions

Keith Goolsbey
Cycorp Inc.
3721 Executive Center Drive
Suite 100, Austin, TX 78731

Abstract

Cyc comprises a large, contextualized, common sense knowledge base (KB) which is encoded in an expressive representation language (essentially FOL with a few key extensions) and paired with an inference engine optimized for the classes of queries we most frequently encounter. These queries tend to mix relatively shallow reasoning within one of a large number of idiosyncratic subtheories with relatively deep reasoning within one of a very small number of stylized subtheories. The constraints of these queries in a large and expressive KB combined with the need to efficiently react to KB elaboration together provide a unique set of design challenges that are extremely stressful for the solutions provided by the current state of the art FOL theorem provers. The solutions to these challenges currently adopted by the Cyc inference engine will be presented within the context of a new suite of TPTP problems that are derived from Cyc's KB and typical queries and are intended to demonstrate Cyc's design challenges for investigation by the wider community.

First Order Reasoning on a Large Ontology

Adam Pease[1], Geoff Sutcliffe[2]
[1]Articulate Software
apease[at]articulatesoftware.com
[2]University of Miami
geoff[at]cs.miami.edu

Abstract

We present results of our work on using first order theorem proving to reason over a large ontology (the Suggested Upper Merged Ontology – SUMO), and methods for making SUMO suitable for first order theorem proving. We describe the methods for translating into standard first order format, as well as optimizations that are intended to improve inference performance. We also describe our work in translating SUMO from its native SUO-KIF language into TPTP format.

1. Introduction

There are two main areas of effort in this work. The first is to take a language that appears to be beyond first order, and translate it into the strict first order form needed for standard first order theorem provers. The second is in developing techniques that allow standard provers to perform well on reasoning problems on a large ontology. Most first order theorem provers, particularly those whose development has been done using the TPTP (Sutcliffe & Suttner, 1998) library for testing, have been optimized to perform well on proofs that require deep reasoning on a very small number of axioms, on the order of 10, or on proofs with a small number of rules but very large numbers of ground facts. Reasoning over a large ontology such as SUMO requires a spectrum of reasoning, from simple matching and unification to deep multi-step proofs, but most typically has a key problem of finding a small number of relevant axioms in a sea of irrelevant ones. There are also certain axioms that are needed much more frequently than others. Current ATP systems are not tuned to cope with these two distinctive aspects of reasoning over a large ontology. The most general way of framing a solution is to trade space for time, caching what are anticipated to be frequently used results.

The Suggested Upper Merged Ontology (SUMO) (Niles & Pease, 2001) is a free, formal ontology of about 1000 terms and 4000 definitional statements. It is provided in the SUO-KIF language (Pease, 2003), which is a first order logic with some second-order extensions, and also translated into the OWL semantic web language (which is a necessarily lossy translation, given the limited expressiveness of OWL). SUMO has also been extended with a number of domain ontologies, which together number some 20,000 terms and 70,000 axioms. SUMO has been mapped to the WordNet lexicon (Fellbaum, 1998) of over 100,000 noun, verb, adjective, and adverb word senses (Niles & Pease, 2003), which not only acts as a check on coverage and completeness, but also provides a basis for work in natural language processing (Pease & Murray, 2003) (Elkateb et al, 2006) (Scheffczyk et al, 2006). SUMO is now in its 75th free version; having undergone five years of development, review by a community of hundreds of people, and application in expert reasoning and linguistics. Various versions of SUMO have been subjected to formal verification with an automated theorem prover. SUMO and all the associated tools and products are available at www.ontologyportal.org .

1.1. The SUO-KIF Language

SUO-KIF, the Standard Upper Ontology Knowledge Interchange Format (Pease, 2003) was created as a variant of the KIF language (Genesereth, 1991) and designed to support the SUMO project. It retains the LISP-like syntax of the original KIF, but simplifies the language somewhat by including only logical operators in the language itself, leaving any ontology that employs the language to define and handle issues such as class and instance declarations and the difference between necessary and sufficient definitions (if any). It has a relatively "free" syntax, allowing higher-order constructs such as variables in the predicate position, quantification over formulas, and no restrictions such as prohibiting predicates and instances sharing names. On the other hand, the syntax is more restricted than some other variants of KIF in that constructs that have little use in common sense knowledge representation, such as empty conjunctions, are not allowed. Variables are denoted by a leading "?" character, and universal quantification, existential quantification, implication, and biimplication are shown as "forall", "exists", "=>" and "<=>", respectively. Quantifier lists are delimited by parentheses and quantified variables have no explicit sort syntax.

2. Conversion to First Order Logic

Since 2002 a customized version of Vampire (Riazanov & Voronkov, 2002) has been the primary system available for reasoning over SUMO, as part of the open source Sigma system (Pease, 2003). While the customizations allow Vampire to read SUO-KIF format, there are some restrictions, most notably on those aspects of SUO-KIF that appear to be beyond first order. Several transformations are required.

The first transformation is related to handling the type signature of predicates and functions. Provers such as Vampire are unsorted, and variables range over the Herbrand universe. However, SUMO specifies the signature of each predicate and function. When run in an unsorted prover, these specifications can have the unintended effect of generating contradictions. Because variables can be of any type, they can be bound to a term that is incompatible with the encompassing predicate or function's signature. The axiom that specifies the signature then contradicts that variable binding. In addition, by allowing variables to be any type, the prover's search may find variable bindings that cannot be part of the eventual successful solution, so there is an efficiency cost, as well as a problem for finding an accurate proof.

To solve this problem, we relativize the formulae by generating additional preconditions for each rule in the ontology, which then limits every formula to being considered only if type requirements have been met. For example,

```
(=>
   (and
       (instance ?TRANSFER Transfer)
       (agent ?TRANSFER ?AGENT)
       (patient ?TRANSFER ?PATIENT))
   (not
       (equal ?AGENT ?PATIENT)))
```

is transformed into

```
(=>
    (and
        (instance ?AGENT Agent)
        (instance ?PATIENT Object))
    (=>
        (and
            (instance ?TRANSFER Transfer)
            (agent ?TRANSFER ?AGENT)
            (patient ?TRANSFER ?PATIENT))
        (not
            (equal ?AGENT ?PATIENT)))
```

Note that a naïve implementation of this approach would be to state

```
(=>
    (and
        (instance ?AGENT Agent)
        (instance ?TRANSFER Instance)
        (instance ?TRANSFER Process)
        (instance ?PATIENT Object))
    (=>
        (and
            (instance ?TRANSFER Transfer)
            (agent ?TRANSFER ?AGENT)
            (patient ?TRANSFER ?PATIENT))
        (not
            (equal ?AGENT ?PATIENT)))
```

but since **?TRANSFER** is already constrained by the first clause of the original rule, those additional preconditions are not necessary.

There is an efficiency cost with using sortal prefixes, since they increase the number of literals that must be proved in order to derive each conclusion. We would expect the use of sortal prefixes to improve correctness, but at the cost of speed (and some space). The use of sortals has not provided any obvious benefit so far (see Section 5), possibly because we have not allowed each test to run for a long enough time. Further testing is planned.

The second transformation deals with SUO-KIF's row, or sequence variables, which follow a scheme proposed in (Hayes & Menzel, 2001). They are denoted by the '@' symbol in KIF statements. They are analogous to the Lisp language's **@REST** variable. This is not first order if the number of arguments it can handle is infinite. However, if row variables have a definite number of arguments, they can be treated like a macro, and become first order. For example,

```
(=>
    (and
        (subrelation ?REL1 ?REL2)
        (?REL1 @ROW))
    (?REL2 @ROW))
```

becomes

```
(=>
    (and
        (subrelation ?REL1 ?REL2)
        (?REL1 ?ARG1))
    (?REL2 ?ARG1))
```

```
(=>
    (and
        (subrelation ?REL1 ?REL2)
        (?REL1 ?ARG1 ?ARG2))
    (?REL2 ?ARG1 ?ARG2))
```

etc.

Note that this "macro" style expansion has the problem that unlike the intended semantics of row variables, it is not infinite. If the macro processor only expands to five variables, there is a problem if the knowledge engineer uses a relation with six. Because of that, Sigma's syntax checker must prohibit relations with more arguments than the row variable preprocessor expands to. Alternatively, we could first determine the maximum number of relation arguments used in the KB, and then perform macro expansion up to that number of arguments.

The third transformation universally quantifies all free variables. For example

```
(=>
    (and
        (subrelation ?REL1 ?REL2)
        (?REL1 ?ARG1))
    (?REL2 ?ARG1))
```

becomes

```
(forall (?REL1 ?REL2 ?ARG1)
    (=>
        (and
            (subrelation ?REL1 ?REL2)
            (?REL1 ?ARG1))
        (?REL2 ?ARG1)))
```

The fourth transformation eliminates the use of variables as predicates and functions. A typical SUMO axiom that uses a variable in a predicate position is

```
(=>
   (inverse ?REL1 ?REL2)
   (forall (?INST1 ?INST2)
      (<=>
         (?REL1 ?INST1 ?INST2)
         (?REL2 ?INST2 ?INST1))))
```

This illustrates a case of variables, **?REL1** and **?REL2**, being used as predicates. Strictly speaking, variables in a predicate position are not first order. However, if we adopt the simplifying assumption that such variables can range only over those predicates that appear in the formulae in use, the statements become first order. All that is needed is a simple syntactic transformation to make them appear so to a standard first order prover. To do this, a "dummy" predicate called "**holds_X__**" is prepended to every atom, where **X** is the arity of the predicate plus 1. This yields the following axiom

```
(=>
   (holds_3__ inverse ?REL1 ?REL2)
   (forall (?INST1 ?INST2)
      (<=>
         (holds_3__ ?REL1 ?INST1 ?INST2)
         (holds_3__ ?REL2 ?INST2 ?INST1))))
```

The inclusion of the arity in the "**holds_X__**" predicate is necessary to support provers that do not support variable arity predicates, and the trailing **__** avoids potential

conflicts with user predicates (which by convention should not end with **__**). An analogous approach is taken for functions, using an "**apply_X__**" function. For example

```
(=>
    (and
        (attribute (GovernmentFn ?AREA) ?TYPE)
        (instance ?TYPE FormOfGovernment))
    (governmentType ?AREA ?TYPE))
```

becomes

```
(=>
    (and
        (holds_2__ attribute (apply_2__ GovernmentFn ?AREA) ?TYPE)
        (holds_3__ instance ?TYPE FormOfGovernment))
    (holds_3__ governmentType ?AREA ?TYPE))
```

These "**holds_X__**" and "**apply_X__**" wrappers are added to all atoms (in SUMO every predicate has arity at least two), and to all non-constant function terms, even if the predicate or function position is not a variable. This consistent treatment allows the same unification possibilities as prior to the transformation, so that no completeness is lost. The transformation has an added benefit of improving performance for those provers which index clauses primarily on the predicate name.

The fifth transformation is significant. SUMO includes statements that are truly second order. For example

```
(=>
    (instance ?DEVICE MeasuringDevice)
    (hasPurpose ?DEVICE
        (exists (?MEASURE)
            (and
                (instance ?MEASURE Measuring)
                (instrument ?MEASURE ?DEVICE)))))
```

is an axiom that states that a **MeasuringDevice** has the purpose of being used as an instrument in a **Measuring** action. Because **hasPurpose** takes a formula as its second argument, it is not first order, and there is no simple trick or assumption that can be made to reduce it to first order. The only solution available is to lose most of the semantics of the statement, and turn it into an uninterpreted list. After transformation (omitting other transformations for clarity) the axiom becomes

```
(=>
    (instance ?DEVICE MeasuringDevice)
    (list hasPurpose ?DEVICE
        (exists (?MEASURE)
            (and
                (instance ?MEASURE Measuring)
                (instrument ?MEASURE ?DEVICE)))))
```

While **exists**, **and**, etc., all lose their semantics, at least it is possible for a theorem prover to unify over the list, retaining some limited possibility for reasoning with the statement. To choose a bit clearer artificial example, supposing we had

```
(believes Mary
  (likes Mary Bill))
```

an answer for **(believes Mary (likes ?X Bill))** could be found because although **(likes Mary Bill)** becomes an uninterpreted list after the transformation, it can still be subject to unification. However, if we had instead

```
(believes Mary
  (and
    (likes Mary Bill)
    (likes Sue Bill)))
```

an answer for `(believes Mary (likes ?X Bill))` could not be found because the two lists are not unifiable.

3. Conversion to TPTP

While the conversions described above result in an essentially first-order form, there are several aspects that are beyond the "traditional human-readable" format of the TPTP language, as used by many current provers. The TPTP language uses Prolog-like user terms and atoms, uses infix notation for binary operators, has a separate namespace for operators, and provides a separate namespace for defined functors and predicates. Additionally the TPTP language does not support arbitrary lists. These differences are dealt with in the translation to TPTP format as follows.

A stack-based algorithm is used to convert from the SUO-KIF prefix form for binary operators, stacking the translated form of an operator when found at the start of a formula, copying it off the top of the stack for insertion between operand formulae, and popping it off the stack at the end of the formula. As user terms and atoms are encountered they are translated to Prolog's prefix form, with variables prefixed by "`V_`", and function and predicate symbols prefixed by "`s_`". All hyphens in user terms are translated to underscores. Some defined functions are translated to corresponding equivalents from the TPTP language, starting with a "`$`" (note that the TPTP standards for defined arithmetic functions and predicates are in the process of being set as this paper is being written, so some minor changes may be necessary in this aspect of the translation to TPTP format). In the TPTP language double quoted strings are always interpreted as themselves so that different strings are known to be not equal. In the translation SUO-KIF double quoted strings are converted to single quoted constants, and non-printable characters - carriage return, new line, tab, and formfeed - are replaced by spaces. For example

```
(forall (?REL ?OBJ ?PROCESS)
   (=>
      (and
         (holds_3__ instance ?REL CaseRole)
         (holds_3__ instance ?OBJ Object)
         (holds_3__ ?REL ?PROCESS ?OBJ))
      (exists (?TIME)
         (holds_3__ overlapsSpatially
             (apply_3__ WhereFn ?PROCESS ?TIME) ?OBJ))))
```

is translated to

```
fof(name,axiom,
    ! [V_REL,V_OBJ,V_PROCESS] :
      ( ( holds_3__(s_instance,V_REL,s_CaseRole)
        & holds_3__(s_instance,V_OBJ,s_Object)
        & holds_3__(V_REL,V_PROCESS,V_OBJ) )
     => ? [V_TIME] :
          holds_3__(s_overlapsSpatially,
              apply_3__(s_WhereFn,V_PROCESS,V_TIME),V_OBJ) )).
```

In the translation to first-order form described in Section 2, it is explained that truly second order constructs are dealt with by losing most of the semantics by conversion to uninterpreted lists. This translated form is not directly usable in the TPTP format, as there

is no support for arbitrary lists. The current solution is to lose even more of the semantics, by single quoting such expressions, thus treating them as constants. In this way the possibility of unification over the list elements is lost - only unification of the whole is possible. Part of the reason for taking this simplistic approach is that operators have a separate namespace in the TPTP language, e.g., rather than SUO-KIF's **and** TPTP uses **&**. As a result TPTP operators cannot be treated as constants in a list function. The list solution can be implemented in the translation to TPTP format by retaining the SUO-KIF forms of operators (which look like TPTP constants), and forming atoms with a "**list_X__**" predicate to represent lists. For example

```
(=>
    (instance ?DEVICE MeasuringDevice)
    (hasPurpose ?DEVICE
        (exists (?MEASURE)
            (and
                (instance ?MEASURE Measuring)
                (instrument ?MEASURE ?DEVICE)))))
```

would be translated to

```
fof(name,axiom,
    ! [V_DEVICE,V_MEASURE] :
      ( holds_3__(s_instance,V_DEVICE,s_MeasuringDevice)
     => holds_3__(s_hasPurpose,V_DEVICE,
             list_3__(s_exists,V_MEASURE,
                 s_and(s_instance(V_MEASURE,s_Measuring),
                       s_instrument(V_MEASURE,V_DEVICE))) )).
```

4. Optimization

It is always possible to compare a prover optimized for a given set of problems to one that has not and show disappointing results for the unoptimized prover (Ramachandran et al, 2005). Our challenge has been to develop a set of simple optimizations that allow a set of standard, general-purpose, first-order provers to perform well on SUMO.

A first simple optimization is to cache transitive relationships. Almost any practical query on SUMO requires reasoning about subclass and instance relationships at some point during a proof. A standard prover does not give any special priority to SUMO's axiom of transitivity, so many proofs attempts can spend a lot of time searching dead end solution paths, when the answer is found mostly in a succession of applications of just one axiom. A simple way to solve this is to cache all the subclass relationships. This means that if SUMO authors have stated **(subclass C B)** and **(subclass B A)** that our optimization code also generates **(subclass C A)**, prior to any query being asked.

While prefixing all clauses with "**holds_X__**" is effective in making SUMO first-order, as described above, it might not be the most efficient strategy. An alternative approach is to instantiate all predicate and function variables with all the predicates and functions with the same arity. For example

```
(=>
    (instance ?REL TransitiveRelation)
    (forall (?INST1 ?INST2 ?INST3)
        (=>
            (and
                (?REL ?INST1 ?INST2)
                (?REL ?INST2 ?INST3))
            (?REL ?INST1 ?INST3))))
```

can be instantiated with **subclass** to yield

```
(=>
   (instance subclass TransitiveRelation)
   (forall (?INST1 ?INST2 ?INST3)
       (=>
           (and
               (subclass ?INST1 ?INST2)
               (subclass ?INST2 ?INST3))
           (subclass ?INST1 ?INST3))))
```

To avoid proliferating too many such instantiations however, the processor has to take into account restrictions in the axioms themselves. A naïve approach would instantiate the above axiom with predicates such as **agent,** which are not transitive. Our intuition about the computational advantages of this approach are not supported by current test results, as explained below.

5. Tests

The table below reports preliminary results of testing the translation and optimizations. The tests were performed using a 2002 version of Vampire on a 3.2GHz PC with 2.9GB of memory. The default query timeout was set to 180 seconds. The results are for tests performed on a KB consisting of SUMO plus the Mid-Level Ontology (MILO).

The heading element "sortals" refers to the addition of type constraint antecedents to axioms. The heading element "holds" refers to the addition of the artificial predicate "**holds_X__**" to every clause. When tests are run with the option "instantiate", predicate variable instantiation is used instead. The heading element "caching" refers to pre-computing the transitive closure of subsumption relations. The values in the "Overall Ranking" row were computed with the following off-the-cuff algorithm: for each table:

1. Find the lowest value for Avg. total seconds. Call this LV.
2. For each column, take the average number of failed queries (i.e., 50 - Avg. # of successful queries). Call this IA.
3. For each column take the Avg. total seconds value. Call this TS.
4. For each column, compute an index I using this formula: I = ((IA/50) * (TS/LV)). Essentially, this is an ad hoc index of "badness", giving equal weight to avg. number of failed queries and avg. total time per query run per parameter configuration (column). The smaller this index, the "better" the overall performance for this configuration of parameter settings relative to the other configurations of settings.
5. Assign a rank to each column, based on the computed "badness" index, with 1 being best and 8 being worst.

Surprisingly, the optimizations we have implemented appear to give better performance with our 2002 version of Vampire in only a few configurations, and the results are difficult to interpret. Caching the subclass hierarchy does not generally improve the query success rate. We surmise that the detrimental effect of caching is related to the greatly increased size of the KB, but explanation of the actual causes requires further investigation.

Before running the tests, we expected that the introduction of sortals, by constraining the search space, would improve both query success rate and answer time. However, we

found instead that, for all combinations of using holds and caching, the introduction of sortals degrades both success rate and performance. We believe that the degradation results from the fact that sortals add extra literals which must be solved for each proof. It may be that none of our current tests adequately targets the main problem that the introduction of sortals was intended to solve: making predicate argument type constraints more accessible to our 2002 version of Vampire, and thereby preventing spurious conclusions. The instantiation of predicate variables (*i.e.*, no use of **`holds_X__`** prefixes) resulted in some improvement over using holds prefixes.

	no sortals, instantiate, no caching	**sortals, instantiate, no caching**	**no sortals, holds, no caching**	**sortals, holds, no caching**	**no sortals, instantiate, caching**	**sortals, instantiate, caching**	**no sortals, holds, caching**	**sortals, holds, caching**
Avg. % successful	86%	82%	86%	30%	76%	82%	50%	32%
Avg. num. successful	43/50	41/50	43/50	15/50	38/50	41/50	25/50	16/50
Avg. total seconds	4,010	5,814	9,689	12,140	6,849	8,430	10,003	9,377
Normalized avg. total time	0.33	0.48	0.8	1	0.56	0.69	0.82	0.77
Overall Rank	1	2	5	8	3	4	6	7

Table 1: Summary of Aggregate Performance per Run by Parameter Cluster

6. Conclusion

We plan to continue our experiments along several dimensions. We need to expand the variables tracked when running tests to include the numbers and types of formulas ("rules", Horn clauses, unit clauses, *etc.*), and the number and characteristics of the proof(s) used to obtain each answer. We need to expand the number of tests and ensure that they are representative of the queries typically posed in current applications. We need to run in different memory configurations, to determine the impact of memory paging on performance when the knowledge base is very large. We need to run tests on SUMO alone, and on SUMO plus all of its domain ontologies. We need to run on all the provers in the current TPTP suite, including the most recent version of Vampire. The evolving set of tests is available at

http://sigmakee.cvs.sourceforge.net/sigmakee/KBs/tests/

and the Sigma system that runs these tests is available at

http://sigmakee.sourceforge.net/ .

Acknowledgments

This work has been funded by a number of sources, including the US Air Force, Army CECOM, and DARPA. We are grateful for their investment. Some of this most recent work has been helped from collaboration and discussion with German Rigau and his students and colleagues at Universitat Politècnica de Catalunya and La Universidad del País Vasco. We also appreciate the contributions of the anonymous ESARLT reviewers.

References

Elkateb, S., Black, W., Rodriguez, H, Alkhalifa, M., Vossen, P., Pease, A. and Fellbaum, C., (2006). Building a WordNet for Arabic, in *Proceedings of The fifth international conference on Language Resources and Evaluation (LREC 2006)*.

Fellbaum, C. (ed.) WordNet: An Electronic Lexical Database. MIT Press, 1998.

Genesereth, M., (1991). "Knowledge Interchange Format", In Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning, Allen, J., Fikes, R., Sandewall, E. (eds), Morgan Kaufman Publishers, pp 238-249.

Hayes, P., and Menzel, C., (2001). A Semantics for Knowledge Interchange Format, in *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology*.

Niles, I & Pease A., (2001). "Towards A Standard Upper Ontology." In *Proceedings of Formal Ontology in Information Systems (FOIS 2001)*, October 17-19, Ogunquit, Maine, USA, pp 2-9. See also http://www.ontologyportal.org

Niles, I., and Pease, A. (2003) Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology, *Proceedings of the IEEE International Conference on Information and Knowledge Engineering*, pp 412-416.

Pease, A., (2003). The Sigma Ontology Development Environment, in *Working Notes of the IJCAI-2003 Workshop on Ontology and Distributed Systems*. Volume 71 of CEUR Workshop Proceeding series. See also http://sigmakee.sourceforge.net

Pease, A., (2004). Standard Upper Ontology Knowledge Interchange Format. Unpublished language manual. Available at http://sigmakee.sourceforge.net/

Pease, A., and Murray, W., (2003). An English to Logic Translator for Ontology-based Knowledge Representation Languages. In *Proceedings of the 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering*, Beijing, China, pp 777-783.

Ramachandran, D., P. Reagan, K. Goolsbey. First-Orderized ResearchCyc: Expressivity and Efficiency in a Common-Sense Ontology. In *Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications*. Pittsburgh, Pennsylvania, July 2005.

Riazanov A., Voronkov A. (2002). The Design and Implementation of Vampire. *AI Communications*, 15(2-3), pp. 91—110.

Scheffczyk, J., Pease, A., Ellsworth, M., (2006). Linking FrameNet to the Suggested Upper Merged Ontology, in *Proceedings of Formal Ontology in Information Systems (FOIS-2006)*, B. Bennett and C. Fellbaum, eds, IOS Press, pp 289-300.

Sutcliffe G., Suttner C.B. (1998), The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning* 21(2), 177-203.