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/ .

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.