OntoMorph: A Translation System for Symbolic Knowledge *

Hans Chalupsky
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292 U.S.A.
hans@isi.edu

Abstract

A common problem during the life cycle of knowledge-based systems is that symbolically represented knowledge needs to be translated into some different form. Translation needs occur along a variety of dimensions, such as KR language syntax and expressivity, modeling conventions, representation paradigms, etc. As a tool to support the translation problem, we present the OntoMorph system. OntoMorph provides a powerful rule language to represent complex syntactic transformations, and it is fully integrated with the PowerLoom KR system to allow transformations based on any mixture of syntactic and semantic criteria. We describe OntoMorph's successful application as an input translator for a critiquing system and as the core of a translation service for agent communication. We further motivate how OntoMorph can be used to support knowledge base merging tasks.

1 Introduction

A common problem during the development of ontologies and knowledge-based systems is that symbolically represented knowledge needs to be translated into some different form. For example, integration of independently developed knowledge-based components [Cohen *et al.*, 1998], merging of overlapping ontologies [Valente *et al.*, 1999], communication between distributed, heterogeneous agents, or porting of knowledge-based systems to use a different knowledge representation infrastructure commonly require translation, since every encoding of knowledge is based on a multitude of representational choices and assumptions. Translation needs go well beyond syntactic transformations and occur along many dimensions, such as expressivity of representation languages, modeling conventions, model coverage and granularity, representation paradigms, inference system bias, etc., and any combination thereof.

Traditionally, such translations are either performed manually via text or knowledge base editors or via special-purpose translation software. Manual translation is slow, tedious, error-prone, hard to repeat and simply not practical for certain applications. Special-purpose translation software is difficult to write, hard to maintain and not easily reusable.

Being confronted with translation problems on a frequent basis, we are developing the OntoMorph system to facilitate ontology merging and the rapid generation of knowledge base (KB) translators. OntoMorph combines two powerful mechanisms to describe KB transformations: (1) *syntactic rewriting* via pattern-directed rewrite rules that allow the concise specification of sentence-level transformations based on pattern matching, and (2) *semantic rewriting* which modulates syntactic rewriting via (partial) semantic models and logical inference supported by an integrated KR system. The integration of these mechanisms allows transformations to be based on any mixture of syntactic and semantic criteria, which is essential to support the translation needs enumerated above. The OntoMorph architecture facilitates incremental development and scripted replay of transformations, which is particularly important during merging operations.

*To appear in A. G. Cohn, F. Giunchiglia, and B. Selman, editors, Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000), Morgan Kaufmann Publishers, San Francisco, CA. All quotes should be from, and all citations should be to the published version.

2 The Translation Problem

The general problem we are considering is shown in Figure 1: Given some source knowledge base KB_s we

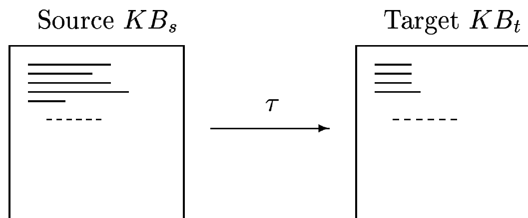

Figure 1: The knowledge translation problem

want to design a transformation function τ to transform it into a target knowledge base KB_t. A fundamental assumption in this formulation is that source and target KBs are describable by a set of sentences in some linear, textual notation, where *sentence* means some independent syntactic unit as opposed to a well-formed logical formula associated with a truth value. This does not exclude graphical languages such as, for example, SNePS [Shapiro and Rapaport, 1992] or Conceptual Graphs [Sowa, 1992], since they usually also have some linear syntax to textually describe their networks. The translation does not necessarily have to span a whole knowledge base. In some cases, it might only involve single expressions.

A common correctness criterion for translation systems is that they preserve semantics, i.e., the meaning of the source and the translation has to be the same. This is not necessarily desirable for our transformation function τ, since it should be perfectly admissible to perform abstractions or semantic shifts as part of the translation. For example, one might want to map an ontology about automobiles onto an ontology of documents describing these automobiles. Since this is different from translation in the usual sense, we prefer to use the term *knowledge transformation* or *morphing*.

3 Dimensions of Mismatch

Despite the fact that the function τ might perform arbitrary semantic shifts, the most common scenario is to translate between different models of the same general domain. Unfortunately, these models can and in practice do differ along a multitude of dimensions. The most commonly encountered mismatches are outlined below.

KR language syntax: Every KR language comes with its own syntax, which is probably the most mundane but nevertheless annoying mismatch. For example, here are three different ways of defining automobiles as a subclass of road vehicles, one for Loom [MacGregor, 1991], a KL-ONE-style description logic, one for MELD, the representation language used by CYC [Lenat, 1995] and one for KIF [Genesereth, 1991]:

```
Loom:  (defconcept Automobile
         "The class of passenger cars."
         :is-primitive Road-Vehicle)

MELD: (#$isa #$Automobile #$Collection)
      (#$genls #$Automobile #$RoadVehicle)
      (#$comment #$Automobile
        "The class of passenger cars.")

KIF:  (defrelation Automobile (?x)
        "The class of passenger cars."
        :=> (Road-Vehicle ?x))
```

Apart from different surface syntax, there are also different *syntactic conventions* such as the spelling of names that are really part of the culture of the language users. For example, CYC names are mixed-case without hyphens as opposed to the hyphenated, case-insensitive spelling usually used with the other languages.

KR language expressivity: Every KR language trades off representational expressiveness with computational tractability. For example, negation, quantification, defaults, modal operators, representation of sets, etc. are supported by some languages and not by others. When translating between languages of different expressiveness, difficult choices have to be made in how to map certain representational idioms. For example, to represent that the typical capacity of a passenger car is five, we could use the following representation in Loom:

```
(defconcept Automobile
 :is-primitive Road-Vehicle
 :defaults (:filled-by passenger-capacity 5))
```

To represent the same in ANSI KIF which does not support defaults, one would have to resort to something like the following and then leave it up to some extra-logical means to properly reason with typicality assertions:

```
(defrelation Automobile (?x)
 :=> (and (Road-Vehicle ?x)
          (typical-passenger-capacity ?x 5)))
```

Modeling conventions: Even if the KR language and system for source and target KB are the same, differences occur because of the way a particular domain is modeled. For example, a choice one often has to make is whether to model a certain distinction by introducing a separate class, or by introducing a qualifying attribute relation. E.g., to distinguish between tracked and wheeled vehicles, one could either introduce two subclasses of `Vehicle` called `Tracked-Vehicle` and `Wheeled-Vehicle`, or use an attribute relation as in `(traction-type My-Car wheeled)`. Which representation to choose is in most cases just a matter of taste or convention.

Model coverage and granularity: Models differ in their coverage of a particular domain and the granularity with which distinctions are made. This is often the very reason why ontologies are merged. For example, one ontology might model cars but not trucks. Another one might represent trucks but only classify them into a few categories, while a third one might make very fine-grained distinctions between types of trucks based on their general physical structure, weight, purpose, etc.

Representation paradigms: Different paradigms are used to represent concepts such as time, action, plans, causality, propositional attitudes, etc. For example, one model might use temporal representations based on Allen's interval logic [Allen, 1984], while another might use a representation based on time points. Section 5 describes a situation where two different representations of "purpose" had to be reconciled with the help of OntoMorph.

Inference system bias: Last but not least, another reason why models often look a certain way is that they were constructed to produce desired inferences with a particular inference engine or theorem prover. For example, in a description logic such as Loom, certain inferences are well-supported by the classifier, while others are only supported at the instance or individual level. This trade-off can influence one's choice whether to model something as a class or as an individual. See [Valente *et al.*, 1999] for a discussion of modeling examples exhibiting inferencing bias.

4 OntoMorph

To facilitate the rapid specification of KB transformation functions such as τ described above, OntoMorph combines two powerful mechanisms: (1) *syntactic rewriting* via pattern-directed rewrite rules that allow the concise specification of sentence-level transformations based on pattern matching, and (2) *semantic rewriting* which modulates syntactic rewriting via (partial) semantic models and logical inference.

4.1 Syntactic Rewriting

To allow translation between arbitrary KR languages that can differ widely in their syntax, expressiveness, and underlying knowledge model, OntoMorph uses *syntactic rewriting* as its core mechanism. Input expressions are first tokenized into lexemes and then represented as syntax trees whose subtrees represent parenthesized groups (similar to Lisp s-expressions). The tree structure exists only logically; a tree is represented internally as a flat sequence of tokens.

For example, the expression `f(g([x],y))` would be represented by the token sequence

`'f' '(' 'g' '(' '[' 'x' ']' ',' 'y' ')' ')'`

which, logically, represents the syntax tree shown in Figure 2. The significance of the tree structure is that complete subtrees can be matched by a single pattern variable, and that sequence variables do not consume tokens beyond subtree boundaries.

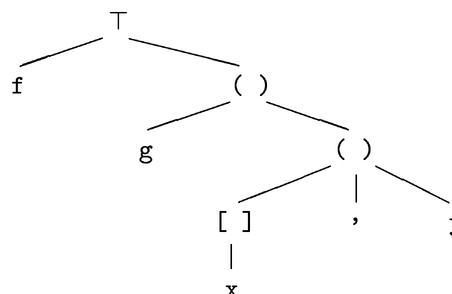

Figure 2: Syntax tree representation of `f(g([x],y))`

OntoMorph's syntactic rewrite rules have this general form:

$$pattern \;\Rightarrow\; result$$

The left-hand-side pattern matches and destructures one or more syntax trees while the right-hand side generates new trees of the desired format by explicitly specifying new structure, reassembling some of the destructured information and by possibly further rewriting some subexpressions. For example, a very simple rule to convert a MELD type assertion into its Loom analogue would look like this (pattern variables are prefixed with a '?'):

```
(isa ?x ?class) ==> (tell (?class ?x))
```

The ability to describe such transformations in a very direct and concise fashion was an important design objective for OntoMorph. When researching the relevant parsing and pattern-match literature and technology, we found that a language called Plisp (or Pattern Lisp) [Smith, 1990], which in turn is a direct descendent of the Lisp 70 pattern matcher [Tesler *et al.*, 1973], came closest to our intuitions on how such transformations should be represented and executed. Unfortunately, none of these systems is alive and well anymore, so we had to develop our own version.

4.1.1 Pattern Language

OntoMorph's pattern language and execution model is strongly influenced by Plisp, even though the actual surface syntax is quite different. The pattern language can match and destructure arbitrarily nested syntax trees in a direct and concise fashion. A short overview of the available constructs is given below:

Literals such as `foo`, `"bar"`, `42`, `(`, `(a (b c))`, etc., have to be matched by identical literal tokens (or token sequences).

Variables (indicated by a ?-prefix), e.g., `?x`, `?why` or the anonymous variable `?`, can match individual input tokens such as `foo` or a token sequence representing a tree such as `(a (b c))`. Once a variable is bound, it can only be matched by literal tokens matching its binding.

Sequence variables (indicated by a ??-prefix), e.g., `??h`, `??tail` or the anonymous variable `??`, can match tree subsequences such as `c (d)` in the tree `(a b c (d))`. For example, the pattern `(??x b ??y)` matches the tree `(a b c (d))` by binding `??x` to the single-element sequence `a` and `??y` to the sequence `c (d)`. Sequence variables cannot consume tokens beyond subtree boundaries.

Grouping (expressed via braces) defines compound patterns. For example, the pattern `{a ?x c}` can match the token sequence `a b c`. Groups are also used to apply pattern modifiers such as repetition to compound patterns.

Alternatives (expressed via vertical bars) define disjunctive patterns such as `{a | (b ?x) | c d}`. The pattern matches if one of its components succeeds.

Optionals such as `{a b [c]}` are syntactic sugar for the more verbose `{a b | a b c}` notation.

Repetition (expressed with the usual `*` or `+` notation) indicates that a pattern can be matched multiple times. As a generalization, an `m-n` range can be supplied to mandate that there have to be at least `m` and at most `n` matches. For example, `{a | b}+` matches any sequence of `a`'s and `b`'s with length ≥ 1, `{a | b}*1-2` matches only those sequences with lengths between 1 and 2.

Input binding binds the input matched by a complex pattern to a single variable. This is useful if a pattern has alternatives and it is necessary to refer to what was actually matched by it in the right-hand side of a rewrite rule (without alternatives, the same could be achieved by literally repeating the pattern). For example, `?x := {a | (b ?y) |c}` matched against `(b d)` binds `?x` to `(b d)`.

Below is an example pattern that combines various of the elements described above to match and destructure a Loom concept definition (note, that the example only covers some aspects of the Loom concept language). The alternatives in combination with the repetition construct allow the keyword/value pairs to appear in any order. The construction for the `:annotations` keyword extracts a documentation string (which might appear in various ways) while ignoring everything else:

```
(defconcept ?name
  {?is := {:is | :is-primitive} ?def |
   :characteristic ?c |
   :annotations
    ?a := {(documentation ?d) |
           (:and ?? (documentation ?d) ??) |
           ?}}*0-3)
```

The pattern matches and destructures concept definitions such as this one:

```
(defconcept Dog
   :annotations
      (:and Class (documentation "Canine"))
   :is-primitive Animal)
```

4.1.2 Execution Model

Rewrite rules are applied according to the following simple execution model: Initially, an input stream is constructed consisting of the token sequence representing the input expression. When a rewrite rule is applied, its left-hand-side pattern consumes tokens from the input stream by matching them against the elements of the pattern. If the pattern succeeds, the right-hand-side result is assembled and the resulting tokens are pushed back onto the input stream where they replace the consumed input and become available as input to further rewrite rules. For example, assume we have the following input stream:

```
'(' 'isa' 'car1' 'Ford' ')' '(' ...
```

Now we apply the type transformation rule from before:

```
(isa ?x ?c) ==> (tell (?c ?x))
```

Applying the rule modifies the input stream to the following:

```
'(' 'tell' '(' 'Ford' 'car1' ')' ')' '(' ...
```

Assembly of a rule result involves collecting its right-hand-side component tokens from left to right into a temporary store. Literal tokens such as the tell above are simply copied, variables are substituted by their bindings, and functions and recursive rule invocations (explained below) are evaluated and their results collected. Once all right-hand-side components have been successfully evaluated, the content of the temporary store is prepended to the input stream where it replaces the input consumed by the left-hand-side.

Rewrite rules are always assembled into rule sets of the following form:

(defruleset *name*
  $pattern_1 \Rightarrow result_1$
  ...
  $pattern_n \Rightarrow result_n$)

The individual rules are implicitly OR-ed and tried in sequence. The ruleset succeeds with the result of the first successful element rule.

Explicit invocation of named rulesets is the primary mechanism to achieve recursion, which is necessary to handle the translation of recursive structures. Apart from this computational aspect, grouping rules improves modularity, and it also greatly improves efficiency, since it restricts the set of rules tried to rewrite any given subexpression.

While matching a pattern and also during the assembly of the right-hand-side result which might involve further rewrites, a rule may fail. In that case execution backtracks to the most recent match choice point. After all input has been consumed and no more rules need to be applied, the process terminates and the resulting state of the input stream constitutes the result of the rewrite operation which is then either printed to some storage medium or used directly as part of a KB operation such as assertion or retrieval.

4.1.3 Function Calls and Rule Invocations

To allow the parsing and rewriting of recursive structures, other rulesets as well as built-in functions can be invoked explicitly anywhere in a pattern. Such invocations are written with an angle bracket syntax to distinguish them from the regular syntax tree notation. For example, the call <Term ?x> invokes a function or ruleset called Term on the argument ?x. Before the function is called, its arguments are evaluated and the results pushed back onto the input stream from which they are then consumed. Excess or missing arguments are left on or filled in from the remainder of the input. When a function or ruleset invocation on the left-hand side of a rule returns, its result gets pushed back onto the input where it immediately becomes available to subsequent pattern elements. On the right-hand side (as described above), the result gets first collected in a temporary store until all right-hand-side tokens of the rule have been evaluated.

The following two rule sets constitute a simple transformation system for arithmetic expressions (note, that the + and * symbols need to be escaped to treat them as ordinary characters):

```
(defruleset Term
  (?op := {\+ | - | \* | /} ?x ?y)
     ==> (?op <Term ?x> <Term ?y>)
  (1\+ ?x) ==> (\+ <Term ?x> 1)
  (1- ?x) ==> (- <Term ?x> 1)
  (square ?x) ==> (\* <Term ?x> <Term ?x>)
  ?x ==> ?x)

(defruleset Condition
  (lt ?x ?y)
     ==> (negative? (- <Term ?x> <Term ?y>))
  (gt ?x ?y) ==> <Condition (lt ?y ?x)>)
```

To apply these rules, we can use the OntoMorph function rewrite which takes an input expression and a start rule as arguments. For example,

```
(rewrite (gt (/ (1+ M) N) (square N))
         Condition)
```

returns the following result:

```
(negative? (- (* N N) (/ (+ M 1) N)))
```

Currently, OntoMorph uses a Lisp-style reader to tokenize the input into individual lexemes. Future versions will allow the specification of customized tokenizers in order to support the translation of languages with different lexical conventions.

4.2 Semantic Rewriting

Syntactic rewriting is a powerful mechanism to describe pattern-based, sentence-level transformations.

However, it is not sufficient if the transformations have to consider a larger portion of the source KB, possibly requiring logical inference. A simple example of such a transformation is conflation. Suppose one wants to conflate all subclasses of `Truck` occurring in some ontology about vehicles into a single `Truck` class. This involves among other things the rewriting of all type assertions involving trucks. Using syntactic rewriting alone, one would need a rule such as the following which explicitly lists all subtypes of `Truck`:

```
(defruleset Conflate-Truck-Types
  ({Light-Truck | Heavy-Truck | ...} ?x)
    ==> (Truck ?x))
```

For large taxonomies this is of course neither elegant nor feasible. Instead of the purely syntactic test based on truck class names, a semantic test is needed to check whether a particular class is a subclass of `Truck`.

To facilitate the utilization of semantic information, OntoMorph is built on top of the PowerLoom knowledge representation system. PowerLoom is a successor to the Loom system that supports definitions and rules in a typed variant of KIF combined with a powerful inference engine and a classifier. Wherever a function call is legal in a rewrite rule, a PowerLoom function can be called to change or access the state of the current KB.

One way to solve the conflation problem is to establish a partial mirror of the source KB within an intermediate PowerLoom KB. This can be done with a specialized set of rewrite rules that import source sentences representing taxonomic relationships, but ignoring all other information, for example, by only paying attention to `subset` and `superset` assertions. This step can be viewed as the first pass of a two-pass translation scheme. In the second pass, the actual translation rules are applied, but now they can also access the semantic information established in the first pass. Making use of the imported taxonomic knowledge, the following rule can conflate all truck types:

```
(defruleset Conflate-Truck-Types
 {(?class ?x) <ask (subset-of ?class Truck)>}
    ==> (Truck ?x))
```

The left-hand-side contains a group of patterns which is treated as a conjunction. The first conjunct `(?class ?x)` simply matches any type assertion. The second one calls `ask` which triggers a PowerLoom query. Note that `?class` will be substituted with the matched class name, thus, the query will be fully ground. Since `ask` is a boolean-valued function, its result will simply be treated as a test instead of being pushed back onto the input stream.

Using semantic import rules, an arbitrarily precise image of the source KB semantics can be established within PowerLoom (limited only by the expressiveness of first-order logic). Then syntactic rewrite rules can use the imported semantic information to perform rewrites based on any mixture of syntactic and semantic criteria.

Obviously, the precision of the semantic import will affect the quality of the translation. For example, in the scenario above the semantic import only considered `subset` and `superset` assertions. Depending on the nature of the source KB, there might be other information and rules that would allow one to infer additional taxonomic relationships. These would then not be inferable within the partial PowerLoom mirror KB which might adversely affect the translation quality.

Whether this is a problem and how to best solve it has to be decided on a case-by-case basis. One solution is to use PowerLoom as an interlingua and import everything from the source KB (again, this is limited only by the expressiveness of PowerLoom). The disadvantage of this scheme is that one effectively needs two sets of translation rules, one to translate from the source into PowerLoom, and one to go from PowerLoom to the target representation. Alternatively, it might be possible to call out to the KR system that has the source KB loaded and use its inferencing capabilities directly. This can either be done via some special-purpose API, or, if supported, via a protocol such as OKBC [Chaudhri *et al.*, 1998]. Which route to take will depend on a variety of pragmatic factors. For the OntoMorph applications constructed to date, importing partial semantic information into PowerLoom was sufficient to support all rewriting needs.

5 OntoMorph Applications

OntoMorph has already been successfully applied in a couple of domains. One involved the translation of military scenario information for a plan critiquing system. In the second it formed the core of an agent translation service called Rosetta, where it was used to translate messages between two communicating planning agents that used different representations for goals.

5.1 Course of Action Critiquer

One of the challenge problems that drove the second phase of DARPA's High Performance Knowledge Bases (HPKB) project [Cohen *et al.*, 1998] was to develop critiquing systems for military courses of actions

(or COAs) which are high-level, plan-like descriptions of military operations. To represent a particular COA, scenario information from a graphical sketch pad was fused with information from a natural language description of the COA by a program called the Fusion engine. The combined description of the COA was represented in CYC's MELD language and then fed to five independent critiquing systems built by different teams. Only one of the critiquers was using CYC directly and did not have to translate the Fusion engine output. All others had to use some form of translation system. Many different scenarios had to be handled in a tight evaluation schedule, thus, manual translation was not an option. OntoMorph was chosen to translate the Fusion output for the critiquer based on the EXPECT knowledge acquisition system [Gil, 1994] which uses Loom to represent its knowledge. What follows is a list of translation issues that arose, and how they were solved:

Different Names: While most of the names generated by the Fusion engine were shared by the EXPECT critiquer, some of them differed due to parallel independent development of critiquers and ontologies as well as personal style. Renaming was taken care of with simple rules like the following:

```
(DEFRULESET rename-collection
  Fix-MilitaryTask ==> FIX
  {ProtectingSomething |
   ProtectingPhysicalRegion} ==> PROTECT
  Translation-LocationChange ==> MOVE
  ...)
```

Different Syntax: OntoMorph started with a KIF translation of the Fusion output which still contained various MELD idioms that needed to be translated into Loom syntax. For example, `isa` assertions such as `(isa task1 Fix-MilitaryTask)` had to be translated into the Loom idiom `(FIX task1)` (which here also involved a name change). MELD frame predicates were also easily translated into Loom with the following rule:

```
(DEFRULESET rewrite-frame-predicate
  (relationInstanceExistsCount
    ?relation ?instance ?type ?count)
  ==> (:ABOUT <rewrite-term ?instance>
        (:EXACTLY ?count
          <rename-relation ?relation>
          <rename-collection ?type>)))
```

Different Representations: The most challenging difference to overcome was the different representations used to represent the purposes of tasks. The Fusion engine used an idiom that related a task with a proposition whose truth was supposed to be brought about by carrying out the task. For example, to state that the purpose of the task carried out by `BlueDivision1` was to protect `Boundary1`, the following representation was used:

```
(taskHasPurpose BlueDivisionTask
  (thereExists ?p
    (isa ?p
         (CollectionSubsetFn
           ProtectingSomething
           (TheSetOf ?obj
             (and (objectTakenCareOf
                     ?obj Boundary1)
                  (performedBy
                     ?obj BlueDivision1)))))))
```

This can roughly be paraphrased as follows: The purpose of `BlueDivisionTask` is to bring about the existence of an event `?p` that is an instance of the event type `ProtectingSomething` restricted by the set of events in which `BlueDivision1` takes care of `Boundary1` (the restriction is expressed via the `CollectionSubsetFn` construction). This representation goes far beyond the expressiveness of Loom which does not have a way to represent higher-order sentences such as the above. It also did not meet the requirements of the EXPECT critiquer, which needed a reified purpose representation such as the following:

```
(AND (PROTECT protect00)
     (PURPOSE-ACTION protect00)
     (PURPOSE-OF BlueDivisionTask protect00)
     (ACTION-OBJ protect00 Boundary1)
     (WHO protect00 BlueDivision1))
```

The final version of the Fusion engine only used three structurally different purpose representation patterns. Each of them could be handled by an OntoMorph rule such as the following:

```
(DEFRULESET rewrite-purpose-pattern1
  {(taskHasPurpose ?task
     (thereExists ?var
       (isa ?var
            (CollectionSubsetFn
              ?type
              (TheSetOf ?action ?body)))))
   <generate-unique-name
     <rename-collection ?type>>
   ?purpose}
  ==> (AND
        (<rename-collection ?type> ?purpose)
        (PURPOSE-ACTION ?purpose)
```

```
        (PURPOSE-OF ?task ?purpose)
        <rewrite-purpose-setof-body
          ?body ?action ?purpose>))
```

The mapping between the two representations is very direct and makes good use of OntoMorph's destructuring facilities for syntax trees. The only complication is the extra right-hand-side function call to create a skolem individual needed to represent the reified purpose. This is taken care of by a call to the built-in function `generate-unique-name` which bases the generated name on the supplied argument (in this case, the renamed base event type). It does not consume anything from the input stream but simply pushes the result back onto it where it is then consumed by the `?purpose` variable.

Missing Representations: Some information needed by the EXPECT critiquer such as COA substructure and task/subCOA associations was not explicitly represented and needed to be recovered by some of OntoMorph's semantic rewrite features, e.g., by keying in on "meta-information" such as where in the Fusion output certain assertions were made.

For example, to associate a task with a particular sub-COA, it was necessary to track what tasks were performed by what unit which was handled by the following two rules:

```
(DEFRULESET track-COA-assertion
  (unitAssignedToTask ?task ?unit)
  ==> <!ASSERT
        (AND (Term ?task) (Term ?unit)
             (unitAssignedToTask
               ?task ?unit))>)

(DEFRULESET get-task-assigned-to-unit
  {?unit
   <@RETRIEVE \?t
     (= (unitAssignedToTask \?t) ?unit)>
   ?task}
  ==> <OBJECT-NAME ?task>)
```

The first rule creates a PowerLoom assertion for each `unitAssignedToTask` statement in the Fusion scenario. PowerLoom expects all its objects to be typed before they are used which is the reason for the additional `Term` assertions. The second rule retrieves the task recorded for a particular unit which was then used to associate it with the sub-COA in which the particular unit was involved. Note, that the `?t` variable within the PowerLoom `retrieve` statement is escaped, since it is a retrieval variable and not a pattern variable of the rewrite rule. The `?unit` variable, however, is a pattern variable, thus, its binding is substituted before the retrieval is executed and is seen by PowerLoom as an ordinary constant.

The complete translator was comprised of about 30 rulesets, 10 of which were necessary just to track unrepresented COA structure. The size of the translator was about 15 kilobytes of text.

5.2 Rosetta Agent Translation Service

Rosetta is a prototype of an ontology-based translation service operating in a domain of planning agents. It reengineers some aspects of a technology integration effort described in [Cox and Veloso, 1997] which connects the ForMAT case-based planning tool [Mulvehill and Christey, 1995] with the Prodigy/Analogy planner [Veloso, 1994; Veloso *et al.*, 1995]. In the original experiment, special-purpose translators were constructed to allow ForMAT and Prodigy/Analogy to communicate. Rosetta is an attempt to show how these translators can be replaced with a more flexible, general-purpose translation architecture that promotes reuse and that can scale up to large numbers of heterogeneous, communicating agents. The full motivation and details of the Rosetta architecture are given in [Blythe *et al.*, 2000]. Here we will only touch on some aspects and how they are handled by the OntoMorph system.

The main idea behind Rosetta is that it provides a representation interlingua in conjunction with a repository of broad-coverage as well as domain-specific ontologies that can be used to represent content expressions exchanged by heterogeneous communicating agents. Each agent is associated with a wrapper that (1) translates its message content language into the interlingua used by Rosetta, and (2) if necessary, aligns terms of the agent ontology with Rosetta's ontologies. Within Rosetta, each agent is associated with a model that represents relevant aspects of the agent's domain. As motivated in Section 3, using the same KR language and system to model a domain does not by itself eliminate the need for translation, since different representations can be used to express the same semantic content. To facilitate translations between such different representations, Rosetta has a library of representation reformulation rules.

To translate a message between agents A and B, agent A first uses its wrapper to translate the message content into Rosetta's format and sends it to Rosetta. Rosetta then checks whether any reformulation rules need to be applied to make the message understandable by agent B, and, if so, applies them. The result-

ing message is then sent to agent B which uses its own wrapper to translate it into its internal format. One of the advantages of this architecture is that the portion of the necessary translation mappings encodable in the wrappers grows only linearly with the number of different agent classes.

The uses of OntoMorph within this scenario were twofold: (1) It provided an obvious solution to implement the agent wrappers by primarily relying on its syntactic rewriting features. (2) Its semantic rewriting features were used to implement the necessary representation reformulation rules. For example, the following top-level rule was used to translate a goal posted by the ForMAT planning tool into the Rosetta representation (note, that only the most relevant aspects of these rules are reproduced to save space):

```
(defruleset format-to-rosetta-wrapper
  {(:goal ?goal)
   <translate-format-goal-to-rosetta ?goal>
   ?translated-goal}
 ==>
  (message
    (content
      (find (object plans)
            (for (Objective-Based-Goal
                    ?translated-goal))))
    ...) ...)
```

This rule translates a ForMAT request such as

```
(:goal
  (G-144 :Send-Hawk
         ((force 42nd-Batt)
          (geographic-location Big-Town))))
```

into the following representation understandable by Rosetta (the message content language used for this prototype is based on the verb clause goal language used by the EXPECT system):

```
(message
  (content
    (find (object plans)
          (for (Objective-Based-Goal
                 (send-unit
                   (object 42nd-Batt)
                   (to Big-Town))))))
  ...)
```

Once this message arrives at Rosetta, it is handled by its top-level translation rule whose main purpose it is to trigger the translation of content expressions:

```
(defruleset translate-rosetta-message
  {(message
    {(content ?content) |
     ...}*4-4)
   <map-performative ?content>
   ?mapped-performative}
 ==>
  (message
    (content ?mapped-performative)
    ...) ...)
```

One of the interesting aspects of the communication between ForMAT and Prodigy/Analogy is that ForMAT uses an objective-based or verb-centered representation such as "send troops to X" to represent its goals. Prodigy/Analogy, on the other hand, needs to be given a state-based goal representation such as "troops deployed at X" to generate a plan. To be able to represent these different kinds of goals as well as other planning-related aspects, Rosetta employed the PLANET ontology developed by Blythe and Gil [Blythe *et al.*, 2000]. To translate between objective-based and state-based goals, Rosetta uses a (heuristic) reformulation rule that looks for the primary effect of the planning operator describing the objective-based goal to serve as its state-based translation. Here are two of the central reformulation rules involved in this mapping:

```
(defruleset map-objective-to-state-based-goal
  {?goal-instance ??roles
   <find-equivalent-operator ?goal-instance>
   ?operator
   <get-primary-effect ?operator> ?effect}
 ==>
  (State-Based-Goal
    <map-operator-and-roles
      ?operator ?effect (??roles)>))

(defruleset find-equivalent-operator
  {?goal-instance
   <@most-specific-named-descriptions
     <retrieve-tuples all \?op
      (and (member-of ?goal-instance \?op)
           (context-of
             \?op <get-agent-model
                    <current-receiver>>)
           (exists \?effect
             (role-type primary-effects
                        \?op \?effect)))>>
   ?equiv-operator}
 ==>
  <object-name ?equiv-operator>)
```

The first thing Rosetta does is to find a planning operator in its model of the Prodigy/Analogy agent

that is a suitable match for the operator requested by ForMAT. It does so by looking for the most specific operator description that matches the description of the goal posted by ForMAT by using a PowerLoom subsumption test. After the operator is found, its primary effect is used as a state-based goal description that can be passed on to Prodigy. Once the top-level `translate-rosetta-message` rule terminates, the translated message looks like this and is sent to Prodigy:

```
(message
  (content
    (find (object plans)
          (for (State-Based-Goal
                 (is-deployed
                   (object 42nd-Batt)
                   (at Big-Town))))))
  ...)
```

Finally, the Prodigy/Analogy wrapper translates that into the following which can be sent directly to the planner:

```
(:find-plans
   (is-deployed 42nd-Batt Big-Town))
```

The Rosetta application provides a nice testbed for all aspects of OntoMorph. Syntactic rewriting is exercised in the agent wrappers, semantic rewriting is exercised to perform representation reformulations, and a mixture of both controls the scripting of the overall translation process. Furthermore, the tight integration with the PowerLoom KR system and the interpreted nature of the rewrite rules provide for a very productive, incremental development cycle.

6 Related Work

Ontolingua [Gruber, 1993; Fikes *et al.*, 1997] is an attempt to avoid the translation problem by providing a centralized ontology repository that encourages reuse, and an ontology specification language that serves as an interlingua whose representational primitives can be translated into a variety of target KR languages by special-purpose translators. However, since the generated translations cannot be controlled, modifications such as changing modeling conventions or performing semantic shifts is not possible. While avoiding translation is always a good strategy, it is not always possible such as in the case of distributed, heterogeneous agents. Using one big, centralized ontology as done by CYC has similar drawbacks. In particular, it becomes problematic when a smaller system that only relies on a portion of the ontology needs to be fielded. Another alternative to translation is the use of lifting axioms as done in [Frank *et al.*, 1999]. Lifting axioms can only be used in systems expressive enough to support them. Another drawback is that they perform translations via logical inference at query time, which could be prohibitively expensive.

Since part of OntoMorph can be viewed as a parser specification system, it is legitimate to ask how it compares to other parsing technology such as YACC, definite clause grammars, natural language parsers such as ATNs, etc. YACC parsers are only applicable to context-free languages that are LR(1), which is too restrictive for a general-purpose translation system. Natural language parsers such as ATNs could in principal be used to implement a rewrite system, but since they are geared towards parsing of natural language sentences instead of arbitrary syntax trees, the specification would be less direct and more difficult. Definite clause grammars probably come closest to our desiderata for direct and concise specification of transformation rules, however, extra support would be necessary to support certain conveniences of the OntoMorph pattern language such as sequence variables and bounded repetition of compound patterns. Additionally, the integration with a KR system such a PowerLoom would still be missing which is a crucial part of OntoMorph's utility. Similar objections hold for languages such as POP-11 which already provide some of the pattern match functionality needed by OntoMorph, but lack the combination of features and the integration with a KR system such as PowerLoom.

7 Future Work: Ontology Merging

One of the primary motivations for the development of OntoMorph was to support merging of overlapping ontologies. Merging two or more source ontologies into a merged ontology involves the following steps:

1. Finding semantic overlap or *hypothesizing alignments.*
2. Designing transformations to bring the sources into mutual agreement.
3. Editing or *morphing* the sources to carry out the transformations.
4. Taking the union of the morphed sources.
5. Checking the result for consistency, uniformity, and non-redundancy and if necessary repeating some or all of the steps above.

These steps have different degrees of difficulty and are supported to various degrees by the state of the art. For example, techniques for hypothesizing alignments have been developed during large-scale ontology merging tasks as described in [Knight and Luk, 1994; Hovy, 1998; McGuinness *et al.*, 2000], and consistency checking is already fairly well supported by today's KR systems. Designing the necessary transformations is probably the most difficult and least automatable task, since it involves understanding the meaning of the representations. Additionally, this step often involves human negotiation to reconcile competing views on how a particular modeling problem should be solved.

At the center of every merging operation is step 3, since before ontologies can be merged they have to be transformed into a common format with common names, common syntax, uniform modeling assumptions, etc., which always involves some of the transformation operations described in Section 3. Since merging is an iterative process, it is very important that these transformations can be specified easily and carried out repeatedly and automatically with a tool such as OntoMorph. This is even more important in the context of tracking changes to one of the sources in a later re-merge. Without a clear and executable specification of the transformations used in the initial merge, much of the merging work has to be redone by hand. By using a tool such as OntoMorph, many of the necessary transformation rules will be reusable as is, and only the changed and extended portions of the modified source ontology will require adapted or new rewrite rules.

8 Conclusion

We presented the OntoMorph translation system for symbolic knowledge. OntoMorph provides a powerful rule language to represent complex syntactic transformations, and it is fully integrated with the PowerLoom KR system to allow transformations based on any mixture of syntactic and semantic criteria. OntoMorph facilitates the direct and concise description of transformations to solve knowledge translation needs along a multitude of dimensions. OntoMorph's has been successfully applied as an an input translator for a critiquing system and as the core of a translation service for communication among heterogeneous agents.

Acknowledgements

This research was supported by the Defense Advance Research Projects Agency under Air Force Research Laboratory contracts F30602-97-1-0194 and F30602-97-1-0195. Many thanks to Yolanda Gil and Bob MacGregor for their comments on earlier versions of this paper.

References

[Allen, 1984] J.F. Allen. Toward a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[Blythe *et al.*, 2000] J. Blythe, Y. Gil, H. Chalupsky, and R.M. MacGregor. Supporting translation among planning agents. Internal Project Report, USC Information Sciences Institute, 2000.

[Chaudhri *et al.*, 1998] V.K. Chaudhri, A. Farquhar, R. Fikes, P.D. Karp, and J.P. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 600–607, Menlo Park, July 26–30 1998. AAAI Press.

[Cohen *et al.*, 1998] P.R. Cohen, R. Schrag, E. Jones, A. Pease, A. Lin, B. Starr, D. Easter, D. Gunning, and M. Burke. The DARPA High Performance Knowledge Bases project. *Artificial Intelligence Magazine*, 19(4):25–49, 1998.

[Cox and Veloso, 1997] M.T. Cox and M.M. Veloso. Controlling for unexpected goals when planning in a mixed-initiative setting. In E. Costa and A. Cardoso, editors, *Proceedings of the Eighth Portuguese Conference on Artificial Intelligence (EPIA-97)*, volume 1323 of *LNAI*, pages 309–318, Berlin, 1997. Springer.

[Fikes *et al.*, 1997] R. Fikes, A. Farquhar, and J. Rice. Tools for assembling modular ontologies in Ontolingua. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 436–441, Menlo Park, July 27–31 1997. AAAI Press.

[Frank *et al.*, 1999] G. Frank, A. Farquhar, and R. Fikes. Building a large knowledge base from a structured source. *IEEE Intelligent Systems*, 14(1):47–54, 1999.

[Genesereth, 1991] M.R. Genesereth. Knowledge interchange format. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 599–600, San Mateo, CA, USA, April 1991. Morgan Kaufmann Publishers.

[Gil, 1994] Y. Gil. Knowledge refinement in a reflective architecture. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 520–526, Menlo Park, CA, USA, July 31–August 4 1994. AAAI Press.

[Gruber, 1993] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Aquisition*, 5(2):199–220, 1993.

[Hovy, 1998] E.H. Hovy. Combining and standardizing large-scale, practical ontologies for machine translation and other uses. In *Proceedings of the First International Conference on Language Resources and Evaluation (LREC)*, Granada, Spain, 1998.

[Knight and Luk, 1994] K. Knight and S.K. Luk. Building a large-scale knowledge base for machine translation. In *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1*, pages 773–778, Menlo Park, CA, 1994. AAAI Press.

[Lenat, 1995] D. Lenat. CYC: A Large Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11):32–38, November 1995.

[MacGregor, 1991] R.M. MacGregor. Inside the LOOM description classifier. *ACM SIGART Bulletin*, 2(3):70–76, 1991.

[McGuinness *et al.*, 2000] D.L. McGuinness, R.E. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*, San Francisco, CA, 2000. Morgan Kaufmann.

[Mulvehill and Christey, 1995] A. Mulvehill and S. Christey. *ForMAT – a Force Management and Analysis Tool.* MITRE Corporation, Bedford, MA, 1995.

[Shapiro and Rapaport, 1992] S. C. Shapiro and W. J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, January–March 1992.

[Smith, 1990] D.C. Smith. *Plisp User's Manual.* Apple Computer, August 1990.

[Sowa, 1992] J. F. Sowa. Conceptual graphs as a universal knowledge representation. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 75–93. Pergamon Press, Oxford, 1992.

[Tesler *et al.*, 1973] L. Tesler, H. Enea, and D.C. Smith. The Lisp70 pattern matching system. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 671–676, Standford, CA, August 1973. William Kaufmann.

[Valente *et al.*, 1999] A. Valente, T.A. Russ, R.M. MacGregor, and W.R. Swartout. Building and (re)using an ontology of air campaign planning. *IEEE Intelligent Systems*, 14(1):27–36, 1999.

[Veloso *et al.*, 1995] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

[Veloso, 1994] M.M. Veloso. *Planning and learning by analogical reasoning*, volume 886 of *LNAI*. Springer, New York, NY, 1994.