Overcoming Ontology Mismatches in Transactions with Self-Describing Service Agents

Drew McDermott
Yale University
drew.mcdermott@yale.edu

Mark Burstein
BBN Technologies
burstein@bbn.com

Douglas Smith
Kestrel Institute
smith@kestrel.edu

Abstract. One vision of the "Semantic Web" of the future is that software agents will interact with each other using formal metadata that reveal their interfaces. We examine one plausible paradigm, where agents provide *service descriptions* that tell how they can be used to accomplish other agents' *goals*. From the point of view of these other agents, the problem of deciphering a service description is quite similar to the standard AI planning problem, with some interesting twists. Two such twists are the possibility of having to reconcile contradictory *ontologies* — or conceptual frameworks — used by the agent, and having to rearrange the data structures of a message-sending agent so they match the expectations of the recipient. We argue that the former problem requires human intervention and maintenance, but that the latter can be fully automated.

1 Introduction

Suppose an agent is given the task of buying the paperback edition of "Robo Sapiens" for less than $25.

The agent must carry out several tasks:

1. Find other agents that might be able to help carry out the given action. (A *broker agent* would perform this part.)
2. For each such agent, get a description of what service it provides. This description must be expressed in a formal language, such as DAML (DARPA Agent Markup Language).

3. If the goal description and the service description do not use the same ontology, find a common framework to translate them to. An *ontology* is a "conceptual scheme," a way of talking about the world.[1]

4. Find and execute a *plan* for satisfying its goal, that is, a series of interactions with a given bookseller that result in the agent acquiring a copy of the book. The primitive actions of the plan will be actions that send and receive messages. Building and decoding these messages may require further translation, between what one agent wants to receive and what the other knows.

One of the key questions we address in this paper is how agents' goals and servers' service description can be represented, and what is necessary to make the two mesh. Many treatments of such problems assume that representations can be as simple as lists of keywords and values

```
(''Task: buy; Thing-to-buy: book; Price: (< $25); ....'')
```

Such notations work fine as long as all tasks fit within a preimagined framework, but are unable to express anything novel.

We prefer to use notations that respect the degrees of freedom we're likely to require in the future. It seems inescapable that such notations will have the power of formal logic:

```
(do-for-some (λ(m - Merchant b - Book)
               (and (= (title b) "Robo Sapiens")
                    (sells m b)
                    (< (price m b) (* 25 $))))
             (λ(m - Merchant b - Book)
               (buy-from m 1 b)))
```

`(do-for-some` p a`)` means, "For some object(s) x satisfying predicate p, do `(`a x`)`." We use Lisp-style notation for logical constructs. Function application is written `(`*function* arg_1 `...`arg_n`)`, even if the *function* is traditionally written using infix notation. So `(* (+ 3 4) 5)` is the Lisp way to write `(3+4)*5`.[2]

The notation `(`λ `(`*params*`)` e`)` denotes a function whose parameters are *params* and whose value is e. We use the term *body of* the λ-expression to refer to e. Although it's not our emphasis in this paper, all expressions must be *typable*, meaning that it must be possible to assign consistent types to all their subexpressions. When necessary for typability or perspicuity, parameters can have declared types, indicated using the notation `(`λ `(...`*param* `-` *type*`...) ...`). λ-expressions have many purposes. The first λ-expression in our example is a predicate, because its body is of type `Proposition`. The second denotes a function from merchants and books to actions, so that applying it to a particular merchant and book yields a particular action, namely, buying one copy of that book from that merchant. The

[1] Original meaning: the philosophical study of being. As used in AI, the word "ontology" has come to mean "what is represented as existing."

[2] We depart from Lisp notation in two contexts. We represent finite sets using braces and tuples using angle brackets. Lisp purists may prefer to read {`a, b, c`} as `(set a b c)`, and `<a b c>` as `(tuple a b c)`.

combination of `do-for-some` and λ work together to define a "quantifier" for actions, analogous to the usual existential quantifier $\exists(x \in S)P(x)$ in mathematical logic. The action `(do-for-some` p a`)` is carried out whenever the agent does `(`a x`)` for some x satisfying p. There is no presupposition that it achieves this by, say, finding an x that satisfies p, then doing `(`a x`)`. In the present case, it might search for a plan for `(buy-from m96 1 b97)`, where `m96` and `b97` are placeholder constants labeled with the constraints that `b97` be *Robo Sapiens*, and that `m96` be a merchant that sells `b97` for less than $25. Or it might pursue it in some other way entirely; the logic doesn't care.

In this paper, we focus on the question how these logic-based representations can be used, and in particular what happens after brokers have done their work, so that two or more agents know of each other's existence and possible usefulness. At that point the task becomes getting the agents to talk to each other in order to solve a common problem. For clarity, we will adopt the following terminology: the *planning agent* is the one whose point of view we are taking, i.e., the buyer in our example; the *target agent(s)* are those the planning agent is trying to interact with. We assume the target agents are not under our control. They share some of the notational assumptions we make, but we must take their notations as we find them.

2 Using Self-Describing Agents

One of our notational assumptions is that each target agent will have a *service description* embedded in the interface it presents to the world, which one may visualize as a web page. This description will have an internal and an external form. The external form is "web-friendly," in the sense that it looks like XML, and, when appropriate, can be displayed and browsed through. Such a language is under development under the label "DARPA Agent Markup Language," or DAML (`http://www.daml.org`), which is an extension of RDF, the Resource Description Framework.

(See `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.)

So what we have been writing as

```
(book-isbn book21 "0-262-13383-0")
```

might be encoded on the web more like this:

```
<rdf:Description about=''#book21''>
        <pub:book_isbn>0-262-13383-0</pub:book_isbn>
</rdf:Description>
```

However, these are simply two alternative syntaxes for the same thing, which is represented internally as an abstract syntactic object.

The first hurdle to overcome is that the two agents must "speak the same language." Two different booksellers (e.g., Amazon.com and Barnes & Noble) must use the same industry-specific vocabulary in their service descriptions. If they don't, then we have an *ontology translation problem*, an issue we'll address in section 3.2,

making only two remarks here: (1) Within an industry there will be strong motivation to adopt a standard vocabulary, as is indeed already happening with XML; (2) the main place the translation problem will arise is when satisfying a request requires interaction of agents from multiple communities.

Assuming for now that the service description is in the same language as our service request, what we have to do is verify that there is a way of carrying out the request by talking to the target agent. (In general, we may have a collection of target agents to talk to, but we'll ignore that.)

This kind of verification is close to what AI researchers call a *planning problem*: Given a description of a system, an initial state of the system, and a goal, find a sequence of actions that achieve the goal in that system. Here the service description plays the role of system description and initial state. Once the action sequence has been found, during the *planning phase*, it must be executed. During this *plan execution* phase, the actions are executed in order. It is reasonable (we hope) to assume that the planning agent will succeed if it executes the plan; but there may well be situations where the plan exits prematurely with some sort of failure indication. In that case the agent may give up, or *replan*, starting from the situation it finds itself in halfway through the original plan.

Let's look at an example of planning and execution, involving a fictional bookseller we'll call "Nile.com." One thing you can do at Nile.com's web page is find out if they have a book in stock. Nowadays this is done by using the search facility, and visually inspecting the output, looking for phrases such as "In stock, usually ships within 24 hours." In an agent-oriented world, actions such as filling in a form and pushing a button will have dual descriptions in terms of agents sending messages. Similarly, outputs will be defined in terms of formal languages, as well as being displayable for human consumption.

We will formalize this by having `send` and `receive` actions:

- `(send` *agent message*`)`: Send the given message to the given agent; creates a message id that the sending agent can use to identify replies.
- `(receive` *agent message-id*`)`: Receive a message, sent in reply to the original sender's message.

The message to the bookseller is of the form `(search {<`key_1`, `val_1`>, ..., <`key_k`, `val_k`>})`.[3] The response is a list of "book descriptions," giving important information about each book that matches the search keys. These descriptions will also be in an XML dialect, but as usual we will use a more compact notation.

So the plan we are looking for might begin:

```
(series (tag s1
           (send Nile.com
                 (test-in-stock
                      <<author "Philip K. Dick">
                       <title "Ubik">>)))
        (tag r2 (receive Nile.com (value s1))))
           (test (= (value r2) empty-set)
```

[3] As before, what's actually sent is a piece of XML. This is an ongoing area of research; W3C's effort is described at `http://www.w3.org/MarkUp/Forms/`.

```
                (fail (not-in-stock ...))
                ...))
```

In this plan, the `tag`s allow us to give names to steps in the plan. The `value` of a step is the result it returns. The value of `s1` is a message id that later `receive`s can refer to. The value of `r2` is the set of tuples received in answer to the `in-stock` query.

To formalize this in terms a planner can understand, we create *action definitions* such as

```
(:action (send ?a - agent ?msg - Message)
     :vars (?id - Message-id)
     :value ?id
     :effect (reply-pending ?a ?id ?msg))

(:action (receive ?a - agent ?id - Message-id)
     :vars (?msg - Message)
     :precondition (reply-pending ?a ?id ?msg)
     :effect (forall (?d - (Lst (Tup Attribute String))
                     ?sv - Message)
                (when (and (= ?msg (test-in-stock ?d))
                           (this-step-val ?sv))
                      (know-val (has-in-stock ?a ?d)
                                ?sv))))
```

This is an extension of PDDL (Planning Domain Definition Language) notation, which is in standard use in the AI planning world[8, 9]. The details of the notation are not important here, but the gist is that sending a message creates a message id, so that a later reception can know what it's a response to. In addition, in the case where the message sent was an "in-stock" inquiry, one result of the action is that the planning agent knows whether the target agent has the book in stock. In other words, by executing this action the planning agent will have acquired new information.

This way of representing the effects of `receive` is too clumsy for practical use, because to be realistics the effect specification would have to list the effects of all the possible `send`s that the `receive` could be in answer to. A better idea is to have assertions of the form

`(message-exchange` *message-id*
sent-message
received-message
effect`)`

and have the `:effect` field of `:receive` consult these assertions:

```
:effect (when (and (this-step-val ?sv)
                   (message-exchange ?id ?smsg ?sv ?e))
              ?e)
```

Obviously, an AI planner can solve problems involving actions that acquire information only if it can reason about situations in which it doesn't already know everything. As it happens, many planning algorithms, including some of the most

efficient, cannot. They require it to be the case that the initial state of the world, the set of possible actions, and the effects of every action are all known. The only uncertainty is which action sequence will bring about a desired result. There has been much research on relaxing these assumptions, but no approach that is obviously correct.

Fortunately, the version of the problem we are confronted with is not as bad as the general case, because our agent knows at planning time exactly what it will and will not know at plan-execution time. In addition, we can avoid tackling extremely general formalizations of what it means for an agent to know something. For automated agents, we can appeal to the difference between computable and noncomputable terms. A term is *computable*[4] if it can be "evaluated," yielding a canonical term for an object of its type. For instance, the term `(+ 5 4)` is computable, because we can hand it to a programming-language processor and get back `9`. We will use the term *computational* for a term like `9` that is canonical in the sense alluded to, meaning that it can take part in further computations using standard algorithms. We write `(val (+ 5 4) 9)`, where `val` is a variant of equality that applies only to computable terms and their computational values. By contrast, `(number-of-planets sun)`, while it may also happen to denote nine, is not a computational representation of nine in the way the term `9` is. It is not even computable, because we cannot simply ask a Lisp system to evaluate `(number-of-planets sun)` and expect to get back `9`.[5]

A plausible principle for agents is

> *To know something is to have a computable term whose value is (a computational representation of) that something.*

We formalize this principle by introducing predicates expressing what the planning agent knows. (We currently do not provide for reasoning about what the target agents know; we believe that there is little symmetry between the two cases, because even if the planning agent believes that a target agent has a computable term denoting something, the planning agent won't know what that term is or how to evaluate it.)

One such predicate is `(know-val` e r`)`, which means that the agent knows the value of expression e, and that the value is the value of computable term r. For example, the agent might record

```
(know-val (book-isbn book21) (value step14))
```

meaning that `(val (value step14)` s`)` if and only if s is a string giving a legal ISBN (International Standard Book Number) for `book21`. Here we make use of the fact that after a plan step p has been executed, `(value` p`)` is a computable term.

We will also require a predicate `(know-val-is` e r v`)`, which is roughly equivalent to

[4]We adopt this term with some hesitation, because its usual meaning has somewhat different connotations. However, we can't think of a better one.

[5]Of course, there may be programs, say, a front end to a database of astronomical facts, in which one can do exactly this; in that context the term `(number-of-planets sun)` *would* be computable.

`(and (know-val` e r`) (val` r v`)))`

except that the planner will avoid trying to make such a goal true by changing the value of r.

A computational term representing a finite set is the familiar $\{r_1, \ldots, r_k\}$, where r_i is a computational term representing the i'th element of the set. Sometimes it is sufficient for an agent to have a partial listing of a set. To represent that situation, we have two further predicates

- `(known-elements` S r`)`: Meaning that r is a computable term whose value is a computational representation of the set of all objects the planning agents knows to be elements of S.
- `(known-elements-are` S r $\{r_1, \ldots, r_k\}$`)`: In which the elements are spelled out.

2.1 Proposed Planning Algorithm

Most previous work in the area of planning with incomplete knowledge—so-called *contingent planning*—has been done in the context of partial-order planning [4, 11, 12]. This fact is mainly a historical accident, because work on planning with incomplete knowledge happened to coincide with a period when partial-order planning was popular.

We are adding a contingent-planning ability to our Unpop planner [7], which is in the family of *estimated-regression search* planners [1]. These systems build a plan by starting with a null series of actions and adding actions to its trailing end. At each stage, it attempts to add the action that will maximally reduce the estimated effort required to finish the plan. The effort is estimated by constructing a tree of subgoals that relates the original goal to the current situation. The branches of the tree are simplified versions of the actual sequence of actions that will be required to solve the problem. The tree, called the *regression graph*, can be computed efficiently, but is only a heuristic estimate of the actual actions required, because many interactions between actions are ignored.

To handle contingent planning, Unpop must be modified thus:

1. The output of the planner can't be a simple sequence of actions; it must include `if-then-else` tests that send execution in different directions based on information gathered.
2. As a consequence, the space searched by the planner cannot be a simple space of action sequences. One alternative would be to let the space be the set of "action trees," each branch of which corresponds to a sequence. However, this idea has a couple of bugs that we will discuss below.
3. Given a goal of the form `(know-val` e `...)`, the planner must either verify that the planning agent already knows e, or find an action whose `value` can be be used to construct e. For a goal of the form `(send` a m`)`, the planner must verify that it knows, or can come to know, sufficient information to build m

To deal with this last issue, the planner must index actions by the values they compute, in much the way that planners traditionally index them by the effects they

can bring about. However, there are some differences. There will seldom be an exact alignment between what the planning agent knows and what it needs to know. For instance, if the value of an action `(ask-name ?c)` is `<(last-name ?c) (first-name ?c)>`, and the planning agent wants to know the last-name of `D`, it will have a goal `(know-val (last-name D) ?r)`. The term `(last-name D)` can be extracted from the action value by using the function `(elt` t i`)`, which gets the i'th element of a tuple t. So all the planner has to do is propose the action `(ask-name D)`, which will have, among other things, the effect `(know-val (last-name D) (elt (value` S`) 1))`, assuming S is the step with action `(ask-name D)`. A bit of care must be taken here to ensure that S is a placeholder for the correct step, which of course doesn't exist yet. We discuss this issue at greater length in section 3.1.

Let's look more closely at the search-space issue. As we said above, the most straightforward idea is to think of a partially constructed plan as a tree of actions, with the branch points occurring after information-gathering steps. A plan is completed successfully when every branch leads to a successful conclusion. One bug with this idea is that it may be asking too much to require every branch of a plan to succeed. Often there is a "normal" result of an information-gathering step, such that it is reasonable to expect the normal result to occur. If it might not occur, the best thing for the planner to do is tack on a short branch saying "Give up!" The resulting branching plan has one branch that succeeds and one that fails, which is perfectly all right. If Nile.com might not have your book, that is no reason to give up on the attempt to deal with them. Hence rather than require all branches to succeed, we require just one to succeed, hopefully the most likely one.

Another problem has to do with efficiency. Suppose a plan has a branch point fairly early, leading to subplans P_1 and P_2. In general, the planner has to do a search through different partial versions of P_1 and P_2. Suppose it eventually finds versions P_{11}, P_{12}, ..., P_{1m} of P_1, and versions P_{21}, ... P_{2n} of P_2. Using the tree representation, we must represent these as mn distinct trees. The numbers m and n might be around 50 in a realistic case, so we have 2500 different plans to think about. Worse, the computation the planner does for, say, $P_{1,23}$ is the same regardless of whether it is paired with $P_{2,13}$ or $P_{2,32}$, so the planner will have to do the same work over and over.

The best search space therefore turns out to be the one we started with: a set of sequences of steps, each representing a partial plan. The only difference is that each sequence may be annotated with zero or more *knowledge notes* recording what the planner will have learned at various points in the sequence. There is also a difference in what the planner must do when a complete sequence is found. It now may discard all the competing plans that reflect the same knowledge notes, and keep working on plans that represent other knowledge states. For instance, the planner may find a plan for buying a book assuming that there is a paperback edition. Having found it, it may continue to look for a plan to handle the case where it discovers that there is no paperback edition.

When the planner runs out of patience, it returns however many branches it can cobble together. If during execution it diverges from the branches it predicted would succeed, it must replan. In some such cases, the new information it has will allow it to find a good plan; but many times the problem will just not have a solution.

So far, we seem to be assuming that service descriptions contain specifications of the effects of individual atomic transactions with the server. These are indeed important, but in practice many servers will also provide "scripts," that is, standard sequences of transactions that can be used to accomplish common goals.

For instance, a bookseller might provide a script for the action `(buy-from` m n d`)`, meaning "Buy n copies of something answering to the description d." (To keep things simple, we suppress the price argument we used earlier.) That script might look like:

```
(:method buy-from
    :params (?m - Merchant
             ?quant - Integer
             ?d - Item-description)
    :vars (?r - (Set ISBN) ?isbn - ISBN)
    :precondition
        (and (forall (x) (if (?d x) (is Book x)))
             (know-val-is (image book-isbn
                                 (set-of-all ?d))
                          ?r {?isbn}))
    :expansion
        (series (send ?m
                      (verify-in-stock ?isbn))
                ...))
```

The notation `(set-of-all ?d)` is the set of all objects matching description `?d`. In traditional notation that would be written $\{x|$`(?d ?x)`$\}$. The function

`(image` f l`)`

creates a list with elements `(`f l_0`)`, `(`f l_1`)`, ..., `(`f l_{n-1}`)`, so

`(image book-isbn ...)`

changes a list of books into a list of their ISBNs, a computational object.

The idea behind scripts is that if the planning agent just wants to carry out the action `(buy-from ...)`, or any action that fits one of the scripts, it can save searching for a plan by just finding and tuning the appropriate script. (Tuning might include filling in actions to achieve goals for which the script supplies no action.)

This style of planning is usually called *hierarchical*, because the problem is to instantiate hierarchies of actions using large building blocks rather than assembling sequences of individual actions. Hierarchical planning is fairly well understood, and tends to be efficient when it is applicable at all (because the script writer has done most of the work already). There is an interesting research question here about how to get a planner to do both hierarchical and sequential planning. Our approach will be to augment the notion of partial plan to include partially expanded scripts as well as open goals, but the focus of this paper is on agent-communication issues, so we won't go into this any further. However, we do point out that the goal we started with, `(do-for-some ...(`λ`(...) (buy-from m 1 b)))`, is actually an

action rather than a *propositional goal*, so we've been assuming that actions are part of problem specifications all along.

3 Ontology and Data Structure Translation

It's time we turned to our principal topic, which is how to cope with ontology and data-structure mismatch. We begin with the latter.

3.1 Glue Code

Assuming that the planning agent and target agent use the same ontology, there is still a potential mismatch problem. Suppose that the planning agent is dealing with a book seller that offers a discount if you order 10 or more books, not counting bulk orders. Somewhat artificially, let's suppose that the planning agent is responsible for sending the total at some point. That is, the planner contemplates executing the action:

```
(send G (non-bulk-total
           (size (set-of-all
                    (λ (b)
                      (intention (buy-from G 1 b)))))))
```

This looks complex, so let's break it down into parts.

```
(set-of-all (λ (b) (intention (buy-from G 1 b))))
```

is the set of all books `b` such that the planning agent intends to buy exactly 1 copy of `b` from `G`. The function `non-bulk-total` is a constructor that builds a message to send to the target agent — a computational object.

Obviously, the planning agent should know what it plans to buy. Using the principle of section 2, that means it must have a computable term for it. Suppose the following is true in the initial situation:

```
(know-val (image (λ(b k)
                   <(name (author b)) (title b) k>)
                 (set-of-all
                     (λ (b k)
                       (intention
                           (buy-from G k b))))
          pending-orders)
```

This formula states that the computable variable `pending-orders` contains (by stipulation) a set of triples <*author title quantity*> for every book the planning agent intends to buy some quantity of. Let's explain that more gradually. The `set-of-all` expression here is similar to the one we need to send, except that it denotes a set of tuples <b k> for every book b that the agent plans to buy k copies of. These tuples are not computational, but we can convert it to something that is by using `image`. While a book or an author is an abstract object in a universe of discourse, the name of the book or author is just a string, and the number of copies the agent intends to buy is represented as a sequence of binary digits. Furthermore, the use of `know-val` announces that the variable stored in `pending-orders` is

computable, and its value will be a purely computational object, namely an ordinary tuple holding two strings and an integer.

The message the agent needs to send, and the data it has in its possession, are tantalizingly closely related, but not identical. We need a procedure that translates from what the agent knows to what it needs to send. We call such a procedure *glue code*, because it connects two things together. In [2] we discussed how to generate glue code automatically; the same approach will work in this context, with some minor modifications to the assumptions we make about the source side. In the original paper we assumed that the things the agent "knows" are strung together in a tuple; now we posit that these entities are the values of an unordered collection of computable terms, of which only a subset may be relevant to building a particular data structure.

Space does not permit us to explain in detail how the algorithm works. We treat the glue-code-generation problem as finding a computable function f such that

(*f "things agent knows"*) = *"things agent needs"*

The right-hand side is called the *target*, the arguments to f are called the *source*. The algorithm operates by transforming the target until it contains only terms that appear in the source, in which case f can be produced by λ-abstraction (replacing terms with variables).

The output of the algorithm in our example should be

```
(non-bulk-total
    (size (filter (λ (b k) (= k 1))
                  pending-orders)))
```

The value of

(filter p l)

is a copy of list l containing just the elements satisfying predicate p. In this context, it means that we discard from `pending-orders` all the tuples corresponding to bulk orders.

The planning context adds another dimension to the problem of glue-code generation. In addition to the computable terms that the planner knows about, it must also entertain the possibility of generating new computable terms of the form (`value` *step*), where *step* is a new step added to the plan. The open research question is how to fit this into the computation of the regression graph.

3.2 Ontology Translation

We now turn to the most difficult problem that web-based agents must cope with, the problem of reconciling disparate ontologies, or representational frameworks. The reason it is so difficult is that it often requires subtle judgments about the relationships between the meanings of formulas in one notation and the meanings of formulas in another. Furthermore, there is no obvious "oracle" that will make these judgments. For instance, we cannot assume that there is an overarching (possibly "global") ontology that serves as a court of appeals for semantic judgments. There are times when such a strategy will work, but only after someone has provided a

translation from each of the disparate ontologies to the overarching framework, and there is no reason to expect either of these translation tasks to be any easier than the one we started with. Indeed, the more the overarching framework encompasses, the harder it will be to relate local ontologies to it. Hence the work of ontology reconciliation inevitably involves a human being to do the heavy lifting. The most we can hope for is to provide a formal definition of the problem, and software tools[6] to aid in solving it.

The goal of these tools is to develop and maintain *ontology transformations*. An ontology transformation is a mechanism for translating a set of facts expressed in one ontology (O_1) into a set of expressions in another ontology (O_2), such that the new set "says the same thing" as the original set.

Ontology translation is partly a matter of syntax and partly a matter of logic. The logical issues include:

- *Vocabulary:* What symbols does the ontology use and what do they refer to?
- *Expressiveness:* What logical constructs are allowed?

The expressiveness issue may not sound ontological, but it can be. For instance, if the ontology allows us to talk about possible truth, it may commit us to assuming the existence of possible but nonactual worlds in which propositions false in this world are true.

In addition to such purely logical issues, computational questions about how facts are structured and accessed are often mixed into the ontology question. Examples:

- *Implicit content:* What facts are represented implicitly in a given formalism? For instance, if the formalism allows a list of objects at a certain point, does it imply that the list comprises all the objects with a certain property?
- *Indexing:* How are facts associated with "keys" so that they can be retrieved when necessary? Specifically, is every fact associated with a class of object it is true of?
- *Efficiency:* Is the language restricted in such a way as to make some class of inferences more efficient?

Past work in the area of ontology transformation [6, 3] has addressed both logical and computational issues. We think it is more enlightening to separate them out. From the point of view of logic, computational issues affect mainly the *concrete syntax* of an ontology. Therefore it ought to be possible to find an abstract version O_a of any ontology O, such that any set of facts expressed in O can be translated into a set of facts in O_a. Furthermore, all abstract ontologies use the *same* syntax, so that there is no longer any need to mix syntactic and computational issues into logical ones. In other words, we assume that an ontology transformation $O_1 \rightarrow O_2$ can always be factored into three transformations $O_1 \rightarrow O_{1a} \rightarrow O_{2a} \rightarrow O_2$. This may not seem like an improvement at first, but it has some advantages. First, it allows us to focus on abstract→abstract transformations, and put syntax on the back burner. Second, the translation $O \rightarrow O_a$ should not be very difficult, because it is essentially a matter of "parsing" a set of facts; going in the other direction, $O_a \rightarrow O$

[6] Such as those described by [10].

is a matter of "generating" the concrete representation of a set of facts. Third, the transformation $O \leftrightarrow O_a$ has to be done just once for each ontology.

One might object that not all the content of a set of facts can be pulled out and made into explicit formulas, and therefore that our decomposition, however tidy, will not work in practice. We take this objection seriously, but for now our principal reply is that for ontologies in which it is valid the transformation problem is not very well defined no matter what approach you take to it.

Hence we will continue to employ our tactic of focusing on abstract rather than concrete data structures. We will assume that all facts are expressed in terms of *formal theories*, each of which we take to contain the following elements:

1. A set of *types*.
2. A set of *symbols*, each with a type.
3. A set of *axioms* involving the symbols.

In addition we introduce the concept of a *dataset*, that is, a set of facts expressed using a particular ontology. This concept abstracts away from the actual representations of, say, Nile.com's current inventory, and treats it as a set of identifiers and facts about them, which uses symbols from that ontology.

Once we have cleared away the syntactic underbrush, the ontology-transformation problem becomes much clearer. One is likely, in fact, to see it as trivial. Suppose one bookseller has a theory O_1 with a predicate `(in-stock x - Book t - Duration)`, meaning that `x` is in stock and may be shipped in time `t`. Another bookseller expresses the same information in its theory O_2, with two predicates, `(in-stock y - Book)` and `(deliverable d - Duration y - Book)`. We are presented with a dataset D_1 that is in terms of O_1, which contains fragments such as

```
(:constants ubik blade-runner - Book)
(:axioms (in-stock ubik (* 24 hr))
         (in-stock blade-runner (* 4 day))
         ...)
```

To translate this into an equivalent dataset that uses O_2, we must at least find a translation for the axioms. The types and constants need to be handled as well, but we'll set that aside for a moment. We will use the notation $D_1 \rightarrow D_2$ as a mnemonic for this sort of transformation problem.

With this narrow focus, it becomes almost obvious how to proceed: Treat the problem as a deduction from the terms of one theory to the terms of the other. That is, combine the two theories by "brute force," tagging every symbol with a subscript indicating which theory it comes from. Then all we need to do is supply a "bridging axiom" such as

```
(forall (b t) (iff (in-stock₁ b t)
                   (and (in-stock₂ b)
                        (deliverable₂ t b))))
```

which we can use to translate every axiom in D_1. More precisely, we can use it to augment the contents of D_2. Any time we need an instance of `(in-stock₂ x)` and `(deliverable₂ y x)`, the bridging axiom will tell us that `(in-stock₂`

`ubik)` and `(deliverable`$_2$ `(* 24 hr) ubik)` are true (and maybe other propositions as well). We then discard the subscripts, and we're done. Furthermore, elementary type analysis tells us that `ubik` is of type `Book`$_2$.

This idea is similar to the *lifting axioms* of [5]. The main difference is that they focused on axioms of the form `(if (`*axiom in one domain) axiom in another)*, whereas we use `iff`. The reason for the difference is that we are interested in inferring facts of the form `(not (in-stock`$_2$ `x))`; we could avoid this sort of inference if we could rely on a closed-world assumption for the predicate `in-stock`.

Of course, the deductive approach does not solve all problems. Here is a list of some of the remaining issues:

1. It is potentially reckless to reduce ontology transformation to theorem proving. In the example, the required deduction was easy, but in general it could be undecidable, after finding zero, one, or two axioms, whether there are any more. However, we are inclined to think that most of the theorem-proving problems that arise during ontology translation are straightforward.

2. We attached subscripts to predicates and types, but not to other identifiers. That implies that we can just take a symbol like `ubik` over to the target theory. But suppose the target dataset must be compatible with some existing O_2 dataset, and the symbol `ubik` is already in use. In principle the deductive framework can accommodate this situation, by including a test for whether `ubik`$_1$ and `ubik`$_2$ refer to the same object, i.e., whether we can prove `(=` `ubik`$_1$ `ubik`$_2$`)`. It is often easy to show that they are not equal, by showing that they are of different types. But suppose we can't prove either that the two identifiers are equal or that they are unequal. What do we do then? Also, do we have to test all pairs of symbols for equality? (Two symbols could easily be provably equal even though they are spelled differently.)

   We glossed over similar problems with variables and types, when we wrote `(forall (x y) ...)`, implying that `x` and `y` could live in both ontologies. We may want to allow that as a special case, but in the general case it is necessary to provide transformations for the values of variables. To modify our example somewhat, suppose that the types of the arguments of `deliverable` are actually `Integer` and `Book`, so that `(deliverable 24` b`)` means that b ships within 24 hours. But let's also suppose that the symbol `Book` happens to denote exactly the same sort of thing in both domains. Then our bridging axiom might become:

```
(forall (b - Book
         t1 - Duration1 t2 - Integer)
    (if (= t1 (* t2 hr)))
        (iff (in-stock1 b1 t1)
             (and (in-stock2 b2)
                  (deliverable2 t2 b2)))))
```

   Note that equality and Integer are not domain-specific. (Put another way, there is a standard ontology where such general-purpose things live, and all other ontologies inherit from it.)

3. As has been observed before, two ontologies often carve the world up differently. They may have different "granularity," meaning that one makes finer distinctions than the other; of course, O_1 might make finer distinctions than O_2 in one respect, coarser distinctions in another.

The last issue is likely to be the most troublesome. Here's an example: Suppose O_1 is the ontology we have been drawing examples from, a standard for the mainstream book industry. Now suppose O_2 is an ontology used by the *rare* book industry. The main difference is that the rare-book people deal in individual books, each with its own provenance and special features (e.g., an autograph by the author). Hence the word "book" means different things to these two groups. For the mainstream group, a book is an abstract object, of which there are assumed to be many copies. If a customer buys a book, it is assumed that he or she doesn't care which copy is sent, provided it's in good condition. For the rare-book industry, a book is a particular object. It may be an "instance" of an abstract book, but this is not a defining fact about it.

For example, if you buy Walt Whitman's *Leaves of Grass* from Amazon.com, you can probably choose from different publishers, different durabilities (hardcover vs. paperback, page weight), different prices, and various other features (scholarly annotations, large print, spiral binding, etc.). However, you certainly can't choose exactly which copy you will receive of the book you ordered; and you probably can't choose which poems are included, even though Whitman revised the book throughout his life. The versions in print today include the last version of each poem included in any edition.

If you buy the book from RareBooks.com, then there is no such thing as an abstract book of which you wish to purchase a copy. Instead, every concrete instance of *Leaves of Grass* must be judged on its own merits. Indeed, making this purchase is hardly a job for an automated agent, although it could be useful to set up an agent to tell you when a possibly interesting copy comes into the shop.

Let's look at all this more formally. Suppose that the planning agent uses the industry-standard ontology (O_2), and the broker puts it in touch with RareBooks.com, with a note that although it bills itself as selling books, its service description uses a different ontology (O_1). If after trying more accessible sources the planning agent's goal can't be achieved, then the broker may search for an existing ontology transformation that can be used to translate RareBooks's service description from O_1 to O_2.[7]

Let us sketch what some of the bridging axioms between O_1 and O_2 might look like. In particular, we need to infer instances of $(\texttt{is Book}_2\ x)$ given various objects of type $\texttt{Book}_1$ with various properties. Objects of type $\texttt{Book}_2$ we will call *commodity books*; an example is the Pocket Books edition of *Mein Kampf*. Objects of type $\texttt{Book}_1$ we will call *collectable books*; an example is a copy of *Mein Kampf* once owned by Josef Stalin. It is roughly true that many, but not all, rare books can be thought of as instances of particular commodity books. Two rare books are instances of the same commodity book if they have the same publisher, the same title, the "same" contents, and the same characteristics (e.g., hardcover, large print,

[7] If it can't find one, all it can do is notify the maintainers of the ontologies of the problem; there is no way for the broker, the planning agent, or the end user to find a transformation on the fly.

and such).[8] We can produce the following bridge axioms:

```
(:functions (book-type x - Book1) - Book2)
(:axioms (forall (b11 b21 - Book1)
             (iff (and (= (publisher1 b11)
                          (publisher1 b21))
                       (= (title1 b11) (title1 b21))
                       (= (phys-charac1 b11)
                          (phys-charac1 b21))
                       (< (revision-dif1 b11 b21) 1.5))
                  (= (book-type b11) (book-type b21))))
         (forall (b1 - Book1)
             (= (buy1 b1)
                (buy2 (book-type b1)))))
```

This should all be self-explanatory, except for the predicate `revision-dif`, which we suppose is in use in the rare book business to express how many revisions are found between an earlier and later copy of an author's work. We have introduced a new function `book-type`, which maps individual collectable books to their types, which are commodity books.

For axioms such as these to do the planning agent any good, it must be possible for the planning agent to use them to translate a rare-book dealer's service description. Suppose the agent is trying to buy a copy of *Lady Chatterly's Other Lover*, a little-known[9] sequel to D.H. Lawrence's famous work. Having exhausted the usual sources, it attempts to deal with RareBooks.com. The planning agent first translates the service description, so that all actions are in terms of `(book-type` b`)` instead of b. Assuming it can find a way to carry out its plan, at the last stage it must translate its messages back into talking about collectable books. This requires producing glue-code in the combined axiom set. Similarly, the first step in deciphering a message from the target agent is to apply glue code to rearrange the data structures into something the planning agent can decode.

4 Conclusions

Here are the main points we have tried to make:

1. Interagent communication requires a sophisticated level of representation of knowledge states, action definitions, and plans.
2. This representation can only be logic-based; no other notation has the expressive power. Embedding this logic in some form of XML/RDF/DAML notation is a good idea for web-based agents, but puts nontrivial demands on the representational power of those notations.
3. In spite of the expressivity, there are algorithms for manipulating logic-based expressions that might overcome computational-complexity problems.

[8] An easy way to tell if they are the same would be to check if they have the same ISBN, but the ISBN system has been in effect for only thirty years, so it won't apply to many rare books.

[9] in fact, fictitious

4. In particular, planning algorithms are a natural fit to the idea of a *service description*. The service description specifies the possible interactions with an agent; a plan is a sequence of interactions to achieve a specific goal. Finding such plans is more or less what planning algorithms do.
5. Planning algorithms will, however, have to be extended in various ways, in order to cope with disparities between what it knows and what the target agent wants to receive.
6. There are two key disparities that must be dealt with: ontology mismatches and data-structure mismatches. The former requires human management of a formal inter-theory inference process. The latter requires automatic generation of "glue code" to translate data structures.

This is obviously work in progress. We are in the process of adapting our Unpop planner to handle hierarchical and contingency planning, and connecting it to the glue-code generator. We are building the architecture for managing ontology transformations.

Acknowledgements: This work was supported by DARPA, the Defense Advanced Research Projects Agency. Thanks to Dejing Dou for input.

References

[1] B. Bonet, G. Loerincs, and H. Geffner. A fast and robust action selection mechanism for planning. In *Proc. AAAI-97*, 1997.

[2] M. Burstein, D. McDermott, D. Smith, and S. Westfold. Derivation of glue code for agent interoperation. In *Proc. 4th Int'l. Conf. on Autonomous Agents*, pages 277–284, 2000.

[3] H. Chalupsky. Ontomorph: A translation system for symbolic logic. In *Proc. Int'l. Con. on Principles of Knowledge Representation and Reasoning*, pages 471–482, 2000. San Francisco: Morgan Kaufmann.

[4] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proc. Third International Conf. on Knowledge Representation and Reasoning*, pages 115–125, 1992. Morgan Kaufmann.

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

[6] T. Gruber. Ontolingua: A Translation Approach to Providing Portable Ontology Specifications. *Knowledge Acquisition* , 5(2):199–200, 1993.

[7] D. McDermott. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems*, pages 142–149, 1996.

[8] D. McDermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science, 1998. (CVC Report 98-003).

[9] D. McDermott. The 1998 Ai Planning Systems Competition. *AI Magazine* , 21(2):35–55, 2000.

[10] P. Mitra, G. Wiederhold, and M. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *Proc. of Conf. on Extending Database Technology (EDBT 2000)*, 2000.

[11] M. Peot and D. Smith. Conditional nonlinear planning. In J. Hendler, editor, *Proceedings of the First International Conf. on AI Planning Systems*, pages 189–197. 1992.

[12] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *J. of Artificial Intelligence Research* , 4:287–339, 1996.