to appear in Journal of Autonomous Agents and Multi-Agent Systems, 2001
(short version in Proceedings of the Agents 2000 Conference, Barcelona, Spain, May 2000)

Derivation of Glue Code for Agent Interoperation[1]

Mark Burstein, GTE/BBN
Drew McDermott, Yale University
Douglas R. Smith, Kestrel Institute
Stephen J. Westfold, Kestrel Institute

Abstract

Getting agents to communicate requires translating the data structures of the sender (the *source* representation) to the format required by the receiver (the *target* representation). Assuming that there is a formal theory of the semantics of the two formats, which explains both their meanings in terms of a neutral *topic* domain, we can cast the translation problem as solving higher-order functional equations. Some simple rules and strategies apparently suffice to solve these equations automatically. The strategies may be summarized as: decompose complex expressions, replacing topic-domain expressions with source-domain expressions when necessary. A crucial issue is getting the required formal theories of the source and target domains. We believe it is sufficient to find partial formalizations that grow as necessary.

1 Introduction

This paper is about getting agents to communicate with each other. By "agent" we mean programs that operate at a high enough semantic level that they can form new connections to other programs in order to get a job done. To make such a connection, an agent must find other agents that might carry out a task on its behalf, and then establish a dialogue with them. Several researchers have examined facets of this interchange, including how agents might search for each other [13], how they might communicate once they have linked [7], and what "speech acts" they might employ [2]. However, the most pressing problem is getting them to *speak the same language.* This is our focus here.

Suppose one agent, A, needs a certain fact, and agent B can supply it. Assuming that some previous "brokering" or "advertising" phase has brought the two agents together, there remains the problem that the way A represents facts and the way B represents them are probably not compatible. It is necessary to interpose a translation program between the two. We call this *glue code.* The problem is to generate glue code automatically.

For example, suppose we are given a scheduling agent S that produces an airlift schedule. For each aircraft, the schedule gives the sequence of flights that it is scheduled to make.

[1]This work was supported by the Air Force Research Lab (Rome, NY) and the DARPA CoABS program under contracts F30602-98-C-0168 and F30602-98-C-0169.

We also have an agent R that maintains what is known as a "commitment matrix," a table that specifies, for each time slot, the number of each type of aircraft that are committed to scheduled flights (i.e. not free for allocation) in that time slot. The problem is to derive a translator f from schedules to commitment matrices, so that R is able to accept the information derived from the scheduler according to its declared input specification format (API).

Some commonly considered approaches to this problem are to:

- Engineer the agents to be compatible in advance by changing one or the other to accept or generate the form required by the other.
- Attempt to develop a "general purpose" translation agent that will convert all messages that can be produced by agents using one semantic model or ontology into equivalent messages using a second ontology, to the extent that translations between those ontologies are well defined.

We are developing a third approach, namely, to submit the specification of the source and target messages to an agent that produces a very specific translator for that purpose. This has the potential advantage of being much more efficient if similar messages will be sent frequently once the agents have been "introduced", or when the data to be passed in a single message contains a large number of similar forms.

This approach to the problem can be considered as a form of the *automatic programming* problem, but one that we believe is simpler than the general case. It "feels like" an exercise in moving data around, with a bit of condensing, summarization, and totaling thrown in. On the other hand, problems like this one are not trivial. The reader may wish to stop and try to produce the glue code for S and R by hand.

In what follows, we will describe our framework, and illustrate with several examples. Although, we do not have a full implementation of our approach as yet, we have developed several prototypes that can handle a variety of examples like those described in this paper. At the end of the paper, we will talk about opportunities and challenges in automating the process more fully, and in applying it in a distributed agent environment.

In addition to the agent-communication work we mentioned at the outset, much work has been done by the database community on the problem of translating between databases, where it is called the problem of *schema integration,* with subproblems of *query translation* and *value translation.* [10, 5, 3, 9]. The main differences are:

1. Database researchers assume that the main problem is to translate queries (and their results) from one formalism to another. Queries are written in a standard language such as SQL[4], and the only issue is how the relation and argument names are mapped. We want to be able to translate an arbitrary formula (or functional expression) from one formalism to the other.

2. Database researchers assume that the results of a query are tables in a standard format. Hence if you can find a translation of a query you automatically can translate the result. We will tackle the more general case of automatically generating data-structure-translation code given expressions that describe what one agent wants and what another can produce.

2 The Setting

To explain our approach, we will start with a problem that *is* almost trivial. Suppose that we have two agents that use data structures that represent the same population of people, but that record different information about them. In particular, let S record the name, identification number, and age of each person, and let T record the name and the id number of the person if he/she is over age 30 and 0 otherwise. S is the output of agent 1, and T is the input to agent 2. If agent 1 is to feed information to agent 2, then we must derive a translator f such that $f(S) = T$. We call S the *source* domain, and T the *target* domain.

This problem is solvable only if S and T capture "the same" information, or, more precisely, if all the information to be included in T occurs in S. But by itself a database captures no information at all. It's just a meaningless pile of symbols unless its users understand what the symbols are supposed to mean, and use the database in accordance with that understanding. Such an informal notion of meaning is sufficient if the users are all humans, but not in the case we're concerned with, where the glue-code generator is supposed to be automatic.

Hence we must supply every agent with a formal semantics for its input and output representations. This may seem like a big commitment, but we see no way to evade it. We'll avoid being distracted by this issue here, but return to it in Section 7.

A key requirement of the formal semantics for agents is that if two agents are to be connected, then their semantics must *overlap*. If one of the agents contains the symbol `person`, and the other contains the symbol `employee`, then there must be a semantic domain whose topic is people, and the symbols must be defined in roughly the same terms from that domain. It must be possible to infer, for instance, that if a data structure in one domain represents information about `employees`, it can be translated into a data structure in the other that represents information about `persons`. This requirement raises many tough issues, which again we defer for later discussion. We will simply assume that there is an abstract common theory of the application domain that includes a dataset P with the property that we can define S and T in terms of P. The common theory of the application domain provides symbols for the concepts, operations, and relationships in the domain. Its axioms constrain the meaning of the vocabulary. In our example, we specify an abstract theory, called a *spec*, of persons P of sort[2] $set(person)$ where each *person* is an object that has a *name*, *id*, *age*, and other attributes, as follows:

[2]In this paper, we use the word "type" informally, reserving the word *sort* for technical contexts.

spec *Person-DB* **is**
sort *Person*
op $P : set(Person)$
op $name : Person \rightarrow string$
op $id : Person \rightarrow nat$
op $age : Person \rightarrow nat$
... other sorts, operations, and axioms ...
end-spec

We will call the domain common to both S and T the *topic domain* and the specification above the *topic theory*. Sorts like *nat* and *string* correspond to manipulable data objects, which can be used as concrete representations of objects of sorts like *Person*. As we will see, agents cannot manipulate pure topic-domain entities directly, but can only perform computations over their representations.

We use a λ-calculus notation for functions. As usual in the functional-programming community, we will assume that a λ expression has just one argument, but this argument will often be a tuple. The i'th element of a k-tuple $t = (x_1, \ldots, x_k)$ will be written $\Pi_i^k(t) = x_i$. Application of function f to argument x will be written $f\,x$. So if $f : A_1 \times A_2 \times \ldots \times A_k \rightarrow B$ (i.e., f's sort is "function from k-tuples to B, where the i'th element of its argument is from domain A_i"), then rather than write $f = (\lambda\, t\ g(\Pi_1^k(t), \ldots, \Pi_k^k(t)))$, we will write $f = (\lambda\,(x_1, \ldots, x_k)\ g(x_1, \ldots, x_k))$. In some circumstances, when we write $(\lambda\, x\ \ldots)$, we intend to cover the case where x is a tuple, but we don't currently care how long it is.

Now let's suppose that S and T both represent aspects of the set P. S is a set of triples (*name, id, age*), one per person, so we can express it as

$$S = image(\lambda\, p\ (name\, p, id\, p, age\, p), P)$$

where $image(f, X) = \{f(x) | x \in X\}$. We can express S more concisely using a couple of abbreviatory devices. *Function tupling* allows us to write

$$(name,\ id,\ age) \text{ for } \lambda\, p\ (name\, p, id\, p, age\, p)$$

and we write $F \star X$ for $image(F, X)$. So we can write S as

$$S = (name, id, age) \star P$$

Similarly,

$$T = (name, \lambda\, p \textbf{ if } age\, p > 30 \textbf{ then } id\, p \textbf{ else } 0) \star P$$

The problem is to find a function f such that $f(S) = T$. This type of equation will occur repeatedly in what follows, so we write it as

$$\begin{aligned} &Find : f((name, id, age) \star P) = \\ &\quad (name, \lambda\, p \textbf{ if } age\, p > 30 \textbf{ then } id\, p \textbf{ else } 0) \star P \end{aligned}$$

It seems clear that we can construct f by creating a function that does something to each element of the input dataset S. In other words, if we could solve:

$$\begin{aligned} &\mathit{Find} : i((\mathit{name}, \mathit{id}, \mathit{age})(x)) = \\ &\quad (\mathit{name}, \lambda p \ \textbf{if} \ \mathit{age}\, p > 30 \ \textbf{then} \ \mathit{id}\, p \ \textbf{else} \ 0)\, x \end{aligned}$$

where x is an arbitrary element of P, then the solution to the original equation would be

$$f = \lambda X \ i \star X$$

We express this as a subgoal-formation rule:

Image-Decomposition Rule
Given $g : A \to B$,
and $h : A \to C$,
To find: $f(g \star Y) \ = \ h \star Y$
Solution : $f = \lambda X \ i \star X$
$\Longleftarrow$ *Find:* $i(g\, x) = h\, x$ for arbitrary $x \in X$.

We use the symbol " $\Longleftarrow$ " to indicate subgoaling.

Because we're trying to find an i that works for all elements x of P, we can treat x as an arbitrary constant from now on. The subgoal may now be rewritten as

$$\mathit{Find} : i(\mathit{name}\, x, \mathit{id}\, x, \mathit{age}\, x) = (\mathit{name}\, x, \textbf{if} \ \mathit{age}\, x > 30 \ \textbf{then} \ \mathit{id}\, x \ \textbf{else} \ 0)$$

To make further progress, we need the concepts of *imitation* and *projection* from higher-order matching theory [6], embodied in the following rules:

Imitation Rule
Given $g : B_1 \times \ldots \times B_n \to C$
To *Find:* $f\, u = g(v_1, \ldots, v_n)$
Solution: $f = \lambda x \ g(h_1\, x, h_2\, x, \ldots, h_n\, x)$
$\Longleftarrow$ *Find:* $h_i\, u = v_i$ for $1 \leq i \leq n$

Projection Rule
To *Find:* $f(a_1, \ldots, a_k) = a_i$
Solution: $f = \Pi_i^k = \lambda(x_1, \ldots, x_k) \ x_i$

That is, we let the structure of f mirror the desired structure on the right-hand ("target") side of the equation.

In our example, imitation with $g = (name,\ \lambda p\ \textbf{if}\ age\, p > 30\ \textbf{then}\ id\, p\ \textbf{else}\ 0)$ allows us to solve for i thus:

$$\begin{array}{rcl} i & = & \lambda\,(n,d,a)\ (h_1(n,d,a), h_2(n,d,a)) \\ \Longleftarrow & & \mathit{Find} : h_1(name\, p, id\, p, age\, p) = name\, p \\ & & \text{and } \mathit{Find} : h_2(name\, p, id\, p, age\, p) = \textbf{if}\ age\, p > 30\ \textbf{then}\ id\, p\ \textbf{else}\ 0 \end{array}$$

The projection rule then gives us $h_1 = (\lambda\,(n,i,a)\ n)$. Imitation applied to h_2 gives

$$\begin{array}{rcl} h_2 & = & \lambda\,(n,i,a)\ \ \textbf{if}\ h_{21}(n,i,a) \\ & & \qquad\qquad\qquad\ \textbf{then}\ h_{22}(n,i,a) \\ & & \qquad\qquad\qquad\ \textbf{else}\ h_{23}(n,i,a) \\ \Longleftarrow & & \mathit{Find} : h_{21}(name\, p, id\, p, age\, p) = (name\, p > 30) \\ & & \text{and } \mathit{Find} : h_{22}(name\, p, id\, p, age\, p) = id\, p \\ & & \text{and } \mathit{Find} : h_{23}(name\, p, id\, p, age\, p) = 0 \end{array}$$

A few more rounds of imitation and projection eventually yield:

$$f = \lambda\, X\ (\lambda\,(n,i,a)\ (n,\ \textbf{if}\ a > 30\ \textbf{then}\ i\ \textbf{else}\ 0)) \star X$$

and $T = f(S)$. In English, to translate source dataset S into target dataset T, translate all the triples in S, leaving the name alone, and combining the age and id as shown.

3 More Complex Examples

If this example seems too simple, here is how we would represent the commitment-matrix problem we described above. We will need some more notational machinery. Let $x \mapsto y$ be the ordered pair (x, y), considered as a component of a finite map; it records the association between element x and the element y it maps to. Let $P \lhd U = \mathit{filter}(P, U) = \{u \in U | P(u)\}$, i.e., the subset of U satisfying predicate P. Let $(\mathit{size}\, U)$ be the cardinality of finite set U.

The topic-domain spec describes schedules:

spec *Schedule-DB* **is**
import *set, cargo, integer, port*
sort *Reservation, time*
sort-axiom $time = integer$
op $R : set(Reservation)$
op $aircraft : Reservation \to string$
op *flt-no* $: Reservation \to string$
op $depart : Reservation \to time$
op $arrive : Reservation \to time$

op $manifest : Reservation \rightarrow set(cargo)$
op $origin : Reservation \rightarrow port$
op $destination : Reservation \rightarrow port$
op $aircraft\text{-}class : Reservation \rightarrow string$
...
axiom $\forall(r1, r2 \in R)$
$r1 \neq r2 \wedge (aircraft\, r1) = (aircraft\, r2)$
$\supset ((arrive\, r1) \leq (depart\, r2) \ \vee \ (arrive\, r2) \leq (depart\, r1))$
end-spec

where integers are used to represent time. The last axiom states that no two flights by the same aircraft overlap.

The source agent is a mission planner MP that produces a description of all reservations in the form:

$$S = (aircraft, depart, arrive, manifest) \star R$$

The target agent requires a summarization of the schedule that indicates how many aircraft are flying at a given time over a given time interval. This summary, known as a *commitment matrix*[3], is

$$T(i,j) = (\lambda t \ \ t \mapsto size((\lambda r \ \ (depart\, r) \leq t < (arrive\, r)) \lhd R)) \star \{i..j\}.$$

(The notation $\{i..j\}$ denotes the set of all integers between i and j inclusive.)

The goal is to derive a translator from an arbitrary schedule *Sched* to a commitment matrix[4]:

$$Find : f(S,i,j) = T(i,j).$$

A translator from S (domain DB) to T (domain MP) might be derived as follows[5]:

Find: $f((aircraft, depart, arrive, manifest) \star R, i, j)$
$= (\lambda t\ t \mapsto size((\lambda r \ \ (depart\, r) \leq t < (arrive\, r)) \lhd R)) \star \{i..j\}$
Solution: $f = \lambda\,(X,i,j)\ (\lambda t\ (g(t,X,i,j))) \star h(X,i,j)$ by imitation.
$\Longleftarrow$
Find: $g(t,\ (aircraft, depart, arrive, manifest) \star R,\ i, j)$
$= t \mapsto size((\lambda r \ \ (depart\, r) \leq t < (arrive\, r)) \lhd R)$
and *Find:* $h(t,\ (aircraft, depart, arrive, manifest) \star R,\ \ i, j) \ = \ \{i..j\}$

[3] A commitment matrix is supposed to give the number of each *type* of aircraft that is in use. For clarity, we have simplified this to a a "commitment vector" that gives the total number of aircraft flying for each time period.

[4] Note that the parameters i and j control how much of S must be used to generate T. For technical purity, we could write this equation as $f(i,j)(S)$, but here and elsewhere we will move freely between expressing a multiargument function with currying and expressing it with tupling, generally favoring the latter.

[5] Technically, the first subgoal produced should be of the form *Find:* $g((\ldots) \star R, i, j) = (\lambda t\ t \mapsto \ldots)$, but we move the t into the arguments of g by the "decurrying" rule alluded to above.

By straightforward imitation and projection we derive

$$h = \lambda\,(t, X, i, j)\ \{i..j\}$$

The other subgoal is attacked using imitation again:

Solution: $g = \lambda\,(t, X, i, j)\ g_1(t, X, i, j) \mapsto g_2(t, X, i, j)$
$\Longleftarrow$
Find: $g_1(t, (aircraft, depart, arrive, manifest) \star R, i, j) = t$
and
[G*] *Find:* $g_2(t,\ (aircraft, depart, arrive, manifest) \star R,\ i, j)$
$= size((\lambda r\ \ (depart\, r) \leq t < (arrive\, r)) \lhd R\ )$

The g_1 part succumbs to projection, with solution $g_1 = (\lambda\,(t, X, i, j)\ t)$. But if we attempt to apply imitation to g_2, we will get something useless:

Solution attempt: $g_2 = \lambda\,(t, X, i, j)\ size(h_{bug}(t, X, i, j))$
$\Longleftarrow$
Find: $h_{bug}(t, (aircraft, depart, arrive, manifest) \star R, i, j) = \ \ (\ldots) \lhd R$

The problem is that $(\ldots) \lhd R$ is of sort $set(Reservation)$, a topic-domain sort, whereas h_{bug} has access only to the source data, $(\ldots) \star R$. There is no reason why h_{bug} could not exist as a mathematical abstraction, but it can't be the denotation of a computable process, because computations output data structure that *represent* reservations, not the reservations themselves. We must find a way to transform the target side of goal [G*] so that it is expressed in terms that can be found in the source representation.

4 Sourcifying Rules

The key heuristic we employ is to replace occurrences of target-domain terms with their images under functions mentioned in the source sides of goals. This heuristic is licensed by the following rule:

$\star$-Introduction Rule
For any $m : A \to B$ and $C \subseteq A$
$C = m^{-1} \star m \star C$
$\Longleftarrow$ m is injective

If we have a problematic occurrence of a topic term C, and we know that $m \star C$ occurs in the source term S, then we try replacing C with $m^{-1} \star m \star C$. The hope is that the m^{-1} part can be simplified away by further transformations, leaving us with a term $(m \star C)$ that

is closer to what is available on the source side. A rule of this kind that pushes a target-side term closer to being identical with a source-side term is called a *sourcifying* rule.

The following rule can be used to remove an inverse function that is introduced by the $\star$-Introduction Rule.

Inverse-Composition Rule
For any $f : B \to C$
and injective $g : B \to A$
To *Find:* $h\,x = f(g^{-1}\,x)$
$\Longleftarrow$ *Find:* $h(g\,y) = f\,y$ for arbitrary $y \in B$

In other words, to find an h that is the composition of f and g^{-1}, find an h such that $f = h \circ g$.[6] To understand the rule, note that, since g is injective, for any $x \in A$, there is a unique $y \in B$ such that $x = g\,y$. Then $h\,x = h(g\,y) = f\,y = f(g^{-1}\,x)$.)

All the previous rules matched an entire goal of the form "*Find:* f $\ldots = \ldots$," producing a solution (and possibly some subgoals). We call them *solution rules.* The image-introduction rule matches a particular subterm of a goal, allowing us to produce a new goal by substituting another term for that subterm (again possibly accompanied by new subgoals). We call rules with this behavior *rewrite rules.* Unless stated otherwise, they are applied left-to-right. That is, $\alpha = \beta$ is used to rewrite terms matching α with terms matching β. Rewrite rules that do not "sourcify" are labeled with the catch-all phrase *simplification rule.* Although in the end we will require dozens of simplification rules, for the purposes of our commitment-matrix example we need only two:

Image-Filter Rule
For any $p : A \to boolean$
and any $f : A \to B$
$p \lhd (f \star X) \;=\; f \star (p' \lhd X)$
$\Longleftarrow$ *Find:* $p'\,x \;=\; p(f\,x)$ for all $x \in X$

"You can switch the order of a filter and image operation if you adjust the type of the filtering predicate."

Size-Image Rule
For any $f : A \to B$
$size(f \star X) = size(X)$
$\Longleftarrow$ f is injective

[6] This looks a little different from the previous solution rules, because the h found in the subgoal happens to be identical to the h the rule finds. So rather than write *Solution:* $h = h'$, and $\Longleftarrow$ *Find:* $h'(g\,y) = \ldots$, we just write the rule as shown.

"If f is injective, it preserves the size of any set it operates on."

Now we derive the commitment-matrix generator by proceeding as shown in Figure 1 with our problematic subgoal.

Putting it all together, we have:

$$f = \lambda(S, i, j)\ ((\lambda t\ t \mapsto (size(\lambda(a, d, r, m)\ d \leq t < r) \lhd S))\ \star\{i..j\})$$

In this example, the Size-Image Rule served as "*deus ex machina*" to cause the function

$$(aircraft, depart, arrive, manifest)^{-1}$$

to vanish completely. In general, the process isn't so abrupt or tidy. Typically several rounds of simplification are necessary to eliminate all such vestiges of the topic domain.

5 Search Management

Of course, there is a big gap between a bunch of higher-order equations and an algorithm for applying them. It is out of the question to fill this gap with a general-purpose higher-order theorem prover, because the search space for even small problems is enormous. However, for this application we believe that there is considerable guidance available. The strategy we followed in the examples above may be summarized thus:

> *To solve $f\ S = T$:*
> Use imitation, projection, and simplification rules whenever the result would consist of operations on source- or target-domain entities.
> When it wouldn't, eliminate a topic-domain term t in the target T by replacing it with a term that may be schematized as $\sigma^{-1}(\sigma\, t)$. Here $\sigma\, t$ stands for a functional expression that appears in the source specification S, and σ^{-1} stands for whatever "upstream adjustment" is necessary to preserve the meaning of T.
> Continue simplifying with imitation and projection. During this process, mark $\sigma(t)$ as *encapsulated,* so that it can be incorporated into other terms, but cannot be decomposed.

We use the term *Source-Substitution Principle* for this idea of replacing t by $\sigma^{-1}(\sigma\, t)$.[7] The principle suggests arranging sourcifying rules by what sorts of σ they handle. Here are two more. The $\lhd$-Introduction Rule is used when $\ldots \lhd t$ occurs in S and t occurs in T. The Representative-Element Rule is used when a set X occurs in S, and only an element $x \in X$ occurs in T.

[7] We have noticed the resemblance of this technique to the "rippling" technique of inductive theorem provers [1], in which the object is to transform a deductive goal into a form that matches an induction hypothesis. Our "encapsulations" correspond roughly to the "wave fronts" of the rippling mechanism. We don't know how deep the resemblance is.

Find: $g_2(t, (aircraft, depart, arrive, manifest) \star R, i, j)$
$= size((\lambda r\ (depart\, r) \leq t < (arrive\, r)) \lhd R)$

Use $\star$-Introduction to get:

Find: $g_2(t, (aircraft, depart, arrive, manifest) \star R, i, j)$
$= size((\lambda r\ (depart\, r) \leq t < (arrive\, r))$
$\lhd\ (aircraft, depart, arrive, manifest)^{-1}$
$\star(aircraft, depart, arrive, manifest)$
$\star R)$
$\Longleftarrow$ (by Image-Filter Rule and the definition of S)
$g_2(t, S, i, j)$
$= size((aircraft, depart, arrive, manifest)^{-1}$
$\star p' \lhd S)$
$=$ (by Size-Image Rule)
$size(p' \lhd S)$
$\Longleftarrow$ $Find : p'(a, d, r, m)$
$= (\lambda r\ (depart\, r) \leq t < (arrive\, r))$
$((aircraft, depart, arrive, manifest)^{-1}(a, d, r, m))$

Using the Inverse-Composition Rule:

$\Longleftarrow$ $Find : p'((aircraft, depart, arrive, manifest)\, r)$
$= ((depart\, r) \leq t < (arrive\, r))$

which succumbs to imitation and projection, yielding:

$p' = (\lambda (a, d, r, m)\ d \leq t < r)$

So that the solution for g_2 can be found easily using imitation and projection:

$g_2 = \lambda (t, X, i, j)\ size((\lambda (a, d, r, m)\ d \leq t < r) \lhd X)$

Note: We have left out the required proof that $(aircraft, depart, arrive, manifest)$ is injective, which follows from the "non-overlap" axiom in the Schedule-DB spec. However, it would be an unusual database that didn't satisfy injectivity in this way.

Figure 1: Derivation of problematic subfunction

$\lhd$-Introduction Rule
For any $p : A \to boolean$ and $B : set(A)$
$B = (p \lhd B) \cup (neg\ p \lhd B)$

where $neg\ p = \lambda\, x\ \neg p\, x$. This is a rewrite rule used to replace a topic-domain term B with the more complex term on the right. The subterm $p \lhd B$ is encapsulated, and the other terms must be further transformed.

Representative-Element Rule
For any $f : A \to B$
and any $g : A \to C$
$g(x) = h(only\_elt((\lambda\, y\ (d\, y) = (d(f\ x))) \lhd f \star X))$
$\Longleftarrow \quad x \in X$
and $d \circ f$ is injective over X
and *Find:* $h(f\ y) = g\ y$ for all $y \in X$.

The function *only_elt*: $set(A) \to A$ denotes the only element of its argument if its argument has one element, and is undefined otherwise: $(only\_elt\{x\}) = x$. In the context of this rule it will always be defined. What the rules says is that, to perform a computation on a topic-domain element x, you may transform it to the element of the source representation $f \star X$ that "represents" it.

This rule applies very generally, and hence is used only as a last resort. It applies to any formula at all with one or more occurrences of a term x such that $x \in X$, provided that $f \star X$ occurs in the source, for some function f, and a function d can be found that allows us to find x in $f \star X$. For example, if the source contains $(a, b) \star X$, and b is a unique identifier for every object, then the rule matches the target $size((\lambda\, u\ \ u > x) \lhd Z)$ with $g = (\lambda\, v\ size((\lambda\, u\ u > v) \lhd Z))$, and suggests rewriting the target as

$$h(only\_elt((\lambda\, (y_a, y_b)\ y_b = b\, x) \lhd ((a, b) \star X))$$

after finding an h such that $h(a\, y, b\, y) = size((\lambda\, u\ u > y) \lhd Z)$ for all $y \in X$. The idea is to use b to find the element $(a\, x, b\, x)$ of $((a, b) \star X)$ that represents x, and hope that there is enough information in that element so that $g\, x$ may be computed by computing $h(a\, x, b\, x)$.

We have worked several examples by hand using our strategy. For instance, consider the following source-data descriptor:

$S = (S_1, S_2)$
where $S_1 = (id, (\lambda\, x\ \ (\lambda\, (k, n)\ n > 0)\ \lhd (cid, (\lambda\, c\ qty(c, x)))) \star C)) \star F$,
and $S_2 = (cid, wcl) \star C$

S consists of two substructures, S_1 and S_2 that specify properties of a set of airplane flights, F, and a set of cargo-item classes C. S_1 is a table giving the number of items of each class aboard each flight as a set of tuples

$$(\textit{flight-id}\, f, \langle(\textit{cid}\, c_1, \textit{qty}(c_1, f)), (\textit{cid}\, c_2, \textit{qty}(c_2, f)), \ldots\rangle)$$

one per flight in the set F. We use the notation $\langle ...\rangle$ to indicate a list of an indefinite number of items. The pair $(\textit{cid}\, c_j, \textit{qty}(c_j, f_i))$ specifies the number of items of cargo type c_j on board flight f_i. $(\textit{cid}\, c)$ is the id of cargo type c. The entry is omitted if $\textit{qty}(c_j, f_i) = 0$. The table S_2 gives the weight of each item class as a set of tuples $(\textit{cid}\, c, \textit{cargo-type-weight}\, c)$ for every cargo type c in the set C.. In addition to *cid*, we use $\textit{id}\, f$ to refer to the id number of a flight; and $\textit{wcl}\, c$ to refer to the weight of an item of class c.

The "Absent Items" problem is to derive a table showing, for each item class, whether it occurs on any flight. The table consists of a list of tuples (*cargo-id*, *boolean*):

$$T = (\textit{cid}, (\lambda\, c\ \ \vee/((\lambda\, x\ \ \textit{qty}(c, x) > 0) \star F))) \star C$$

($\vee/X$ is **true** if and only if some element of X is **true**.)

This problem seems quite a bit harder than the examples we described above, but our method requires no search at all; every possible application of the Source-Substitution Principle is necessary.

We foresee two places where the method will break down:

1. If it is required to chain together an indefinite number of tuple elements, then some induction will be required. For example, given a relational database that shows the immediate successors of each airplane flight, transforming it to an explicit chain requires a recursive "collection" algorithm. Our technique works, but requires some insight, and so will be hard to mechanize.

2. When term rewriting becomes true deduction rather than mere simplification, the number of possible search paths could become large.

For instance, suppose you wanted to produce the first component S_1 of the source for the absent-items problems, which describes the *classes* of items on board, given a data structure that lists the *individual items* on board. S_1 is the target in this problem, and the source is composed of two tables (R_1, R_2), where R_1 is a set of tuples $(\textit{flight-id}, \langle \textit{item-id}_1, \textit{item-id}_2, \ldots\rangle)$, whose second component is a list of the ids of the items on the given flight; and R_2 is a set of tuples $(\textit{item-id}, \textit{item-class-id})$, which specifies for each item which class it belongs to. More formally:

$R = (R_1, R_2)$
where $R_1 = (\textit{id}, (\lambda\, f\ \ \textit{iid} \star \{x | \textit{aboard}(x, f)\})) \star F$
and $R_2 = (\textit{iid}, \textit{cid} \circ \textit{icl}) \star C$

where $id(f)$ is the id of flight f, $iid(x)$ is the id of item x, $icl(x)$ is the class of item x, and $cid(c)$ is the id of class c. The problem is to find g such that $g\,R = S_1$; in other words:

$$\begin{aligned}&\textit{Find: } g((id, (\lambda f\ \ iid \star \{x | aboard(x, f)\})) \star F,\\ &\qquad (iid, cid \circ icl) \star C)\\ &\quad = (id,\ (\lambda x\ (\lambda (k, n)\ n > 0) \lhd (cid, (\lambda c\ \ qty(c, x)))) \star C)) \star F\end{aligned}$$

Our method can handle this, but it requires partitioning the set of all items in R_1 (i.e., $iid \star \{x | aboard(x, f)\}$) by item class and applying the following *deductive rule:*

Sum-partition Rule
For any $f : A \to Real$
and any $Y : set(A)$
$+/\{: \ f\,x | x \in Y\} = +/\{: +/\{: f\,x | x \in s\} | s \in P\}$
$\Longleftarrow$ P is a partition of Y

In this rule, $\{: f(x)|...\}$ is the bag (multiset) of all $f(x)$ for x satisfying the condition "...". $\oplus/b$ applies associative and commutative function $\oplus$ to the elements of bag b. So $+/\{: 3, 3, 4\} = 10$.

The rule says that to add up $f(x)$ for every element of a set, find a partition and add up the subtotals for all the subsets the partition defines. Exactly when to apply deductive rules instead of imitation is the subject of our current research.

6 Glue Code Middle Agents

We have implemented a prototype glue code generation agent in a demonstration system simulating a military disaster relief scenario. In doing so, we have begun to discover and address issues of how such agents would be utilized and deployed.

In our demonstration, a glue code agent is deployed that receives requests for translation services and generates specific translation agents that the sender forwards messages through when the message is of a specific kind, and the ultimate recipient is of a specific agent type. In general, the indexing of these translators may become an issue, and various alternative schemes for, say, adding translation functions to a single middle agent responsible for interactions between two specific classes of agents might be preferable. Indeed, even for our simple case, there is a need to develop two different translation functions, one for the request and one for the response.

In one example from our scenario, an agent S uses a matchmaking service to find another agent T capable of delivering quantities of relief supplies to different locations by specified times. S standardly requests such services with a message like this:

(**distribute :cargo-source** (**:type** ⟨*cargo-type*⟩
:location ⟨*location-id*⟩
:quantity ⟨*quantity-in-lbs*⟩
:available-time ⟨*time-gmt*⟩)
:cargo-destination ((**:type** ⟨*cargo-type*⟩
:location ⟨*location-id*⟩
:quantity ⟨*quantity-in-lbs*⟩
:required-delivery-time ⟨*time-gmt*⟩)*))

That is, a single source load of cargo, available at the stated time, is to be divided and distributed to a number of sites.

The agent that is found to do these deliveries uses a different request form, where the source and destination for a single delivery is specified together with the type and quantity, but where the quantity is in a different unit of measure.

(**deliver :items** (⟨*cargo-type*⟩
⟨*source-location-id*⟩
⟨*destination-location-id*⟩
⟨*quantity-in-kilograms*⟩
⟨*available-time-local*⟩)*)

The glue code generator is established as a middle agent that accepts requests to "spawn" translator agents for particular messages between agents or classes of agents. In this case, agent S uses the description of the service found by the matchmaker to request a translator from the glue code agent. The following request expresses the above two forms in terms of topic domain sets, variables and concepts:

Topic Variables: *req-goals* : *set*(*move-cargo-goal*)
cargo-source : *cargo-availablility*
SourceForm =
('**distribute**
:cargo-source
(**:type** (*cargo-id* (*load-type cargo-source*))
:location (*location-id* (*load-loc cargo-source*))
:quantity (*quantity-to-lbs* (*load-amt cargo-source*))
:available-by (*time-to-zulu* (*load-available-time cargo-source*))))
:cargo-destination
(**image** (λ (*g*) (**:type** (*cargo-id* (*goal-type g*))
:location (*location-id* (*goal-destination g*))
:quantity (*quantity-to-lbs* (*goal-amt g*))
:required-delivery-time (*time-to-zulu* (*goal-deadline g*))))
req-goals)

TargetForm =
(**'deliver :items**
(**image** (λ (g) ((*cargo-id* (*goal-type* g))
(*quantity-to-kgs* (*goal-amt* g))
(*location-id* (*load-loc cargo-source*))
(*location-id* (*goal-destination* g))
(*time-to-local-time* (*load-available-time cargo-source*))))
req-goals))))

Note that the functions on topic variables produce typed quantities in some cases, and so the topic semantic model needs to include translation functions and substitution rules to move between quantities in different units. Our model uses abstract mounts such as *quantity* and *time* with different attributes corresponding to different units such as *quantity-to-lbs* and *quantity-to-kgs*. The rule that handles the translation of pounds to kilograms is as follows:

Weight-Translation Rule
To find: f(*quantity-to-lbs* x) = *quantity-to-kgs* x
Solution : f = *pounds-to-kgs*

This rule is an example of an ontology-specific translation rules. Included among such rules would be such things as converting from mass and acceleration to force, and inferences such as mother of father to grandmother, if those terms were variously defined in a source and target representation.

The LISP code produced for this glue problem, after running through an optimizer, looks as follows, with variable names changed for readability:

```
(lambda (msg-form)
  (let ((cargo-src (nth 4 msg-form)))
    (list 'deliver :items
          (mapcar #'(lambda (req)
                      (list (nth 1 req)
                            (pounds-to-kgs (nth 5 req))
                            (nth 3 cargo-src)
                            (nth 3 req)
                            (zulu-to-local-time (nth 7 cargo-src))))
          (nth 2 msg-form)))))
```

A larger open issue for our approach to inter-agent translation arises from a difference between the potential differences between *requests* and *responses*. It is largely taken for granted in the agent literature that advertisements of agent services will need to include the specification of message formats for messages *to* the agent so advertising. However,

there is little or no mention of the need for advertisements to include the specifications of allowable responses to requests generated by the agent. To translate responses, the glue code generation system will need to know the permissible forms those responses can have, or have direct access to the full ontology of the requesting agent, so that appropriate terms and formulas can be chosen. This is a topic under consideration in our current work extending the approach outlined in this paper.

7 Summary and Prospectus

We have sketched an approach to automatic derivation of "glue code," programs for translating the output of a *source* agent to the input representation of a *target* agent. We have argued that

1. Generating glue code requires a theory of a *topic domain,* the common subject matter of the source and target representations, and mappings from the source and target to the topic.
2. Given these mappings, the problem of glue-code generation can be solved by solving a higher-order equation for the unknown transformation function.

It might be objected that formalizing the mappings from source and target to the topic domain is just as difficult as writing glue code, and so we have traded a hard problem for another problem that might be even harder. We disagree, for a variety of reasons. For one thing, other researchers have proposed independent reasons for wanting a common "ontological" framework for agents (for a survey, see [8]). Even simple advertising and brokering schemes will need some kind of shared vocabulary in which to describe agents. So this problem is already fairly urgent.

Second, if it is impossible to supply formal theories of agent semantics, then we believe it is essentially impossible to glue agents together automatically. The only exceptions will be agents whose outputs are to be processed directly by humans. For instance, a search engine might look for agents that can give weather reports, and if it finds five such agents might send the output of all five to a screen for a human to look at. If one of the weather reports is a marine forecast, and the human is looking for an upper-atmosphere forecast, then the human will skim through it and reject it. However, we don't believe this paradigm is going to apply to the majority of agent interactions. If a human has to be in the loop at "agent-composition time," then agent composition will be unworkable.

Third, if formalization is done right, it has to be done only once per agent. Once the formalization is in place, it can support creation of glue code for multiple partner agents.

Fourth, we don't believe a commitment to formalizing domain theories is a commitment to finding a "formal theory of everything," which is fortunate, because there are good reasons to doubt that there is such a theory. We need to encourage agent producers to formalize

the representations of their agents within some convenient framework. Such frameworks must be public and well advertised, but they don't have to be completely global. All that is necessary is that, when two agents are to be glued together, they are formalized within the same framework, or within two frameworks that are intertranslatable.

The third and fourth points in this list are in conflict. To avoid having to repeat work, we want the formal framework to be as general as possible, but to keep the formalization intuitive and clear, we want to neglect as much as possible. For instance, suppose one thing an agent represents is a list of available airplanes. On the "simple" end of the formalization spectrum we have theories in which `available` is a unary predicate. On the "complex" end we have a theory of what it means for something to be available to someone for some purpose over some time interval subject to a set of use constraints. It must be possible to pick a point on the spectrum that seems to make sense, and possibly generalize it later if that point fails to support liaison with some unanticipated class of agents.

We believe that these "meta" ideas can themselves be formalized to some degree. We need the notion of a *lattice of theories,* in which $T_1 \sqsubset T_2$ if T_2 can express everything T_1 can express but not vice versa. In other words, T_2 makes distinctions that T_1 doesn't make. We think this idea can be developed using the concept of *theory morphism* [12].

The other main unknown in this line of research is how tightly the search for a transformation can be controlled. The derivation strategy that has emerged from our examples is to perform second-order matching modulo the laws of the domain theory. Matching rules are applied unless doing so would expose the terms of the topic-domain theory. At that point we use *sourcifying* rules to wrap topic terms inside constructs from the source domain. We have a preliminary implementation of this strategy that is capable of performing derivations of most of the examples in this paper in less than a second.

A key issue is acquisition of the common domain theory and the domain-specific rules. The rules of the domain theory are used to reformulate various terms to facilitate the matching process. This suggests that a means-end analysis might be useful in detecting the need for rules that translate between concepts on the left and right-hand sides of the translation equation. On a related note, one possibility is to analyze the rules of the domain theory to determine how to orient them, and to construct a proof plan. There is an opportunity here to use our (meta)knowledge of the derivation strategy (or proof plan) to query the domain-expert/user for rules that link certain terms.

Another issue, to be dealt with in the future, is how to transform glue-code programs into a more efficient form once they have been created. This goal is addressed by our past work on program transformation [11].

Acknowledgements: Tao Gui and Duško Pavlović provided helpful comments on drafts of this paper.

References

[1] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence* , 62(2):185–253, 1993.

[2] T. Finin, Y. Labrou, and J. Mayfield. Kqml as an agent communication language. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1997. .

[3] D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in Cis using semantic knowledge. *Int. J. of Cooperative Information Systems*, 1996.

[4] J. R. Groff and P. N. Weinberg. *The Complete Reference SQL*. McGraw-Hill, 1998.

[5] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Proc. Workshop on Management of Semistructured Data*, 1997. Tucson, Arizona.

[6] Gerard Huet and B. Lang Proving and applying program transformations expressed with second-order patterns. *Acta informatica 11* (1978), 31–55.

[7] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* , 13:91–128, 1999.

[8] T. Menzies. Cost benefits of ontologies. *Intelligence*, 10(3):27–32, 1999.

[9] T. Milo and S. Zokar. Using schema matching to simplify heterogenous data translation. In *Proc. Conf. on Very Large Data Bases*, pages 122–133, 1998.

[10] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. DOOD'95*, 1995.

[11] D. R. Smith. Kids: A semi-automated program development system. *IEEE Transactions on Software Engineering* , 16(9):1024–1043, 1990. Special Issue on Formal Methods, September.

[12] Srinivas, Y. V., and Jüllig, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

[13] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *J. ACM SIGMOD Record* , 28(1):47–53, 1999. In (Special issue on Semantic Interoperability in Global Information Systems,).