Inferring Data Transformation Rules to Integrate Semantic Web Services

Bruce Spencer and Sandy Liu

National Research Council of Canada
46 Dineen Drive, Fredericton, New Brunswick, E3B 9W4
{Bruce.Spencer, Sandy.Liu}@nrc.ca, http://iit.nrc.gc.ca/il.html

Abstract. OWL-S allows selecting, composing and invoking Web Services at different levels of abstraction: selection uses high level abstract descriptions, invocation uses low level grounding ones, while composition needs to consider both high and low level descriptions. In our setting, two Web Services are to be composed so that output from the upstream one is used to create input for the downstream one. These Web Services may have different data models but are related to each other through high and low level descriptions. Correspondences must be found between components of the upstream data type and the downstream ones. Low level data transformation functions may be required (e.g. unit conversions, data type conversions). The components may be arranged in different XML tree structures. Thus, multiple data transformations are necessary: reshaping the message tree, matching leaves by corresponding types, translating through ontologies, and calling conversion functions. Our prototype compiles these transformations into a set of data transformation rules, using our tableau-based $\mathcal{ALC}$ Description Logic reasoner to reason over the given OWL-S and WSDL descriptions, as well as the related ontologies. A resolution-based inference mechanism for running these rules is embedded in an inference queue that conducts data from the upstream to the downstream service, running the rules to perform the data transformation in the process.

1 Introduction

Some envision that software components can be described sufficiently so that they can be autonomously found and incorporated, can pass data that will be automatically understood and processed, can recover from faults, and can prove and explain their actions and results. The OWL-S specification for Semantic Web Services [2] has recently been released in which these components are Web Services described using the Web Ontology Language, OWL [13].

The OWL-S release 1.0 proposes to describe Web Services in three layers: the service profile, service model, and service grounding, describing respectively what the service does, how it does it, and how to access it. The service profile gives the information necessary for discovering and for combining (composing) Web Services. By convention, it may describe the services only abstractly and not in

S.A. McIlraith et al. (Eds.): ISWC 2004, LNCS 3298, pp. 456–470, 2004.

complete detail; the inputs, outputs, preconditions, and effects of the service may be advertised at a high level for faster but imperfect matching. The grounding layer is where the complete details would be found for each of the components in these objects, expressed both in OWL-S grounding and Web Service Description Language (WSDL) [12], a low-level specification. By separating the abstract from the concrete specifications, the tasks for which the service profile are used, discovery and composition, are greatly simplified, making the whole architecture more viable.

In general several Web Services can be selected to work together, but in this paper we consider the case where only two Web Services have been selected: one as the producer and the other as the consumer of a stream of data messages. The producer, or *upstream* service will deliver one input message at a time to the *downstream* service, which will consume it. We imagine that one or both of these services were not previously designed to work together, and that they were selected to work together based on their service profiles; now the two service groundings must be coupled autonomously, using descriptions of both levels as well as ontologies available. The setting can be seen as a special case of data integration assisted by semantic markup.

We consider two use cases, a simple one for converting dates from year / month / day order as given by the upstream service, into month / day / year order as required by the downstream service. This simple case allows us to explain the various steps in the proposals. In the second use case we convert flight information from one format to another, reshaping the XML tree containing the data and invoking two different kinds of conversions.

In section 2 we discuss the integration problem in light of two use cases, and give the general technique for building the rules in Section 3. Section 4 gives a presentation of inference queues used to run the rules. In the last two sections, we give related and future work and conclusions.

2 Grounding Level Integration

We intend to automatically integrate Web Services. In our setting, the integration task is done at several levels. We assume that a high-level plan has determined which Web Services can be integrated, based on descriptions of the effects produced by upstream services, and the preconditions required by the downstream one. The abstract planning level will have determined that the data produced by the upstream services provide information needed by the downstream service. Matching upstream to downstream services should be done without knowledge of minor differences in the concrete types of data to be transferred, if reasoning at this level is to be kept unencumbered by overwhelming detail. Once compatibility is established at the upper level, the task of matching at the lower level can begin. Two similar or identical descriptions at the abstract level can refer to very different descriptions at the concrete level.

Therefore, our objective is to create a message of the downstream data type from one or more upstream messages. We assume there exist ontologies describ-

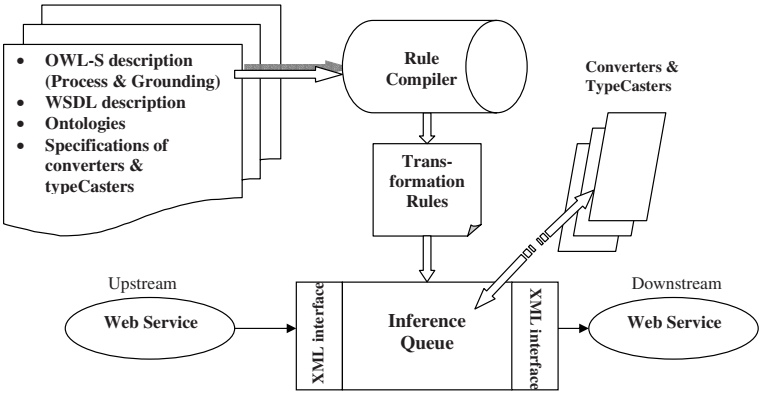

Fig. 1. System Architectural Overview

ing both the upstream messages and downstream messages, and in particular description logic, possibly OWL-S specifications containing the leaf-level parts of these messages. Figure 1 shows the overall system architecture to perform the grounding level integrations. The rule compiler takes the OWL-S descriptions, WSDL descriptions, auxiliary ontologies, and specifications of data converters as inputs, and creates data transformation rules. If the semantic integration is attempted but is not feasible, this will be indicated by a failure to create the rules.

At service invocation time, the set of transformation rules can be run to actually perform the data transformation from upstream to downstream. The intermediate carrier is called an inference queue [11], which accepts input data encoded as facts and rules in first-order logic, and can perform inferences on those facts so that both the input facts and inferred facts are transmitted as output. Such an inference queue loaded with data transformation rules at service composition time, and loaded with upstream data at runtime, will be able to convert that data to the appropriate form and deliver it to the downstream service.

Because the inference queue includes a theorem prover for definite clauses, it is capable of some of the other tasks we identified as necessary for robustness: it can detect when the upstream data is not convertible into data acceptable by the downstream service; it can also perform sanity test on the data. The OWL-S proposal allows us to express preconditions expected by the downstream service and effects expected from the upstream service. Since we may not have control over how these Web Services are written, we should attempt to verify that the expected effects were produced and expected preconditions were met. These checks would also be performed by the rule engine.

In the real world many data types that are not identical syntactically are still convertible, such as temperatures expressed in either Fahrenheit or Celsius, currency in either dollars or Yen, etc. We assume the existence of some mappings between data types, called *converters*, intended to be used for converting data

from one type to another. We also assume that low-level type conversion such as converting between string and numeric can be performed through type casters. The functions of the converters and type casters can be specified, then they can be used by the rule compilers to generate the proper transformation rules. As a result, a number of these standard conversions can be expressed as relations in the bodies of rules, and at runtime the inference engine will perform the conversion.

2.1 Use Cases

Consider the following example: Suppose a Web Service produces a date in the conventional U.S. order month / day / year. In a typical usage scenario this date would be part of a larger message, but for simplicity we consider that this is the entire contents of the message. Suppose this Web Service has been selected as the upstream service so it will provide data to the downstream Web Service. The downstream message will also be composed of just a date, but in this case in the ISO order year / month / day. The decision to connect these two Web Services in this way would be done by a planning system, designed in accordance with the current OWL-S proposal [2]. While building the abstract plan, it determined that these messages are compatible, using only the information at the OWL-S profile and process levels. An ontology of dates states that ISO and U.S. dates are both subclasses of the class date and are composed of a year, a month and a day, as in Figure 2. The planner did not look into the lower level, where the types of the date are slightly different; these differences were considered to be

```
<owl:Class rdf:ID="Date">
</owl:Class>
<owl:ObjectProperty rdf:ID="hasYear">
    <rdfs:domain rdf:resource="#Date"/>
    <rdfs:range rdf:resource="#Year"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasMonth"/>
    <rdfs:domain rdf:resource="#Date"/>
    <rdfs:range rdf:resource="#Month"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasDay"/>
    <rdfs:domain rdf:resource="#Date"/>
    <rdfs:range rdf:resource="#Day"/>
</owl:ObjectProperty>
<owl:Class rdf:ID="ISODate">
    <rdfs:subClassOf rdf:resource="#Date"/>
</owl:Class>
<owl:Class rdf:ID="USDate">
    <rdfs:subClassOf rdf:resource="#Date"/>
</owl:Class>
```

Fig. 2. A Date Ontology in OWL

```
<grounding:WsdlAtomicProcessGrounding
   rdf:ID="WSDLGrounding_CalendarMgmt_setVisitDate">
  <grounding:wsdlInputMessageParts rdf:parseType="owl:collection">
    <grounding:WsdlMessageMap>
      <grounding:damlsParameter rdf:resource="&pm_file;#year"/>
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="http://lclhst/calendar.wsdl#arg0"/>
      </grounding:wsdlMessagePart>
    </grounding:WsdlMessageMap>
    <grounding:WsdlMessageMap>
      <grounding:owlsParameter rdf:resource="&pm_file;#month"/>
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="http://lclhst/calendar.wsdl#arg1"/>
      </grounding:wsdlMessagePart>
    </grounding:WsdlMessageMap>
    <grounding:WsdlMessageMap>
      <grounding:damlsParameter rdf:resource="&pm_file;#day"/>
      <grounding:wsdlMessagePart>
        <xsd:uriReference rdf:value="http://lclhst/calendar.wsdl#arg2"/>
      </grounding:wsdlMessagePart>
    </grounding:WsdlMessageMap>
  </grounding:wsdlInputMessageParts>
</grounding:WsdlAtomicProcessGrounding>
```

Fig. 3. Snippet from the Downstream Grounding in OWL

inessential at planning time. The specific types of date from the upstream and downstream services are described in WSDL as three integers, and these integers are given more meaning by the OWL-S grounding specification. Figure 3 shows a snippet of OWL-S grounding specification for the downstream Web Service.

We compile the data transformation into a rule to be invoked to do this transformation. The messages are passed from upstream to downstream via an inference queue. An inference queue is a conduit for data that contains an inference engine. Data are considered facts, and rules are "plugged in" to tell how to infer new facts, which become the output of the queue. We illustrate the data transformation rules using a Prolog notation, where variables are shown with an upper case letter. The current prototype directly uses the XML messages, but it is possible to use RuleML [1] or SWRL [3]. Any upstream message is assumed to be given to the inference queue as a fact using the unary predicate symbol `up`, where the argument is a Prolog term, in this case containing a date in month / day / year order. The inference queue produces a message meant for the downstream service, also as a fact with the unary predicate symbol `down` and the arguments in ISO order.

The rule for data transformation in this case is

```
down(isoDate(Year, Month, Day)) :- up(usDate(Month, Day, Year)).
```

When a fact, say, `up(usDate(12, 31, 1999))` is entered to the inference queue, a resolution step is done, in which Month is bound to 12, Day to 31 and Year to

1999. The inference drawn is `down(isoDate(1999, 12, 31))`. This is expelled from the downstream end of the inference, to be delivered to the downstream Web Service.

For the second use case, suppose that a European-based Web Service producing flight information is selected to provide input to a Canadian-based Web Service accepting flight information. For the upstream (producing) Web Service the flight has three data items: a source city, a destination city and a cost quoted in Euros. The downstream service expects the flight to be described with two pieces of information: the cost in Canadian dollars and a flight manifest, which itself consists of two pieces of information, the code of the departure airport and arrival airport, respectively. In this case we make use of two converters: from Euros to dollars, and from cities to airport codes. We also need to have the translation mechanism create a message with an internal structure, the manifest object. The final rule generated for this case follows:

```
down(canadaFlight(DollarCurrency,
     manifest(DepartureAirportCode, ArrivalAirportCode))):-
  up(euroFlight(SourceCity, DestCity, EuroCurrency)),
  stringToDouble(EuroCurrency, EuroCurrencyDouble),
  euro2Dollar(EuroCurrencyDouble, DollarCurrency),
  city2AirportCode(SourceCity, DepartureAirportCode),
  city2AirportCode(DestCity, ArrivalAirportCode).
```

There are three tasks: creating the appropriate structure, converting the currency, and converting the two airport locators. Each needs to be reflected in the translation rules. The structure of the both up- and downstream messages must be reflected in the rule. In this case the downstream message, which is the argument of the `down` predicate, contains the term `canadaFlight` which itself contains two arguments, one for the dollar currency and other for the flight manifest, represented by the function symbol (or XML tag) `manifest` and containing two airport codes. So the rule constructor must be aware of and reflect all of the inner structure of both messages. These XML structures are inferred from the OWL-S grounding and WSDL information. Figure 4(a) and (b) shows these two trees representing the XML structure of the messages.

The second task is to place a call to a converter in the body of a rule. It is straightforward in the case of converting the currency amount in the upstream message to a dollar amount in the downstream message. There is a converter available, `euro2Dollar` whose specification shows that has the appropriate types; it maps Euros to dollars. In the conversion rule that is constructed, a call to this converter is included as a goal in the body of the clause. The first argument of this goal is a variable for the Euro amount that is shared within the `up` literal, as the first argument of the `canadaFlight` term. The second argument of the call to `euroToDollar` converter is a variable for the dollar amount that appears also in the `down` literal. All of these placements are straighforward because the types of the converter and the types of the message match. The arrow in Figure 4(c) from `EuroCurrency` to `DollarCurrency` was easy to place because of the type correspondences. It is not always so simple.

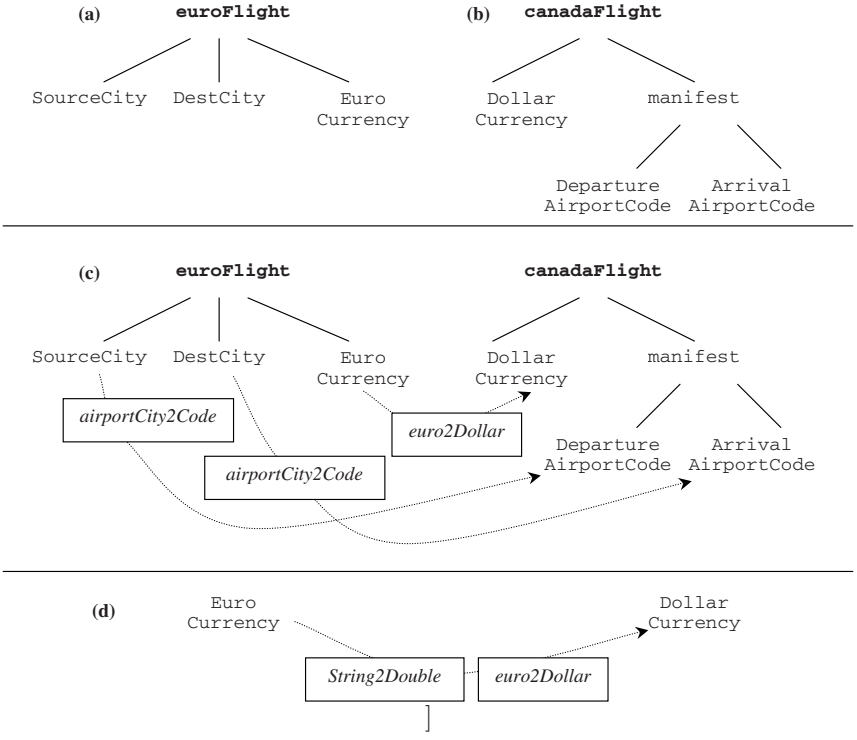

Fig. 4. Data transformation: Reshaping two XML trees

One complication arises because the output type of the converter may not match the required type. If the output type is a subtype of what is required, it must be promoted to the required type. But from a typing point of view, the converter can be used since it is guaranteed to produce a value of the required type. Alternately, if the output type is a supertype of what is required, it must be demoted to the required type. The converter may be still be useful, but a check must be done at runtime to ensure that the actual value is appropriate. On the input side of the converter, similar cases apply.

Yet the discussion so far does not fully account for the rule above. Two airport cities of different types, source and destination, need to be converted to airport codes. We want these two outputs to be of type departure and arrival airport codes, respectively. The converter, `city2AirportCode` will create data values of type airport code, but not specifically of type arrival airport code or departure airport code. Thus requires to promote the input of the converter and demote the output. But there is information lost in the promote step, which leads to confusion. (This will be illustrated in case (a) in Figure 6.) In our first attempt to write this rule, the rule compiler mixed up the types and created calls to the `airportCity2Code` converter in which the `SourceCity` was converted to the `ArrivalAirportCode`, instead of the `DestinationAirportCode`. Promoting the type from `SourceCity` to the more general type `City` causes information about the specialization to be lost.

$$\begin{aligned}
\texttt{SourceCity} &= \texttt{SourceLocation} \sqcap \texttt{City} \\
\texttt{DestCity} &= \texttt{DestLocation} \sqcap \texttt{City} \\
\texttt{DepartureAirportCode} &= \texttt{DepartureLocation} \sqcap \texttt{AirportCode} \\
\texttt{ArrivalAirportCode} &= \texttt{ArrivalLocation} \sqcap \texttt{AirportCode} \\
\texttt{SourceLocation} &= \texttt{DepartureLocation} \\
\texttt{DestLocation} &= \texttt{ArrivalLocation} \\
\texttt{EuroCurrency} &\sqsubseteq \texttt{Currency} \\
\texttt{DollarCurrency} &\sqsubseteq \texttt{Currency}
\end{aligned}$$

Fig. 5. Description Logic Axioms for Airport Example

One impractical way to solve this problem would be to provide one converter that takes a source city and creates an departure airport code, and another which converts a destination city to an arrival airport code. In this way the types of the converters would exactly match the types from the input and output messages. This is unreasonable because the programs for these two different converters would be identical, and each converter would be less useful than the original general converter. Moreover all of these converters would have to be written before the web services were encountered, defeating the goal of dealing with unforeseen Web Services.

Instead, our strategy is to create special converters dynamically from our library of general converters. The input and output types are specialized so they are closer to what is required. In this example, the concept of a `SourceCity` is actually defined as the intersection of a `SourceLocation` and a `City`, shown in Figure 5, while `DepartureAirportCode` is defined as the intersection of a `DepartureLocation` and `AirportCode`. Moreover, the set `DepartureLocation` is declared to be identical to the set `SourceLocation`. Thus the mapping from `SourceCity` to `DepartureAirportCode` can be seen as a specialization of the mapping from `City` to `AirportCode`. (This will be illustrated in case (a.1) in Figure 6.) Using this specialized mapping to convert the provided `SourceCity` gives us the right value for our `DepartureAirportCode`.

To create the correct specializations, we need to do some description logic reasoning. We have a tableau-based implementation of an $\mathcal{ALC}$ reasoner incorporated into our prototype. It is capable of searching among the ancestors (supersets) of a given concept. If two specific concepts are given and it is asked whether the first one can be converted to the second, a search is made among the superconcepts of each, to find some pair of ancestors that is most specific such that a converter is available from one to the other. In our example the ancestor types of the `SourceCity` are `SourceLocation` and `City` and the ancestors of `DepartureAirportCode` are `DepartureLocation` and `AirportCode`. We have a converter between two of these ancestors: `SourceCity` and `AirportCode`). The other two types, `SourceLocation` and `DepartureLocation`, are declared to be equivalent. From this information, the arrow in Figure 4(c) from `SourceCity` to `DepartureAirportCode` can be drawn.

Note that the rule also pays attention to the fact that the Euro currency was reported as an XML Schema string and needs to be converted to a decimal number (double) before being passed to the euro2Dollar converter. See Figure 4(d).

3 Creating Transformation Rules

The general procedure of creating the rule can now be described. A desired downstream Web Service message type is to be constructed, and we have a (set of) candidate upstream message type(s) that may provide values to be used. At higher level of abstraction, a reasoning system, such as a planner, has determined that these messages are compatible. A rule is required that at run-time creates the desired downstream message from the upstream message(s). The rule must contain a `down` conclusion in the head and this must match the structure of the downstream message, in that internal structures are reproduced within the Prolog term structure. An `up` goal with a reproduction of the upstream message is always added to the body of the rule. For each leaf-level data item in the downstream message, given its data type R_2, we need to identify a source R_1 among the data types available in the upstream message.

There are several possibilities. An upstream data type may exactly match or be a subconcept of the downstream data type: $R_1 \sqsubseteq R_2$. In that case we identify this upstream data as the source for the downstream data. For example, if R_1 is `EuroCurrency` and R_2 is `Currency` then we are assured that the value in R_1 is appropriate for R_2. Two occurrences of a shared variable are placed in the rule, one in the position of the target datum in the downstream message and one in the position in this upstream message of the selected matching datum.

An upstream data type may be a superconcept of the target downstream type: $R_1 \sqsupset R_2$. In that case the datum provided at runtime may or may not be of the appropriate type for the downstream message. In this case a check can be placed in the goal to verify that that datum a is of the downstream type, $a \in R_2$, if such a runtime check is possible. For example if R_1 is `Currency` and R_2 is `EuroCurrency` then at runtime we should check that value in R_1 is indeed a `EuroCurrency`.

The other possibilities come from the converters. A converter is defined as a mapping of data values from one type to another where both an element and its image represent the same individual. For example, a converter might map measurements in the imperial system to the metric system, but must map a quantity in one system to an equal quantity in the other. We assume that a converter C maps $S_1 \rightarrow S_2$.

Suppose there is a converter C available to generate data of the target downstream type R_2 from some available upstream type R_1; in other words by good fortune the types exactly match $R_1 = S_1$ and $R_2 = S_2$. Then add a goal to the body of the rule that calls that converter, where in loss at both endts input variable is shared with the position in the upstream message of the identified source datum, and the converter's output variable is shared with the target downstream datum.

Next suppose that the types do not match exactly. There are four cases to consider, shown as Figure 6(a-d), where the S's are either subsets or proper supersets of the R's.

Given a data value of type R_1 if it is guaranteed to be in S_1, as in cases (a) and (b), we can consider the data to have type S_1, promoting it so that we can use it as the input to the converter. If not, then $S_1 \sqsubset R_1$, and since the type is demoted, a check at runtime for being in S_1 is needed to be sure we can use this converter. This check is added as a condition of the rule. Similarly for cases (b) and (d) the output of the converter is guaranteed to be in R_2, while for cases (a) and (c), which are demotions, an additional check is needed.

However, for case (a) information is lost during promotion. It was this problem that mixed up our early prototype when it converted the `SourceCity` into the `ArrivalAirportCode`. It may be possible to retain this information, to preserve the mapping. Suppose that there exists a type T such that $R_1 = S_1 \sqcap T$ and $R_2 = S_2 \sqcap T$. Then we may assume that calling the converter with a datum of type R_1 will generate a datum of type R_2. See Figure 6(a.1). In our example from the previous section, we were able to restrict the domain and range of C by T, as follows: $S_1 =$ `City`, $S_2 =$ `AirportCode`, $R_1 =$ `SourceCity`, $R_2 =$ `DepartureAirportCode`, and $T =$ `SourceLocation` $=$ `DepartureLocation`.

If no such T exists, it may still be possible to specialize the converter. Suppose that there exist types T_1 and T_2 such that $R_1 = S_1 \sqcap T_1$ and $R_2 = S_2 \sqcap T_2$. If $T_1 \sqsubseteq T_2$ then the converted value is sure to be a member of R_2. See Figure 6(a.2). Otherwise if $T_1 \sqsupset T_2$ there is a chance that the output of the converter provided is not in R_2, and another goal should be placed into the rule checking that it is.

If there are two sources of upstream information available for the downstream type, this may mean that the data is not described in sufficient detail to make a unique determination. More semantic information may needed to disambiguate the situation. Ambiguity may also arise if there are different converters that could be applied. We can state that one converter is a better match than the other if it provides the exact types needed. However there seems to be no general technique to prefer using one converter over another. If ambiguities arise at rule generation time, several strategies could be employed: abort, ask a human, deliver a set of candidate rules, choose a probabilistically best one if a model of probability can be imposed, etc. We did not consider this problem further.

Note that in many ways the semantic integration of these Web Serices may fail and even if a rule is produced, it may not map data correctly. This could arise if the OWL-S descriptions are not sufficiently detailed, if no ontology is available to connect the data model of the upstream service to that of the downstream service, and many other reasons. We feel that more experience with marking up real life Web Services is needed before we can prescribe the sufficient conditions for producing valid data transformation rules, and the necessary conditions for succeeding to produce a rule.

In this discussion we have ignored the concrete (syntactic) data types, but at run time, when Web Service data is actually given, it is important to consider the XML Schema datatypes in which the data will delivered, either as xsd:string,

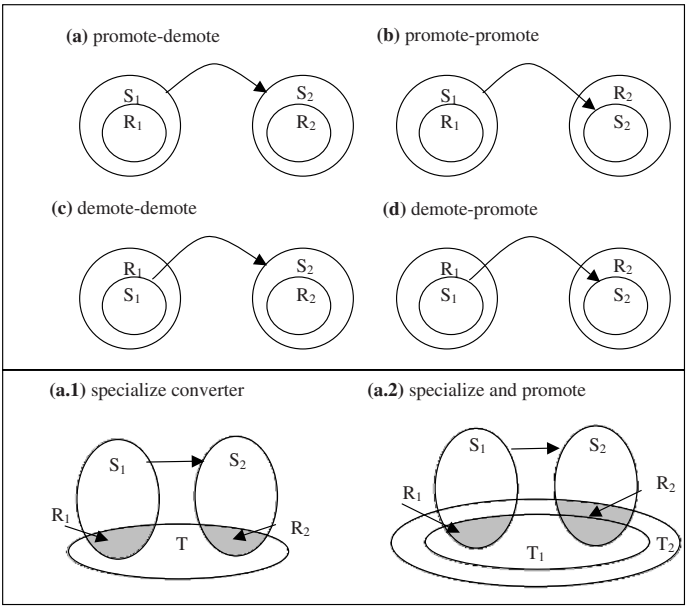

Fig. 6. Cases for specializing the converter

one of the numeric types, etc. The downstream Web Service and the converter will need to be called with the appropriate data type so a type casting (sometime called type conversion) may need to be done. Our compiler also considers this type casting and generates rules with these conversions inserted as necessary. See Figure 4(d).

The rule compiler is written in Prolog, and uses our Prolog implementation of a tableau-based $\mathcal{ALC}$ description logic engine.

4 Inference Queues

The inference queue plays a fundamental role in the delivery system from the upstream to the downstream Web Service. As a queue it provides two main functions: **insert** and **remove**, which are asynchronous calls (i.e. they can be called by separate threads in any order.) The insert operation accepts facts and rules and places them into the queue. It is non-blocking in that at any time a call to insert will succeed without any infinite delays. The remove operation produces facts that are logical consequences of the already inserted facts and rules. It is also non-blocking in the following sense; if there is some fact to be removed, there will be an at most finite delay. Usually any delay is extremely short.

In Section 3 we discuss how the inference queue is incorporated into an engine for composing Web Services. The inference queue takes on the role of transporting the messages from one Web Service to the next. The messages are

NewFacts is a priority queue of facts, ordered by $\preceq$, that have not yet been processed. *OldFacts* is a list of facts that have already been processed. *Rules* is a list of rules, and the first condition of each is designated the *selected goal*. *backgroundProcessingIsComplete* is a boolean variable which recalls whether processing has been done since the most recent **remove** and is initially true. *mostRecentlyRemovedFact* is a fact which has a valid value when *backgroundProcessingIsComplete* is false.

```
synchronized insert(c)
   if not backgroundProcessingIsComplete
      performBackgroundProcessing(mostRecentlyRemovedFact)
   endif
   process(c)
   notify
end insert

synchronized Fact remove()
   if not backgroundProcessingIsComplete
      performBackgroundProcessing(mostRecentlyRemovedFact)
   endif
   repeat
      while NewFacts is empty
         wait
      end while
      select and remove a new fact f_new from NewFacts
   until f_new is not subsumed by any member of OldFacts
   set backgroundProcessingIsComplete to false
   set mostRecentlyRemovedFact to f_new
   return f_new
end remove

synchronized performBackgroundProcessing(f_new)
   for each rule r whose first condition unifies with f_new
      resolve r against f_new producing r_1
      process(r_1)
   end for each
   set backgroundProcessingIsComplete to true
   add f_new to OldFacts
end performBackgroundProcessing

process(c)
   if c is a rule
      add c to Rules
      for each old fact f_old unifying with the selected goal of c
         resolve c with f_old to produce the new result n
         process(n)
      end for each
   else
      add c to NewFacts
   end if
end process
```

Fig. 7. Inference Queue Operations: insert and remove

in XML, but while they are in the inference queue, they are considered to be terms in logic.

The inference queue has six properties that make it suitable for our task. (See [11] for proofs.) (1) The engine computes only **sound** conclusions, which means they are correct according to the meaning of the given formulas. (2) It computes a **complete** set of conclusions, which means that if a conclusion logically follows from the facts and rules that have been inserted into the queue, then such a conclusion will be produced by the inference queue. (3) The output is **irredundant** in that a fact, once produced, will not be produced again. Moreover, suppose one fact is more specific than another, as $p(a)$ is more specific than $p(X)$ for all values X. Then the inference queue will not generate the more specific fact after it has already produced the more general one. (4) The conclusions are generated by **fairly**. This means for every implied conclusion, either it or a more general one will eventually be chosen as the output to a remove request; it will not be infinitely often deferred. Combining this property with completeness means we are guaranteed that every conclusion will eventually be produced, given a sufficient number of remove requests. To ensure this, the inference queue depends on a partial order, $\preceq$ between the facts, and produces them in accordance with this order. (5) The calls to insert and to remove are **thread safe** which means that these calls can occur simultaneously without the risk of the mechanism getting into an unsafe state. (6) Calls to insert and remove are **responsive** which means that there will not be an infinite delay between calls. This remains true even if the facts and rules have an infinite number of consequences.

Figure 7 shows the details of how the inference queue operations are defined. Note that it uses the convention that methods declared to be *synchronized* are mutually exclusive; there can not be threads active in two such methods at once. The insert and remove methods also communicate via *wait* and *notify*, which allows remove to block if the queue is currently empty, but to return a value when a fact becomes available.

Other features of the inference queue can contribute to other parts of a robust Web Services architecture, including the ability to detect, report and recover from errors during service invocation. The inference queue is based on the jDREW open-source libraries [10], which are found on SourceForge.

5 Related Work

The task we perform is similar to that proposed for a broker [4], in that messages from one Web Service are translated to an appropriate form for another, where these Web Services are not in direct communication, but pass through a third party. The abstraction algorithm in [4] is similar to our proposal, but it does not consider converters.

A recent proposal [5] to align OWL-S to DOLCE, is motivated in part by the difficulty to state in OWL-S that the output of one process is the input of another. We are interested in this work and look forward to seeing more details on how the proposal would handle the converters that we have employed.

A number of recent proposals for integration of Semantic Web Services have been presented. We are not aware of any other work chosing to compile the integration tasks into rules, yet the integration tasks are similar. For example, the merge-split of Xu and Embley [14] corresponds to our XML tree reshaping, and their superset/subset value matching corresponds to our matching of basic elements, as described in Section 3. They seem not to have incorporated converter rules. Likewise the work of Giunchglia and Shvaiko [8] and that of Silva and Rocha [9] consider mapping Web Services descriptions. Each of these papers offers insights for using semantic descriptions to assist with the integration problems. We are less focused on this task, and more on the rule compilation and execution tasks, so the approaches can be considered complementary. Indeed, the problem of dynamically specialing converters seems not to have been addressed in related field of database integration.

The inference queue appears related to rule-based systems, with a long history from OPS-5 [6] to modern implementations [7]. They include rules for reacting to an event E under conditions C to perform action A with postcondition P; hence are sometimes called ECAP rules. These systems do not usually have a model theoretic semantics and usually allow one to delete facts. Our system computes one model incrementally and does not allow deletion of facts; instead they are preserved. The RETE match algorithm [6] on which most rule-based systems are implemented bears a resemblance to the resolution-based system we propose in the inference queue, in that it activates the goals in a rule in left-to-right order and does high-speed matches from facts, or working memory elements, to the conditions in the rules. Rule-based systems have a notion of negation as the non-existence of a fact; we propose to allow negation in the inference queue in the future.

6 Conclusion and Future Work

To conclude, we introduce a rule-based approach for integrating semantic Web Services at the grounding level. Suppose a pair of Web Services is selected by an abstract reasoning system, such as a planner, such that the output of one, intended as the upstream service, semantically matches the input of the other, intended to be downstream. Our approach is capable of generating data transformation rules; data from the upstream Web Service is transformed into the data model of the downstream service such that the meaning is preserved. The core transformation steps include reshaping the XML structure (complex data types), followed by mapping the leaf level elements (the basic components). Special convertors can also be employed as built-in utilities for semantic type conversions or type casting between the basic components. The inference queue is proposed to generate messages in their desired formats. Given a set of transformation rules, the inference queue can produce sound, complete, and irredundant results. It is also fair, responsive, and thread safe. We illustrate with two use cases. Our approach compensates for the inflexibility of third-party code and completes the integration solution after Semantic Web Services discovery and composition.

We plan to add a form of negation to the inference queue. Given a stratification of the literals that is consistent with the partial order $\preceq$ employed by the inference queue, we could then guarantee that any consequence removed from the queue is consistent with the model of data so far inserted into the queue. If negative information is later inserted or derived, those previously removed consequences may not be consistent with the model including the new information. We will consider whether such a system could be useful in the realm of Web Services.

Our rules for performing the data transformation are definite clauses for which all of the reasoning in the descrption-logic level has already been done. We also consider to compile to SWRL [3] and defer some description logic reasoning until invocation time, as the data passes through the inference queue. In future work we propose to give the inference queue DL reasoning ability.

References

1. Harold Boley. The rule markup initiative. http://www.ruleml.org, 2003.
2. The OWL Service Coalition. OWL-S 1.0 Release. "http://www.daml.org/services/owl-s/1.0/", 2004.
3. Ian Horrocks et al. SWRL: A Semantic Web Rule Language Combing OWL and RuleML. http://www.daml.org/2003/11/swrl/, 2003.
4. Massimo Paolucci et al. A Broker for OWL-S Web Services. In *Semantic Web Services: Papers from the 2004 AAAI Spring Symposium*, pages 92–99. AAAI Press, 2004.
5. Peter Mika et al. Foundations for OWL-S: Aligning OWL-S to DOLCE. In *Semantic Web Services: Papers from the 2004 AAAI Spring Symposium*, pages 52–59. AAAI Press, 2004.
6. C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19:17–37, 1982.
7. Ernest Friedman-Hill. Jess, the expert system shell for the java platform. Technical report, http://herzberg.ca.sandia.gov/jess/, 2002.
8. Fausto Giunchiglia and Pavel Shvaiko. Semantic matching. In AnHai Doan, Alon Halevy, Natasha Noy, editor, *Proceedings of the Semantic Integration Workshop*, volume Vol-82. CEUR-WS.org, 2003.
9. Nuno Silva and Joao Rocha. Service-oriented ontology mapping system. In AnHai Doan, Alon Halevy, Natasha Noy, editor, *Proceedings of the Semantic Integration Workshop*, volume Vol-82. CEUR-WS.org, 2003.
10. Bruce Spencer. A Java Deductive Reasoning Engine for the Web. www.jdrew.org, 2004. Accessed 2004 Jan 12.
11. Bruce Spencer and Sandy Liu. Inference Quenes for Communicating and Monitoring Declarative Information between Web Services. In *Proceedings of RuleML-2003*, number 2876 in Lecture Notes in Artificial Intelligence, 2003.
12. W3C. Web Services Description Language. "http://www.w3.org/tr/wsdl", 2001.
13. W3C. Web ontology language reference. "http://www.w3.org/TR/owl-ref/", 2004.
14. Li Xu and David W. Embley. Using domain ontology to discover direct and indirect matches for schema for schema elements. In AnHai Doan, Alon Halevy, Natasha Noy, editor, *Proceedings of the Semantic Integration Workshop*, volume Vol-82. CEUR-WS.org, 2004.