Inference Queues for Communicating and Monitoring Declarative Information between Web Services

Bruce Spencer and Sandy Liu

Institute for Information Technology – e-Business
National Research Council of Canada
46 Dineen Drive, Fredericton, New Brunswick, Canada E3B 9W4
Faculty of Computer Science, University of New Brunswick
P.O Box 4400, Fredericton, New Brunswick, Canada E3B 5A6
{Bruce.Spencer, Sandy.Liu}@nrc.gc.ca
http://iit-iti.nrc-cnrc.gc.ca/groups/il_e.trx

Abstract. We introduce the inference queue as a mechanism for communicating and transforming data in Web services choreography. The insert operation provides definite clauses to the inference queue, and the remove operation generates output that is sound, complete, fair, and irredundant. Both operations are thread safe and responsive. The inference queue can form part of a highly configurable data transformation system. Rules can also monitor for events of interest based on the occurrence of certain conditions. Our suggestion of system wide monitoring of communication is complementary to existing Web services proposals.

1 Introduction

Communication from one web service to another is the most basic task of two important recent research efforts in e-business: Web Service Choreography[16] and Semantic Web Services[15]. This information should be meaningful to both parties; it is common in B2B e-business today that both parties will have a common format, perhaps a variant of XML, and will have a common understanding of the intended meaning of each portion of the message, of each XML tag. But for the future we want to consider doing business with automatically discovered Web services, not previously known. Thus the information should be declarative in nature, with meanings assigned from a standard ontology in the general subject area, as suggested by the Semantic Web research effort[14].

RuleML[2] provides the basic syntactic structures from first order logic so that the relations and functions/complex structures are apparent to both parties. RuleML is also used for expressing data transformation rules, that convert syntactic conventions used by one party to syntax understandable by the other party. Such transformations will preserve meaning because the inferences are sound. Beyond transformations, rules are suited to express preconditions that declarative information must meet before its communication can be accepted

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 121–135, 2003.
© Springer-Verlag Berlin Heidelberg 2003

by the intended recipient. Rule engines that mediate the communication serve as monitors of communication within the overall system. They can report any exceptional conditions to high level control systems, or intelligently transform the message so that it becomes acceptable. Thus a rule engine can serve as both a data transformation system and a monitor.

In this paper we introduce the inference queue as a mechanism for communicating and transforming data. We assume that we are reasoning about XML messages sent from one web service to another. It is convenient to use RuleML specifically for these messages as a single RuleML fact can contain the entire message. The inference queue has several characteristics required in the situation we have described so far: it has asynchronous insert, remove and isEmpty methods; it buffers information, allowing the producing process on the insert side to occasionally exceed the capacity of the consuming process on the delete side, so that the producer wastes less time waiting; it is an inference engine that is responsive and fair in the way it generates the complete set of sound conclusions. The inference system also eliminates reporting any results that are duplicates or specializations of previous results.

The inference queue can form part of a highly configurable monitoring system; rules can conclude that some exceptionally interesting event is taking place based on the occurrence of certain conditions. As the evidence of that event passes through the queue, the rule fires. The inference queue possibly has several output ports so these can be configured to transmit reactions to the interesting event. The inference queue also has possibly several input ports, so that data from mixed sources can be combined. A rule may be added with multiple conditions, and facts for these conditions are expected to arrive at distinct input ports. The conclusion of this rule would be produced for the output port(s) when all of its conditions are inserted at the input ports. Thus events arising in separate locations can be monitored, arbitrary conditions on their combination can be expressed and reactions taken when those conditions arise. This allows us to trace information relevant to one business process through a network of Web services, where the links in the network are inference queues.

The rest of the paper is organized as follows: We cover the necessary background on Web services and Web service choreography frameworks. We then introduce the inference queue for transporting data from one web service to another, we define the six properties they need for this task, and establish that they are met by our architecture, which is described in some detail. To make the discussion more concrete, we give an example of a Web service connected to a client with two one-way inference queues, and illustrate an intelligent monitoring system.

2 Web Services and Choreography Frameworks

Web services allow businesses to describe, publish, and invoke self-contained modular software components over the Internet and therefore make distributed computing available Internet-wide. However, a single Web service is often not

sufficient to accomplish a business process that involves multiple parties and/or multiple activities. There is a need to compose a series of Web services together to achieve a new or more useful service. For instance, in order to plan a trip, a user may require the cooperation from a set of services including a flight-scheduling service, a hotel booking service, a car-rental service, and a credit card payment service. This set of services can be considered as a business process. The Business Process Management (BPM) community has long been looking for solutions to standardize the description, modeling, deploying, and monitoring of business processes that contain services provided by heterogeneous vendors. There are several specifications that can serve as candidates to streamline this process including BPML[1], BPEL4WS[7], ebXML BPSS[9], WSCI[6], and DAML-S[4].

3 Inference Queues

An inference queue is a priority queue data structure into which we enqueue (insert) input fomulas and from which we dequeue (remove) inferred output formulas. Syntactic restrictions on both formulas are often imposed to simplify the reasoning task. In this paper we restrict the input formulas to definite clauses and the output to single literal positive clauses (facts) implied by those definite clauses.

Inference queues have standard queue operations (insert, remove and isEmpty) but adapt the standard operational semantics. In response to insert requests, facts and rules are inserted at one end, the *upstream end*; and in response to a remove request, these facts and any implied facts are removed from the other *downstream* end. Unlike normal queues, the size of the output of an inference queue is not necessarily equal to the size of the input.

We define the inference queue also to have four important properties relating to logic and deduction: soundness, completeness, irredundancy and fairness, and to have two properties relating to its simultaneous usage by separate threads: thread safe (assessible by separately running thread or processes) and responsive (no infinite computations are allowed between requests). In this section we describe these six properties of the inference queue in detail, tell why these properties are relevant to communication among Web services, and explain how these properties are guaranteed by our implementation.

3.1 Definitions of the Properties We Seek

Suppose definite clauses $C_1, \ldots, C_n$ have been inserted into the queue, where $\{C_1, \ldots C_n\} \models F_1, \ldots, F_m$, for facts $F_1, \ldots, F_m$. If the queue is capable of delivering, in response to m remove requests, all of these facts (or a covering set of more general facts), then it is **complete**. If it is capable of delivering only entailed facts, it is **sound**.

Suppose that the facts are removed from the queue in the order $F_1, \ldots, F_m$. For every $i \geq 0$ and every $j > 0$ fact F_{i+j} is delivered after fact F_i, and it is not the case for any i that $F_i \models F_{i+j}$. In other words, no fact is more specific than a previous fact. Then the output queue is **irredundant**.

By **fair** we mean that no given conclusion will be infinitely deferred from being produced for a dequeue. Every conclusion will eventually be found. We use fairness to express priority among facts. Exceptional conditions can be given priority, so they are guaranteed to be reported first.

We expect the inference queue to be used by several concurrent programs, or threads, as opposed to a single program thread. A program that can interact with other concurrent programs is **thread safe.** This means two threads cannot interfere with each other's tasks, and usually is done by defining a critical section in the code where only one thread at a time is permitted to run. It also means we have the option to force threads requesting data to wait. In our case a remove request is defined with blocking semantics. This means the thread invoking the remove request will be blocked if there is no element in the queue. The expectation is that another thread will eventually call the insert method, making an element available, and then the removing thread can be notified (re-activated) and the remove request granted. The other requests, to insert a clause into the queue and to ask if the queue is empty, are non-blocking operations.

The queue is also defined to be **responsive** which means that between any pair of requests, no infinite computations are allowed. From a finite set of input clauses, an infinite set of output clauses can arise:

$$\{p(X) \rightarrow p(f(X)), \quad p(a)\} \models p(a), p(f(a)), p(f(f(a))), \ldots.$$

Inference queues address this infinite list by requiring that an infinite sequence of calls to dequeue be made to produce the infinitely many answers. Since the operations are responsive, there will be no infinite operations between these successive calls.

The combined properties of being complete, responsive and fair mean that not only are all conclusions generated (completeness) with no infinite delays (responsiveness), but also that this remains true as new information is being dynamically added or inferred. Thus if a specific fact has been inferred but has not yet been delivered to a remove request, and a sequence of facts is being inferred, such as the infinite computation in the previous paragraph, then eventually that undelivered fact will be delivered, before the infinitely many other facts are all delivered. To accomplish this we depend on the existence of a partial order $\preceq$ (preceeds) between facts. Suppose the inference queue delivers the facts $F_1, \ldots, F_m$ in this order. Then this order must be consistent with the given partial order. In other words $F_{i+j} \not\preceq F_i$.

Beyond the fixed set of clauses, if new facts are being inserted or inferred between remove operations, the partial order is still observed. For example, suppose the fact F_i has been recently removed and F_{i+1} is next in line to go, and a new fact G is inserted or inferred before F_{i+1} is removed, such that $G \preceq F_{i+1}$. Then G will be removed before F_{i+1}. It may also be the case that $G \preceq F_i$, but clearly G cannot be removed before F_i because G was not available at the time F_i was removed. While this appears to disrupt the precedence ordering of the output facts, it is not contrary to the definition. Among the facts available when the remove request is made, a smallest fact is always removed.

Given $Clause_1 C_1, \ldots C_m \rightarrow A$ and $Clause_2 A_1, \ldots, A_n \rightarrow B$ such that there exists i and a substitution θ that unifies A with A_i,
Produce $(A_1, \ldots, A_{i-1}, C_1, \ldots, C_m, A_{i+1}, \ldots, A_n \rightarrow B)\theta$

Note that Robinson's resolution[12] would also allow multiple A_j to be resolved at once, but this is not necessary for completeness of resolution with definite clauses. Here we restrict the application so that $m0$, to enforce unit clause resolutions, and so that $i1$ to select goals in textual order.

Fig. 1. Robinson's resolution applied to definite clauses

We must also assume that the partial order has no infinite strata. In other words, for any given element there is a finite number of elements that are greater than it. This property of no infinite strata is easy to guarantee. For instance if the $\preceq$ relation is based on the number of symbols in the atom, i.e. its length, and there is a finite set of symbols to draw from, then there will be only a finite number of atoms shorter than any given atom.

3.2 A Theorem Prover with the Logic Properties We Seek

The above six properties, as far as the authors know, have not been combined into one theorem prover before, but we have found it is entirely feasible to do so. The ideas are these: (1) allow the theorem prover to incrementally accept new clauses, (2) turn the basic theorem proving loop from an autonomous producer of formula into a demand-driven, or lazy, producer of one fact at a time, and (3) after either operation (a new clause is inserted or a new fact is selected), run part of the theorem prover's machinery to infer all the conclusions that are needed to restore the system to a state ready to service the next request.

The two essential features of any theorem prover are its rule(s) of inference and its strategy for applying those rules, examples of which are shown in Figures 1 and 2, respectively. For the inference queue, we use Robinson's resolution applied to definite clauses. Since this is the only rule of inference and since it is well-known to be sound, **soundness** of the inference queue follows immediately.

Negative literals are selected in clauses in the order in which they appear, which is sufficient since it is well known that resolution's literal selection within a clause is don't-care nondeterministic. In other words, if a proof exists with a specific condition selected first, then the same proof exists with any of the conditions selected first. Thus the general rule of Figure 1 is specialized to select the first condition.

The application strategy is related to those found in theorem provers such as Otter [10], in that the most important strategies and design decisions are represented: The *set of support* restriction [8] is reflected in the fact that the positive clauses are resolved against mixed ones, so the facts form the set of support. Forward subsumption is applied when the a new fact is selected, and it is applied lazily on selection rather then eagerly as facts are generated. This decision requires storing potentially many more new facts, but far fewer subsumption

Three data structures are defined: *NewFacts, OldFacts* and *Rules*.

NewFacts: This is a priority queue for storing single literal definite clauses (facts), ordered by $\preceq$, for that have not yet been processed. Initially it is populated with the facts from the input clases.

OldFacts: This is a list for storing single literal definite clauses that have already been processed.

Rules: This is a list for storing definite clauses with at least one negative literal (condition). These are indexed by one of these negative literals, called the *selected goal*. Assume that the first negative literal is the index. Initially it is populated with the rules from the input clauses.

main loop
  select and remove a new fact f_{new} from *NewFacts*
  while f_{new} is subsumed by some member of *OldFacts*
    select and remove another f_{new} from *NewFacts*
  end while
  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
  add f_{new} to *OldFacts*
end main loop

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

Fig. 2. An application strategy for definite clause reasoning

tests are attempted. Backward subsumption is not applied for similar reasons; the list of old facts would potentially be less redundant if it were applied, in that old facts could not be more specific than new ones, but a far greater number of subsumptions tests would be attempted. Furthermore, forward subsumption guarantees that the list of old facts is absolutely irredundant when the facts are ground. We expect that the inference queue will often be in this situation. The stored facts are separated into two lists: *OldFacts* and *NewFacts* so that we can ensure unification attempts are never tried more than once for any given pair of literal occurrences. The reason is this: For every potentially unifiable pair consisting of a fact and a condition that occurs as the selected condition (the first

condition) in a rule, either the fact is selected from *NewFacts* before the rule is created, or the rule is produced before the fact is selected. There are two cases to consider: Suppose the fact is selected before the rule is created. Since the rule does not yet exist, the fact will be put into **OldFacts** without the resolution being performed. Later when the rule is produced, **process** is called and the resolution against the old fact will be performed. Alternately, suppose the rule is created before the fact is selected. Later, when the fact is selected, it is resolved in the main loop against all existing rules including the one of interest. Then it is put into the old facts and is never resolved against any existing rules, only against new rules. Thus, there is exactly one point in time when any given fact is attempted to be resolved against any given rule condition. No redundant searching is done. This argument also serves to convince the reader of the completeness of the search precedure: all proofs will be built because all possible resolutions are tried. Combined with the notion of completeness of the resolution rule of inference for generating single literal conclusions, we can now conclude that the system is **complete** for generating facts. Note that resolution is not complete in the sense of generating all logical consequences, because it does not generate weaker conclusions like $A \vee B$, which are logical consequences of single literal conclusions like A.

There is one point of caution about the theorem prover in Figure 2. The number of stored clauses, both rules and facts, will grow. Since a ground rule with n conditions in its body may give rise to n stored clauses (with $n-1$, $n-2$, ..., to 0 conditions) there is a quadratic effect on the size of stored clauses. For non-ground clauses the affect, naturally, may be worse.

The set **NewFacts** is a priority queue, ordered on $\preceq$ defined on atomic formulas. Of all of the new facts that can be selected a minimal one is chosen next. When control in the procedure is at the select statement of the main loop, the set **NewFacts** contains all of the facts that have been inferred but not yet reported. It does not necesarily contain all of the facts that are implied by the set of clauses. So it is important to point out that the sequence of selected facts is not necessarily arranged in $\preceq$ order. What is true and important for our purposes, is that no single fact will be left unselected indefinitely long, and that the $\preceq$ order allows us to express heuristically the order in which we would prefer to receive our conclusions – in effect, our order of importance. Thus the system displays **fairness.**

3.3 An Inference Queue with All the Properties We Seek

While the theorem prover in Figure 2 has some of the properties we need, it cannot be used in a multithreaded environment in this form. We need to convert it so that explicit insert and remove operations are defined that are thread safe and responsive. By thread safe, we mean that its internal state will not be disturbed by any set of simultaneous external requests from client processes. By responsive we mean that the system cannot go into an infinite loop in response to requests, even though the clause set may imply an infinite set of atoms.

The main inference queue operations are shown in Figure 3. This program has many features in common with Figure 2. The main difference is the subsystem for background processing. Background processing performs the inferences needed after a new fact is selected, and corresponds to the second half of the code in the **main loop** of Figure 2. Before any other insert and remove operations are done, the status variables are checked to see if this background processing is yet undone. The background processing is separated for two reasons. First the inference queue should react to remove requests as quickly as possible, so it returns a result as soon as one is found, deferring the background processing. Second, since requests are made asynchronously from client threads, there will sometimes be opportunities when there are no requests pending, so the inference queue can do this processing on its own time. Thus there is expected to be a background process, or *daemon*, that monitors the requests and the state of *backgroundProcessingIsComplete*. It does the background processing while the queue is otherwise idle.

To establish that the system is thread safe, one needs to ensure that the mutual exclusion lock is in force anytime the remove operation is running. This lock prevents any other thread from entering code that can affect the internal data structures. Likewise the thread performing the insert operation claims the mutual exclusion lock, preventing another thread from entering either the insert or the remove operation. We use the convention that any procedure that claims ownership of the mutual exclusion lock is declared *synchronized.*

We need the system to be **responsive,** which means there is no chance that any step will take infinitely long. While **process** may appear to contain a potentially infinite loop because of the inner call to itself, each call to it is guaranteed to terminate. Note that for a given parameter in a call to **process**, the argument provided to each inner call is smaller by one literal. Since there will be a finite number of inner calls, limited by the finite number of stored rules, there is no chance for an infinite loop. Similarly background processing will never take infinitely long. Neither insert nor remove has any infinite loops, either.

Although we have not shown it, the operation to tell if the inference queue is empty first ensures that all background processing is done, and then returns true if and only if the *NewFacts* priority queue is empty.

3.4 Variations of the Inference Queue

We allow several input ports to be opened to an inference queue. There is no state information associated with the insert process, so there is no essential difference between having one or many input ports.

It may be of interest for an inference queue client to receive consequences that match a certain pattern only, instead of receiving all consequences. This is easily handled by invoking a unification step as part of the output port's operation; facts that do not meet the pattern are not transferred out of that port.

There may also be several output ports receiving consequences from an inference queue, and the queue services remove requests from each. New output ports can be opened at any time. Suppose that a recently initialized inference

In addition to the three data structures from Figure 2, *NewFacts*, *OldFacts* and *Rules*, there are two non-local variables:

- A boolean state variable *backgroundProcessingIsComplete* recalls whether processing has been done since the most recent **remove**. It is initially true.
- A fact *mostRecentlyRemovedFact* 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
   while NewFacts is empty
      wait
   end while
   select and remove a new fact f_new from NewFacts
   while f_new is subsumed by some member of OldFacts
      select and remove another f_new from NewFacts
   end while
   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
      for each old fact f_old unifying with the selected goal of c
         resolve c with f_old to produce new result n
         process(n)
         add c to Rules
      end for each
   else
      add c to NewFacts
   end if
end process
```

Fig. 3. Inference Queue Operations: insert and remove

queue has delivered several facts to its single output port. Then a second output port is opened. The first several requests from the second port will be serviced by replaying the *OldFacts* list, until it catches up with the first port. If the newer port then makes more requests, the inference machinery will be used to calculate the consequences. Then requests from the original output port will be serviced by replaying these recently inferred facts, which were lately added to *OldFacts*.

This mechanism allows at any time an inference queue to service a request for a full disclosing of what has transpired, and in what order. Thus the *OldFacts* lists doubles as a log file.

We may need a monitoring service whose main interest in an inference queue is not what can be concluded, but what has been sent to other output ports. This need can be met by opening an *observation port*, which is an output port that is not allowed to advance the state of the inference engine. Thus once the remove requests from an observation port are given all of the old facts, further remove requests are blocked and these threads are made to wait. When a dequeue request is made from another (non-observation) port it is serviced with a new fact, and the thread waiting on the observation port is notified and delivered the same fact.

4 Use Cases

The inference queue complements existing Web service choreography proposals, such as WSCI, DAML-S and BPEL4WS. It offers services not currently implemented in those proposals, but mentioned as future work[5](p. 3). We view Web services choreography as the configuration of communicating networks where nodes are instances of Web services; we see the role of the inference queues as the links between these nodes. By using an inference queue for communication among Web services, data transformation and format conversion are possible. In the following sections we offer a small example to illustrate potential advantages of using inference queues for this communication. We further the example to illustrate that an inference queue can provide a monitoring service for tracing the steps of a specific business process in the Web services network. Since the inference queue stores in *OldFacts* all of the messages that have been transmitted, the data is available later for forensic analysis.

4.1 Mediating Communication between Web Services

Suppose a travel agent Web service mediates between a traveller and various agents for flights, events, and hotel reservations for example. This is illustrated in Figure 4.

In this use case, the human client (1) requests a Travel Agent Web service (2) to help him/her purchase a ticket that meets his needs. The Travel Agent service uses Canadian conventions for encoding dates, exclusively. It deals with various Ticket Agent Web services, some of which use different local conventions for encoding data, such as dates. Suppose there is little flexibility in the interfaces of

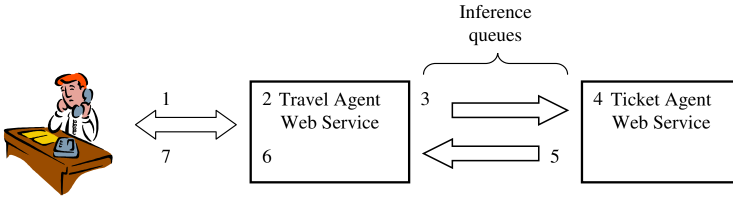

Fig. 4. Web services mediation in two directions

these Ticket Agent services – and they do not support Canadian date encoding, day-month-year. Then the inference queue (3) connecting the Travel Agent to the Ticket Agent can have some rules for date transformation: (Here we use the Prolog convention for displaying rules.)

requestOut(Flight, LocalDate, CustomerNumber) ←
 requestIn(Flight, CanadianDate, CustomerNumber),
 dateConversion(CanadianDate, LocalDate).

When the inference queue receives a request, such as the fact

requestIn('Delta861', '23-06-2003', 'Traveler229'),

following rule is added to *Rules*:

requestOut('Delta861', LocalDate, 'Traveler229') ←
 dateConversion('23-06-2003', LocalDate).

This is eventually used to conclude the fact: requestOut('Delta861', '06-23-2003', 'Traveler229'). Note that the date is now month-day-year. This fact is delivered at the next remove request from the downstream service (4), or to an agent that has the initiative to pass such facts to the input of the Ticket Agent service. Web services may need such an active process since they are usually reactive and not initiators.

When the Ticket agent is ready to issue a purchase order number, which will tell the customer what he needs to know to actually purchase the ticket, it issues the response fact to the return inference queue (5): responseOut('Delta861', 'Traveller229', '06-23-2003', 'PO1242'). That inference queue has the task of insulating the Travel Agent from non-Canadian dates, so it contains the rules.

responseOut(Flight, CanadianDate, CustomerNumber, PurchaseOrderNumber)
 ←
 responseIn(Flight, LocalDate, CustomerNumber, PurchaseOrderNumber),
 dateConversion(CanadianDate, LocalDate).

The fact responseOut('Delta861', 'Traveller229', '23-06-2003', 'PO1242') is eventually issued to the Travel Agent (6) and some appropriate response is given to the Customer (7).

4.2 Monitoring and Exception Handling

Of all of the Web service choreography proposals, BPEL4WS deals most with exceptions; an exception can be thrown by a Web service and caught by a BPEL fault handler. The error handling framework can compensate by attempting to roll back the latest step in the business process. Complementary to this, the inference queue allows faults to be defined and monitored outside of the Web service, so it is possible to have robust processing using Web services that do not incorporate fault handling. In Figure 5 we have extended the example from Figure 4 so that several Web services are being invoked. Each Web service is connected to another Web service by an inference queue, perhaps with a return queue. Each inference queue links to a monitor via an observation output port, represented in the figure by thin lines and described in Section 3.4. In principle there could be a network of Web services linked by inference queues that send information back to a common monitor, which would have access to all communicated facts across the system.

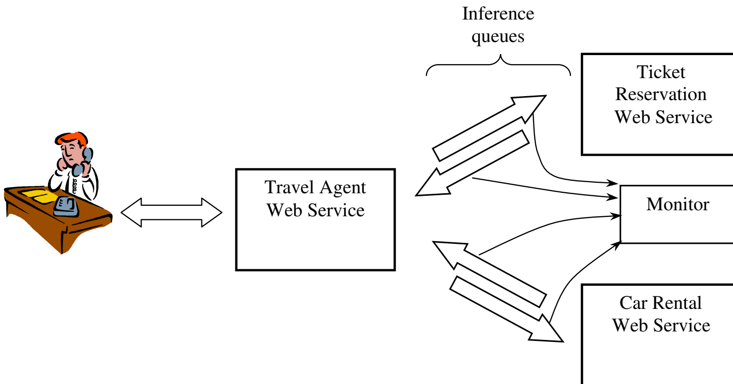

Fig. 5. Web services fault detection and forensic analysis

5 Related Work

The existing Web Service Description Language (WSDL) only defines operations in terms of incoming/outgoing messages and binding details at the syntactic level without outlining the relationship of the messages in a service that encompasses multiple operations. The Web Service Choreography Interface (WSCI) describes how WSDL operations are choreographed and which properties these choreographies expose, such as transaction and correlation. As a result, WSCI complements the shortfalls of WSDL by providing a detailed description of the behaviours in different states for a given Web service described by WSDL. WSCI also describes the collective message exchange among interacting Web services, providing a global, message-oriented view of a process involving multiple Web services.

Note that WSCI only describes a one-sided interface for a Web services. In other words, the message exchange is described from the point of view of one Web service[11]. To compose a set of Web services, we need standards to model the workflow of services. A few emerging standards taking different approaches are attempting to fill in this gap.

The Web Services Modeling Framework (WSMF) propose a comprehensive conceptual model for developing, describing, and composing Web services by appointing the principles of maximal de-coupling and a scalable mediation service. The WSMF defines four main elements: ontologies, goal repositories that define the problems to be solved by Web services, Web services descriptions, and mediators to overcome the interoperability problem.

While WSMF provides a very high-level model for integrating Web services, others are aiming to offer grounded models to realize the vision described by WSMF for Web services orchestration and choreography. Early endeavors in this path include IBM's Web Services Flow Language (WSFL) evolved from Petri Nets and Microsoft's XLANG based on the Pi-Calculus model[3]. These two proposals have since converged into a single specification called BPEL4WS (Business Process Execution Language for Web Services), which unifies the essence of graph-oriented processes from WSFL and structural constructs from XLANG to enable the aggregation of Web services into a process execution model. "As an executable process implementation language, the role of BPEL4WS is to define a new Web service by composing a set of existing services.[17]." Hence, the composition (called the *process* in BPEL4WS) indicates how the service interface fits into the overall execution of the composition.

As BPEL4WS uses WSDL portType information for service description, it inherits the limitation of WSDL, which does not describe side effects, pre- or post-conditions of services, and the expressiveness of service behavior and inputs/outputs are constrained by XML and XML Schema. In contrast, DAML-S, a semantic markup specification for Web services, employs a complementary approach to describe Web services. Indeed, DAML-S is a DAML-OIL ontology for describing Web services. It aims to enable automated Web services discovery, invocation, composition, and execution monitoring by providing sufficient semantic description. A DAML-S document comprises a ServiceProfile, a ServiceModel, and a Service Grounding. The ServiceProfile describes the properties of a service such as input and output types, pre-conditions and post-conditions, and binding patterns to facilitate automatic service discovery where the ServiceModel together with ServiceGrounding provide information for an agent to make use of a service.

One of the important aspects of modeling a business process is to have a mechanism for exception handling, such that when an exception is raised during the course of a business process, the model allow appropriate recovery actions (such as roll-back) to be performed. Web services provide a basis for passing messages between participants in collaboration-based processes. Nevertheless, most of the current proposals do not provide this much-needed monitoring service. DAML-S has the notion of execution monitoring, but has not yet been defined

in the current release; BPEL4WS claims to have exceptions (faults) built into the language via the <throw> and <catch> constructs, the fault concept on BPEL4WS is directly related to the fault concept on WSDL and builds on it. As a result, the fault has to be defined explicitly when describing the messages in a Web service. In WSDL, the optional fault elements specify the abstract message format for any error messages that may be output as the result of the operation. To bridge the gap, the inference queue can be placed between two Web services. Process rules can be stored in the queue, such that the outputs of one service can be input to the queue as facts before they are fed into the next Web service. The inference mechanism can then determine if a Web service has performed the task and has generated results in expected fashion, based on the facts and a set of pre-defined rules.

6 Current Status, Future Work, and Conclusions

The current inference queue system is implement using the jDREW tool kit [13] in Java. We have deployed it as a Web service as well. It accepts input and generates output either in a Prolog-like syntax, or RuleML. We plan to offer it as open source.

In the future we plan to study how declarative descriptions of Web services and the links between them, as in BPEL4WS, may be transformed into running networks of Web services with inference queues proving the communication between them. Each inference queue could be loaded with rules that check the preconditions, in the sense of DAML-S, of the downstream service and monitor the effects of the upstream service.

This paper introduces the inference queue as a means of communicating meaningful information among Web services. It is based on a thread-safe priority queue data structure. It contains first order definite clauses and it inserts inferred facts into the queue, so that the output is sound, complete, fair, and irredundant. The insert and remove operations are responsive; even if there is an infinite set of inferred facts, no operation will need to wait forever. We propose to use the inference queue for transporting information between pairs of Web services; data from one service to the next can be mediated and transformed by the rules in the queue. Most Web services choreography specifications do not support comprehensive monitoring of systems built up from individual Web services, although a need for this is identified in WSMF and DAML-S. We suggest how the inference queues can use their rules to detect exceptional conditions, and pass these exceptions to a monitor receiving information from all inference queues in the system. Proposals for handling exceptions in Web services, such as in BPEL4WS, depend on the Web service detecting problems and throwing exceptions. But if a system architect seeks to incorporate a third-party Web service, s/he cannot depend on that Web service to throw faults appropriate for the rest of the system.

References

1. A. Arkin. Business Process Modeling Language. “http://www.bpmi.org/bpml-spec.esp”, 2002.
2. Harold Boley. The rule markup initiative. http://www.ruleml.org, 2003.
3. BPMI.org. BPML||BPEL4WS: A Convergence Path toward a Standard BPM Stack. “www.bpmi.org/downloads/BPML-BPEL4WS.pdf”, 2002.
4. DAML-S Coalition. DAML Services. “http://www.daml.org/services/”, 2003.
5. The DAML Services Coalition. DAML-S: Semantic Markup for Web Services. In *Proceedings of SWWS'01 The First Semantic Web Working Symposium*, pages 404–411, 2003.
6. Assaf Arkin et al. Web Services Choreography Interface (WSCI)1.0. “http://www.w3.org/TR/wsci”, 2002.
7. T. Andrews et al. Business Process Execution Language for Web Services version 1.1. “ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf”, 2003.
8. L. Wos, D. Carson and G. Robinson. Efficiency, completeness and the set of support strategy in theorem proving. *J. ACM*, 12:536–541, 1965.
9. Paul Levine. ebXML business Process Specification Schema Version 1.01. “http://www.ebxml.org/specs/ebBPSS.pdf”, 2001.
10. W. W. McCune. Otter 3.0 users guide. Technical Report ANL-94/6, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL, 1994.
11. Sun Microsystems. WSCI 1.0 Specification - FAQs. “http://wwws.sun.com/software/xml/developers/wsci/faq.html”, 2003.
12. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.
13. Bruce Spencer. The design of j-drew: a deductive reasoning engine for the web. In Kung-Kiu Lau Manuel Carro, Claudio Vaucheret, editor, *First Colognet Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems*, pages 155–166. Universidad Politécnica de Madrid, September 2002. CLIP4/02.0.
14. W3C. Semantic Web. “http://www.w3.org/2001/sw/”, 2001.
15. W3C. Semantic Web Web Services. “http://www.w3.org/2001/11/11-semweb-webservices”, 2001.
16. W3C. Web Services Choreography Working Group. “http://www.w3.org/2002/ws/chor/”, 2003.
17. S. Weerawarana and F. Curbera. Business Process with BPEL4WS: Understanding BPEL4WS, Part 1. “http://www-106.ibm.com/developerworks/library/ws-bpelcol1”, 2002.