Toward specifying contracts and protocols for Web services

Guy Tremblay
Dépt. d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, QC, H3C 3P8, Canada
Email: tremblay.guy@uqam.ca

Junghyun Chae
Dépt. d'informatique, UQAM

***Abstract*— Web services are emerging as a key infrastructure for providing inter-operation between applications and systems and for providing support for the deployment of e-commerce business processes. One important issue for ensuring the growth of Web services is having ways of describing the available Web services in a precise way.**

One possible step toward providing semantic information for Web services is through the use of formal contract specification—that is, using pre/post-conditions. We present a number of ways in which pre/post-conditions could be introduced into Web services descriptions, for specification as well as dynamic verification purpose.

The use of pre/post-conditions, however, is not sufficient to describe the semantic of a group of *related* operations, for example, to describe the legal sequences in which these operations can/should be used. Although business process description languages like BPEL4WS or WSCI can express such descriptions, these are *procedural* descriptions, not necessarily appropriate for *specification* purpose. We present one possible way in which such descriptions could be provided for Web services, using *path expressions* that can show the order in which the various operations of a Web service can and should be invoked. Static and dynamic uses of these protocol specifications are then described.

I. Introduction

Web services are emerging as a key infrastructure for providing inter-operation between applications and systems and for providing support for the deployment of e-commerce business processes. One important issue for ensuring the growth of Web services is having ways of describing the available Web services in a precise manner.

Precise specifications of services can be used for different purposes. First and foremost, when the service or system is yet to be implemented, they serve as a precise description of the problem to be solved, providing the implementers, and testers, with a specific description of the expected behavior. For those wishing to use some systems or services, appropriate specifications also help document, thus understand, the expected behavior. When searching for Web services, specifications can thus be used to narrow the search space. Finally, specifications, when sufficiently precise and formal, can also be used for verification purposes.

Although various languages and notations are currently available for describing Web services, there exists no consensus yet, even though some standards are emerging [1], [2]. The lack of consensus is particularly clear when it comes to describing the behavior of business processes and their composition, whether it be in terms of orchestration (description of a specific business process) or in terms of choreography (description of the interactions among a group of business processes) [3].

More importantly, some of the available notations (e.g., WSDL [1]) provide little semantic information pertaining to the service's behavior. Other notations, although they can provide precise descriptions of behavior [2], [4], do it in a form which is operational and algorithmic, that is, by describing the expected behavior through programming language-like structures. Such descriptions, however, may not necessarily be considered as being "specifications": "a specification is a statement of *properties* required of a product, or a set of products" [5]. The distinction between a model of the behavior of the system and a specification of its expected properties is one which is made explicitly in model-checking approaches [6], where the formalisms for describing the model's behavior (e.g., automata, transition systems, process algebras) are generally quite different from those use to express the properties (e.g., temporal logic). The same characteristic is true for abstract model or contract-based approaches to specifications [7]–[9], where the properties of the abstract model are expressed in some form of first order logic.

In the following sections, we first present a small number of languages, all defined as XML applications, that have been proposed for describing Web services and business processes [10], [11]. Although they are just a few among various other similar languages, they are representative of the major concepts and most appropriate for the level of specifications we intend to discuss.

Then, we discuss how semantic information for Web services could be provided through the use of formal contract specifications—that is, pre/post-conditions. We present different ways in which such contracts specifications could be introduced and how these contracts could also be used for dynamic verification. However, as we also show, pre/post-conditions are still not sufficient to describe the semantic of a group of *related* operations, for example, to describe the legal sequences in which these operations can/should be used. We then discuss how WSDL service descriptions could be

augmented with *path expressions* specifications, as a means to describe the order in which the various operations can and should be invoked.

II. Existing Languages for Describing Web Services

There exists a large number of notations and languages for describing Web services and business processes [10], [11]. In the following, we present a small sample, to better understand their expressive power and the various levels they describe.

A. WSDL

The Web Services Description Language is an XML-based language for describing Web services in a way that is independent of their implementation technology. The description included in this section is based on the working draft of the standard (version 2.0, dated March 2004) [1].

WSDL separates the abstract functionality of a web service from its concrete implementation. The abstract functionality is described in terms of *interfaces*—previously, called *port types* in WSDL 1.1—, which are collections of related operations. Roughly speaking, an operation is described by a name, a signature, and a message exchange pattern (see below); the signature is a sequence of messages, which are directed data flows whose contents are described in some type system—typically using XML Schemas.

The abstract description makes no mention of the specific location of the service (network address), message format (SOAP or other), or transport protocol (HTTP, TCP/IP). Instead, such information is included in the concrete description or *binding*. A binding may specify message formats and message transmission protocols for an entire interface, or for a specific operation, or for a specific message or fault of a particular operation. An interface may be offered at different access points (called endpoints). Each endpoint may use a different message format (e.g., SOAP) or message transmission protocol (e.g., TCP/IP or HTTP) and is associated with a specific network. Finally, a service groups together endpoints that implement a common interface.

We illustrate the main concepts of the abstract functionality part through a simple example, presented in Figure 1. In this description, we have a single interface, which has a single operation with one input message and one output message, each composed of a single part. The types of the messages are defined under the `<types>` tag as XML schema descriptions. In this example, the types are in-lined, but they can be included from another file. The pattern attribute of the `<operation>` element points to a description of the message exchange pattern that characterizes the invocation of the operation. A message exchange pattern specifies the sequence and cardinality of messages exchanged by the operation. The pattern is defined in terms of named placeholders, and the mapping between the pattern and the messages of the operation is assured via the `messageLabel` attribute of message references—in this case, `in-out`. The WSDL draft specification provides eight common message exchange patterns (in-only, robust in-only, in-out, in-optional-out, out-only, robust out-only, out-in, and out-optional-in) [1].

```
<?xml version="1.0"?>
<definitions name="StockQuote">
  ...
  <types>
    <schema
        targetNamespace=
         "http://example.com/stockquote.xsd"
        xmlns=
         "http://www.w3.org/2000/10/XMLSchema">
      <complexType name="TradePriceRequest">
       ...
      </complexType>
      <complexType name="TradePrice">
      ...
      </complexType>
   </schema>
   </types>

   <message name="GetLastTradePriceInput">
     <part name="body"
           type="xsd1:TradePriceRequest"/>
   </message>

   <message name="GetLastTradePriceOutput">
     <part name="body"
           type="xsd1:TradePrice"/>
   </message>

  <interface name="StockQuotePortType">
       <operation name="GetLastTradePrice"
          pattern=
            "http://www.w3.org/2004/03/wsdl/in-out">
         <input messageLabel="In"
           element="tns:GetLastTradePriceInput"/>
         <output messageLabel="Out"
           element="tns:GetLastTradePriceOutput"/>
      </operation>
  </interface>
  ...
  <binding>
  ..
  </binding>
  <service>
  ...
  </service>
</definitions>
```

Fig. 1. Excerpts from a WSDL specification

Operations can throw exceptions, called faults in WSDL. A fault is characterized by 1) its name, 2) its data contents, and 3) its direction (which distinguishes between exceptions/faults raised by the operation itself and those received by it). Faults are defined per interface (omitting direction information) and then referenced in specific operations using either an `<in-fault>` element or an `<outfault>` element, enabling sharing faults between different operations. The message exchange patterns may specify how faults are handled (e.g., returned in lieu of the output message, or as a separate message [1]).

The recent version of WSDL supports the specification of features and properties, notions absent in WSDL 1.1. A feature describes "an abstract piece of functionality typically associated with the exchange of messages between communicating parties" [1]. Features include things such security, reliability, transaction. The presence of a feature means that a service

supports it and requires partners to do the same (e.g., secure communications).
Features may have accompanying documentation, like all WSDL components (interfaces, operations, messages, etc.), and can have different scopes, including interfaces-level, operations-level, or message-level.
Similarly, properties are used to describe the service's operational parameters, and may be used to control the behavior of a feature. A property can specify values or constraints on values. An example property would be the encryption algorithm, and the constraint would state that it must be a public key encryption algorithm. This property controls the behavior of the security feature. Values and constraints may be defined by referring to types defined using XML Schemas. Properties also have similar scopes and composition model to that of features.

B. BPEL4WS

The Business Process Executable Language For Web Services (BPEL4WS) [4] is a language for modeling business processes, executable or not, written for the Web services context. The basic idea is that a process may be thought of as a collaboration between services or tasks described in the Web services format, more specifically, using WSDL. Whereas WSDL defines a syntax for expressing the interface of services (in the IDL sense, i.e., signature of operations), but says little about the interaction model (or rather assumes a simple one-way or round-trip message passing protocol), BPEL4WS allows to describe the entire interaction sequence. BPEL4WS has two target uses, which are clearly stated and explained:

- executable processes: these describe actual business processes that are internal to an organization and are completely specified (i.e., executable);
- abstract processes, also known as business protocols: these are the parts of a business process of an enterprise that are exposed to outside processes in the context of an inter-enterprise interaction.

This distinction is very helpful and not made in the case of other languages, such as BPML [12]. This, plus the fact that BPEL4WS makes provision for roles and partners, makes it more appropriate for describing inter-organizational processes. Roughly speaking, a BPEL4WS process description consists of a declaration part that introduces various elements needed to describe the process, followed by the actual description of the process, i.e., its behavior. The declaration part includes the following elements:

- A description of the messages exchanged between services. The message structure is similar to that used in WSDL: a message consists of parts, each with a name and a type. The type component is described using XSD types—the use of XSD is not exclusive and BPEL4WS can accommodate other type systems.

```
<message name="POMessage">
  <part name="customerInfo"
        type="sns:customerInfo"/>
  <part name="purchaseOrder"
        type="sns:purchaseOrder"/>
</message>
<message name="InvMessage">
  <part name="IVC" type="sns:Invoice"/>
</message>
```

- A description of the services invoked. The description follows the WSDL standard (as of version 1.1): each service, defined as a `portType` (now called `interface` in WSDL 2.0), consists of a bunch of operations. An operation has a name and a set of parameters. The parameters are simply messages playing the role of inputs and outputs.

```
<portType name="purchaseOrderPT">
  <operation name="sendPurchaseOrder">
    <input message="pos:POMessage"/>
    <output message="pos:InvMessage"/>
    <fault name="cannotCompleteOrder"
           message="pos:orderFaultType"/>
  </operation>
</portType>
```

- A description of the "contracts" between participating process. Each contract—called partner link type—defines roles and associates them with port types (interfaces). For example, in a supply chain example, we have customers, businesses, and their suppliers. The interaction between a customer and the business to fulfill an order, and between the business and its suppliers to restock, are managed by two separate contracts/partner link types, each of which identifies the roles played by each service interface (port type). One of the role describes what is provided by the process being described, whereas the other describes what is required from the other partner; when no specific requirements is placed on the expected partner, a single role can be specified. For example, the following, taken from [4], describes a contract where an `invoiceService` provides access to a `computePricePT` port type but in turn requires the other partner to provide an `invoiceCallbackPT` port type on which the answer can be sent back:

```
<plnk:partnerLinkType name="invoiceLT">
  <plnk:role name="invoiceService">
    <plnk:portType name="pos:computePricePT"/>
  </plnk:role>
  <plnk:role name="invoiceRequester">
    <portType name="pos:invoiceCallbackPT"/>
  </plnk:role>
</plnk:partnerLinkType>
```

- A description of partners. While partner link types define the various contracts, it does not specify which entity will play which side in the contract. A BPEL4WS process thus contains an identification of the various partners involved in the various contracts, the roles they play (partner link type), and the role (`myRole`) the enterprise doing the modeling plays. Those partners will be referred to later in the description of the steps of the process: each step is performed by a partner (or self). Here, we show a description of two partners, where the first is the customer and the second is the `invoiceProvider` playing the role `invoiceService` in the contract (partner link type) `invoiceLT`, where I, myself, play the role of `invoiceRequester`.

```
<partnerLinks>
  <partnerLink
     name="customer"
     partnerLinkType="lns:purchaseLT"
     myRole="purchaseService"/>
  <partnerLink
     name="invoiceProvider"
     partnerLinkType="lns:invoiceLT"
     myRole="invoiceRequester"
     partnerRole="invoiceService"/>
  ...
</partnerLinks>
```

Other elements in the declaration part include local variables defined within the scope of the process and exchanged as inputs/outputs between the process steps, and a description of fault handlers, which specify the desired response in case of a fault.

The process (i.e., its dynamic behavior) is defined using a *flow*, which is a partially ordered set of activities that correspond to invocation of operations, defined in the various services, that will be performed by the partners identified above. Process flow supports sequential activities (using the `<sequence>` tag), concurrent activities (`<flow>` tag), and arbitrary control dependencies between process steps (using `<link>`s) which ensure that a particular process step can only be executed after another step has completed. As with BPML processes [12], BPEL4WS processes include descriptions of compensation handlers and support the notion of scope (called context in BPML) and correlation sets, which are data values that uniquely identify process instances. Figure 2 shows excerpts from a process behavior description. The (purchase order handling) process starts by receiving a purchase order (`PO`) from a customer. It then makes a copy of the customer info from `PO` into the customer info of a shipping request. Then it invokes an operation of the `shippingProvider` partner to ship the order to the customer. This example illustrates, in part, the use of control dependencies: the `requestShipping` operation is the *source* of a control dependency (a link, called `ship-to-invoice`) linking it to the operation (not shown here) `sendShippingPrice` of the `invoiceProvider` partner.

BPEL4WS, unlike WSDL, makes some notion of contract explicit, in the sense of clearly stating a bidirectional relationships between two parties, that is, a `partnerLinkType` is an association between two services. It is implicit that the operations within those services collaborate through message exchange. An example of such an exchange is given in the description of the process, but this may not be the only valid exchange.

C. WSCI and WS-CDL

WS-CDL (Web Services Choreography Description Language) is the latest initiative from the Web Services Choreography Group of the World Wide Web Consortium (W3C) [2]. This language has an interesting history. A draft proposal for WSCI (Web Service Choreography Interface) was issued in August 2002 [13]. That effort seems to have been abandoned, and a new standardization effort launched under the name WS-CDL, which, for all practical purposes, seems to have started from scratch. Indeed, it started with a new requirements document, from which the new language, WS-CDL, was built from the ground up, with no reference to the WSCI effort. Because we feel that there were some useful ideas in WSCI, we first describe the latter. We then briefly comment on the emerging WS-CDL standard.

```
<sequence>
  <receive partnerLink="customer"
           portType="lns:purchaseOrderPT"
           operation="sendPurchaseOrder"
           variable="PO">
  </receive>
  <flow>
    <links>
     <link name="ship-to-invoice"/>
     <link name="ship-to-scheduling"/>
    </links>

    <sequence>
     <assign>
     <copy>
      <from variable="PO" part="customerInfo"/>
      <to variable="shippingRequest"
          part="customerInfo"/>
     </copy>
     </assign>
     <invoke partnerLink="shippingProvider"
             portType="lns:shippingPT"
             operation="requestShipping"
             inputVariable="shippingRequest"
             outputVariable="shippingInfo">
       <source linkName="ship-to-invoice"/>
     </invoke>
     ...
    </sequence>
    ...
  </flow>
  ...
</sequence>
...
```

Fig. 2. Excerpts from a BPLE4WS behavior (flow) description (taken from [4])

The premise of WSCI is that WSDL descriptions are insufficient to describe the true nature of services. Indeed, WSDL defines services in terms of collections of seemingly unrelated operations (port types, i.e., interfaces), but has no notion of state, and does not state the valid sequences of operation invocations. Using the classical example of a travel agent service, a traveler can confirm an itineray—e.g., invoke some operation `confirmItinerary()`—only if that itinerary has already been created—e.g., by invoking an operation `createItinerary()`—and if it was created explicitly for that same customer. Yet, nothing in a WSDL description makes those constraints explicit. To some extent, WSCI brings statefulness into web service descriptions both explicitly by associating properties (variables) with interface definitions, and implicitly by stating the valid operation invocation sequences. Further, WSCI considers that the same web service can be used by several different processes, and thus, a given

service (WSDL interface) may be associated with several valid sequences of uses, where each one would be characterized by a specific process description. These are the basic premises of WSCI.

Figure 3 shows excerpts of a simplified version of a travel agent service. Notice this example uses WSDL 1.1's syntax, and hence, web services are represented by port types instead of interfaces. The `<interface>` tag used here is WSCI-specific, and will be discussed further below.

In this example, the travel agent web service includes two port types (interfaces in WSDL 2.0), `TAtoTraveler`, whose definition is shown above and which interfaces with travelers, and `TAtoAirline`, which interfaces with airlines and whose definition is not shown but is referenced in the process `BookSeats` (line 15). The valid message sequences that the travel agent web service supports are enclosed in the `<interface>` under WSDL's top-level `<definitions>`. Each valid interaction sequence is represented by a `<process>`, and each process is a combination of `<action>`s, where each action corresponds to the invocation of an operation (lines 8-10). The process called `PlanAndBookTrip` (lines 5-14) is a simple `<sequence>` of actions. Other compositions include `<all>` (parallel execution), `<foreach>`, and `<switch>`. Line 3 shows the declaration of a `<correlation>` element, which is referenced in line 12. The idea here is that a travel agent service would typically be handling several "conversations" (process instances) with different customers or with the same customer but for different itineraries. The `<correlation>` element specifies how to uniquely identify a given process instance (or a given thread of conversation). This is identical to BPEL4WS's notion of correlation set. Also, processes define scopes—called contexts—within which variables may be defined—called properties. These properties are typically used to hold data values (i.e., state information) exchanged and manipulated by the different operations of a process.

Also, not shown in this example, processes can raise exceptions. Those exceptions would be specified, along with the steps to handle them. Processes can also specify transaction boundaries around a set of actions to indicate that the whole set should be treated as an atomic action, as well as compensation activities to specify what to do when transactions fail (using a `<compensate>` activity).

In addition to representing the dynamic behavior of individual services, WSCI includes the notion of a global model, which binds together several interfaces, one per web service. In the travel example, the travel agent service interfaces with both travelers and airlines. Similarly, airlines interface with both travel agents—for booking and pricing, say—and with travelers, for ticket delivery. In a global travel scenario, operations from the travel agent service will be talking to operations from the airline web service. The global model simply connects operations from the various interfaces. Figure 4 shows part of the global model for the travel scenario. A `<connect>` element associates the operation that initiates an exchange with the one that completes it.

```
<?xml version = "1.0" ?>
<wsdl:definitions
   name = "GlobalModel"
   targetNamespace=
     "http://example.com/consumer/models"
<!--various imports -->
  <wsdl:import
     namespace=
       "http://example.com/consumer/traveler"
     location=
       "http://example.com/traveler.wsci"/>
  ...
<model name = "AirlineTicketing">
  <interface ref="air:Airline"/>
  <interface ref="tra:Traveler"/>
  <interface ref="ta:TravelAgent"/>

 <!-- Traveler / TravelAgent -->
 <connect operations=
    "tra:TravelerToTA/PlaceItinerary
     ta:TAtoTraveler/ReceiveTrip" />
   ...
 <!-- Travel Agent / Airline -->
 <connectoperations=
    "ta:TAtoAirline/CheckAvailability
     air:AirlineToTA/VerifySeatAvailability"/>
   ...
 <!-- Traveler / Airline -->
   ...
</model>
</wsdl:definitions>
```

Fig. 4. An example WSCI global model for a travel scenario (taken from [13])

WSCI's descendant, the Web Services Choreography Description Language (WS-CDL), is still very much in its infancy [2]. The description of the choreography itself is not much different from that of process in WSCI or BPEL4WS. At the interface level, WS-CDL uses the participant, role, and relationship triad whereby participants play roles in relationships (contracts). In this regard, WS-CDL looks more like BPEL4WS than WSCI.

III. Specifying Contracts of Operations

As shown earlier, WSDL-based specifications of web services and their associated operations are expressed strictly in *syntactic* terms, that is, operations are described simply by giving their signature—the name and types of their arguments and results. Clearly, such specifications provide little information about the exact behavior of the operations. In this section, we examine a number of approaches to provide improved specification for web services operations.

A. Contracts as Pre/Post-Conditions

Using pre/post-conditions for describing and specifying the behavior of operations is a long established practice. Viewing such specifications as *contracts* between the user of a service and the provider of this service has been popularized by Meyer's design by contract approach [9]. A contract necessarily entails some benefits as well as some obligations to both parties, as illustrated in Table I.

Specifying contracts as explicit pre/postconditions requires using an appropriate "formal" specification language. Many such

```
  <? xml version = "1.0" ?>
<wsdl:definitions name = "Travel Agent Dynamic Interface"...>        1.

   <!-- WSDL complex types -->                                       2.
   ...
   <!-- WSDL message definitions -->
   ...
   <portType name = "TAtoTraveler">
     <documentation>
        This port type references the operations the Travel Agent
        performs with the Traveler service
     </documentation>
     <operation name = "ReceiveTrip">
        <input message = "tns:tripOrderRequest"/>
        <output message = "tns:tripOrderAcknowledgement"/>
     </operation>
     <operation name = "BookTickets">
        <input message = "tns:bookingRequest"/>
        <output message = "tns:bookingConfirmation"/>
     </operation>
     <operation name = "SendStatement">
        <output message = "tns:statement"/>
     </operation>
   </portType>

   <-- WSCI's specific additions -->

   <correlation name = "itineraryCorrelation"                         3.
                property = "tns:itineraryID">
   </correlation>

   <interface name = "TravelAgent">                                   4.
     <process name = "PlanAndBookTrip"                                5.
              instantiation = "message">                              6.
       <sequence>                                                     7.
          <action name = "ReceiveTripOrder"                           8.
                  role = "tns:TravelAgent"                            9.
                  operation = "tns:TAtoTraveler/ReceiveTrip">        10.
           </action>
           <action name = "ReceiveConfirmation"                      11.
                   role = "tns:TravelAgent"
                   operation = "tns:TAtoTraveler/bookTickets">
             <correlate correlation="tns:itinCorrelation"/>          12.
             <call process = "tns:BookSeats" />                      13.
          </action>
          <action name = "SendStatement"                             14.
                  role = "tns:TravelAgent"
                  operation = "tns:TAtoTraveler/SendStatement"/>
          </action>
       </sequence>
     </process>
     <process name = "BookSeats" instantiation = "other">            15.
        <action name = "bookSeats"
                role = "tns:TravelAgent"
                operation = "tns:TAtoAirline/bookSeats">
        </action>
     </process>
   </interface>
</wsdl:definitions>
```

Fig. 3. Excerpts from a WSCI example (taken from [13])

	Obligations	Benefit
Client	Must satisfy the pre-condition	Is assured that the post-condition will hold
Provider	Must satisfy the post-condition	Is assured the pre-condition will hold

TABLE I
OBLIGATIONS AND BENEFITS OF CONTRACTS FOR CLIENT AND PROVIDER OF A SERVICE

languages exist, e.g., Z [8], VDM [7], OCL [14], etc. In recent years, however, various ways of using pre/post-conditions in *programming languages* have been proposed, thus lowering the level at which such contracts can be expressed: direct support of pre/post-conditions by the programming language itself (e.g., Eiffel [15]), special comments that can handled by a pre-processor (e.g., the `iContract` tool for Java [16]), simple `assert` instruction (e.g., in C and Java 1.4).

An interesting benefit of including and supporting pre/post-conditions directly in programs is that the associated conditions can be checked *dynamically*. In other words, such contracts can be used for documentation purpose as well as for verification purpose, ensuring at run-time that the software do behave correctly with respect to the specified pre/post-conditions—additional assertions can also be stated, to describe the key states along which the program make progress. Using assertions and contracts is becoming a key practice for professional software development [17], [18] and thus should also be available for the development of Web services.

In what follows, we briefly describe three proposals to the specifications of pre/post-conditions-style contracts in WSDL: DAML-S, an existing proposal, and two alternative proposals, Larch/WSDL and WSDL-Contract.

B. DAML-S

DAML (DARPA Agent Markup Language) is an AI-inspired language, building on RDF and RDF-S, for defining ontologies to be used in the development of the so-called Semantic web. Building on DAML, a group of researchers has developed a markup language to more specifically describe the semantic of Web services, "that provides an agent-independent declarative API capturing the data and meta-data associated with a service together with specifications of its properties and capabilities, the interface for its execution, and *the prerequisites and consequences* of its use" [19], [20].

The proposed DAML-S ontology contains the following three classes of concepts:

- Service profile: Concepts in this category describe "what the service does", that is, they describes functionalities in terms of input and output types, effects (events caused by the execution of the service), etc. Logical properties are used to describe such conditions (although how exactly these properties are expressed is not clearly specified).
- Service model: A concept in this category "tells "how the service works"; that is, it describes what happens when the service is carried out" [20]. More specifically, a particular subclass of service model has been defined, namely, the process model, which in fact consists of two aspects: the process model itself—which describes the process behavior in terms of its composite and atomic actions, *à la* BPEL4WS—and the process control model—which allows agents to monitor and control the execution of a process (this part has not yet been defined).
- Service grounding: A concept in this category describes how the service can be accessed by an agent, for example, using a specific communication protocol.

Because of DAML-S' roots in ontology languages, the syntax presented in the various papers describing DAML-S is based on RDF. A number of elements of DAML-S appear not yet to be fully specified (at least from the examples presented in those papers), for instance, how exactly the preconditions and effects (postconditions) would be described, what kind of logical language would be used, etc.

Probably because it predates the work on BPEL4WS, the DAML-S papers do refer to WSDL, but not to BPEL4WS. However, they do claim to inherit, for their process model description, from previous work on existing process and workflow specification languages.

C. Larch/WSDL

The Larch approach to formal specification [21] is well known for having introduced a *two-tier approach* that leads to the definition of a family of specification languages. Larch's first tier is the Larch Shared Language (LSL), which is essentially a library of (pure) abstract data types described using an algebraic approach—i.e., in terms of signatures (syntax) and recursive equations (semantics) relating the various operations—and which is independent of any specific programming language. This tier thus describes an ideal mathematical world consisting strictly of (immutable) values and functions on such values, where notions such as side-effects or exceptions do not exist. Clearly, this is not the world in which software artifacts live. Software systems and components are described, instead, using a second-tier, which builds on the first-tier for the key mathematical types and concepts. This second tier, because it aims at describing the interface of concrete software components, is *programming language specific*. Thus, numerous variants of Larch interface languages exist, each tailored to the specific interactions mechanisms provided by the target language—for example, LCL (Larch C language) [21] does not provide exceptions whereas Larch/Ada and Larch/C++ [22] do.

Although a Larch interface language is specific to a target language, the specifications clearly cannot be expressed totally in the target language syntax. Thus, a suitable syntax must be developed for each interface language. Specifications written in this language can then be analyzed and some of their properties can be checked using the Larch Prover. Although the specifications can be included as comments in the target language based descriptions, there is no specific support for the dynamic evaluation of the pre/postconditions, as done

for example in the Eiffel language [15]. Thus, the interface specification, although expressed with concepts specific to the target language, does not get fully integrated into the associated implementations.

One possible direction for integrating contracts into web services description would be by defining a Larch/WSDL interface language. However, as mentioned earlier, the integration of the interface specification into the target language (WSDL specification in our case) is not necessarily natural or seamless. Furthermore, LSL is a rich and complex language, whose use requires a lot of expertise. Enriching this library is not easy either, because of the reliance on the algebraic specification method (recursive equations).

D. WSDL-Contract

The Eiffel language, developed by Meyer [15], popularized the use of pre/postconditions for both the specification of operations' contracts and for their dynamic (run-time) verification. More precisely, in Eiffel, mechanisms for the specification of pre/postconditions and other assertions (including loop variants/invariants) have been integrated directly into the language definition. Such specifications can be interpreted logically, that is, as formal specification (and documentation) of the operations. In addition, these assertions can also be interpreted operationally, that is, as conditions that must be checked at run-time to ensure the program is in a correct state relative to its intended specification.

Although Eiffel is, as of our knowledge, the only language to directly integrate such formal specification and verification of contracts into the language, other strategies have been proposed to obtain similar results—formal documentation and specification together with run-time verification of assertions—in other languages. The most common approach, now available for a wide variety of languages (for example, Java [16], C [23], Python [24], etc.), is the use of special comments together with a pre-processor (à la JavaDoc). The fact that the pre/postconditions are specified using comments means that the source program can be handled normally by the regular compiler. On the other hand, if required by the user, the pre-processor can also be used to analyze those special comments and generate, in the executable code, appropriate run-time checks.

One such pre-processor for Java is `iContract` [16], where the special comments are based on the JavaDoc style and the syntax of the logical expressions is based on OCL [14], a part of UML [25]. A simple example of an operation to add an element `o` into a collection `c` is presented in Figure 5.

Using a similar approach for integrating contracts into WSDL descriptions would have a number of advantages. First of all, such an approach is relatively *lightweight* compared to a Larch-based approach, as it does not rest on understanding a complex library (LSL) of algebraic types; instead, knowledge of the key collection operations, *à la* OCL, is generally sufficient. Also, as mentioned earlier, such an approach to the integration of contracts is becoming fairly common, as it is now available in many programming languages. Finally, such an approach would also allow for the integration of dynamic verification of assertions associated with Web services operations, an interesting way to verify the behavior of complex business processes.

```
/**
 Add an element o to collection c.

 @pre !c.contains(o)
 @post c.size() = c@pre.size()+1
 @post c.contains(o)
*/
public void addElement( Collection c, Object o )
{ ... }
```

Fig. 5. An example of an `iContract` specification

We are currently in the process of defining an appropriate XML-based syntax for specifying such contracts for WSDL operations, based on OCL concepts and operations. WSDL 2.0's feature/property mechanism will be used to specify those contract specifications. We are also planning to develop a dynamic contract evaluation engine, in order for the various contract conditions to be verified dynamically. Appropriate assertion violation faults will be defined and will be generated when pre/post-conditions fail to hold.

IV. Specifying Web Services Protocols

Using Web services to integrate various business applications and processes requires being able to describe long sequence of interactions between partners, that is, requires defining appropriate *business protocols* — "formal description of the message exchange protocols used by business processes in their interactions" [4, p. 8].

In the present section, we first explain why describing such protocols is important, and why it cannot be done solely with pre/post-conditions. We then introduce a number of ways in which such protocols can be described, illustrating why the *operational*-type of specifications used in existing languages may not be the best approach. Finally, we discuss how protocols specifications can be used for verification purpose, for example, for ensuring that the protocol is indeed obeyed.

A. The Limits of Pre/Post-Conditions and the Need for Protocol Descriptions

WSDL describes web services in terms of collections of seemingly unrelated operations that users can invoke in any particular order. This may be true for services offering basic search capabilities into an information database, but is rarely the case with services that represent access points to elaborate business processes. More is needed.

BPEL4WS enables us to describe business processes that involve the interaction of several business partners. When the process under description is internal to an organization, the partners could be departments or divisions within the enterprise. When the process is external, the partners would be different enterprises. In either case, the functionality supported by a partner is expressed in terms of a WSDL web service. While the WSDL descriptions of the individual partners are

stateless, the structure of the overall process shows a choreography of their services for the purposes of executing a global process. Using abstract BPEL4WS processes, it is also possible to explicitly state which operation of a given service S should precede which other operation of S, thus showing the overall *orchestration* of operations from the various partners in the context of a specific process.

WSCI also attempts to address WSDL's statelessness by providing specification of valid sequences of operations within each service, regardless of any specific global process in which a given service might be involved. This is unique to WSCI: defining the range of stateful behaviors that a given service can support independently of a usage context. WSCI's global model does provide an example of such a context.

It thus appears that various distinct concepts are needed to capture the semantics of Web services:

1) The Web service API, represented in a WSDL format;
2) The *contracts* (pre/postconditions) of the individual operations;
3) The *protocol* showing the valid sequences of operation calls.

As discussed in the previous section, one of the traditional way of representing operation semantics is by using pre/post-conditions. Such pre/post-conditions do not obviate the need for finding a way to express what are the valid sequences of calls to the operations. If we think of operations as simply steps in an overall process, then we must have an idea about what this processes is. For instance, in the context of a traveling agent service, we could model the semantics of the operation that checks itineraries with the post-condition that the returned itineraries (output message) correspond to the requested dates and destination (input message). However, this does not tell us that *each* inquiry must necessarily conclude either with a cancellation or with an actual booking.

Specifying a business protocol thus aims at describing "the observable behavior of a service and the rules for interacting with the service from the outside [...] [such] that external actors will know at each stage in the given process which messages that service may or must send or receive next" [13, p. 14].

B. Operational or Declarative Descriptions?

Both BPEL4WS (with its abstract business protocols) and WSCI allow the description of the legal sequence of operations associated with a business service. In both cases, it is explicitly stated that these descriptions cannot be executed. For instance, in the case of WSCI, it is said that an interface description "is declarative and cannot, by itself, be executed." [13, p. 14].

However, such specifications are far from declarative. For instance, declarative programming languages can be described as follows [26], :

> Whereas imperative programming gives the computer a list of instructions to execute in a particular order, declarative programming describes to the computer a set of conditions and lets the computer figure out how to satisfy them.

The constructs used to define business protocols in both BPEL4WS and WSCI are clearly based on procedural and imperative control structures. Although such control structures are fine to describe concrete, thus executable, business processes, they may not be the best notation for describing (abstract) business protocols. Let us illustrate this with a simple example.

Suppose we want to define a simple Web service (`intVarMachine`) for managing a non negative integer variable whose operations and associated protocol are the following—note that these operations can be seen as an abstraction of a common pattern of use, where some entity is created, repetitively updated, and then released or finalized:

1) First, the state of the variable must be initialized (with a specific integer value) by calling the `init` operation.
2) Then, a sequence of calls can be performed to `inc` (increase) or `dec` (decrease) the variable's value; each such call requires specifying an integer argument and returns no result. If a call to `dec` would produce a negative value for the variable, an `invalid_dec` fault is signaled.
3) Finally, the content of the variable can be obtained by using the `demandVal` operation, whose result is returned through an asynchronous call (operation `receiveVal` from `incDecPTCallback` port type). Furthermore, once the value of the variable has been obtained, no further call to `inc`, `dec` or `demandVal` can be performed until a new call to `init` is performed.

The WSDL `interface` (`portType`) for this service is presented (in part) in Figure 6. Using BPEL4WS abstract protocols, the required protocol can then be described as presented in Figure 7.

Although the abstract process presented in Figure 7 does describe the correct and allowable sequences of calls, this description is expressed strictly in procedural and operational terms. For example, the possibility of an exception being signaled is modeled through a non-deterministic (opaque) assignment to an `invalidValue` variable, whereas an explicit while-loop has to be used with an associated `terminated` variable to indicate the end of the sequence of calls to `inc`/`dec`.

This procedural specification of business protocols stems from the origins of BPEL4WS or WSCI, which are essentially XML-based representation of constructs found in languages used to describe the behavior of *reactive* systems, for example, the family of process algebraic languages [27]–[29]. The literature on the specification and verification of such systems [30]–[32], including the literature on model-checking [6], has long shown the usefulness of trying to separate the description of the model of the behavior of the system, what BPEL4WS and WSCI are all about, from the specification of the properties expected from that behavior. Furthermore, as the work on bi-simulation has shown [29], behavior descriptions which are sufficiently *abstract* can also be used as specification. It can hardly be argued that descriptions such as the one presented above are at a high level of abstraction.

```
<process name="intVarMachine" ...abstractProcess="yes">
  <partnerLinks>
    <partnerLink name="incDec"
                 partnerLinkType="tns:incDecLT"
                 myRole="incDecService"
                 partnerRole="incDecServiceRequester"/>
  </partnerLinks>

  <variables>
    <variable name="x" messageType="tns:x_nat"/>
    <variable name="dummy" messageType="tns:dummy"/>
    <variable name="r" messageType="tns:x_nat"/>
    <variable name="terminated" messageType="tns:t_boolean"/>
    <variable name="invalidValue" messageType="tns:t_boolean"/>
    <variable name="decErr" messageType="tns:invalid_dec"/>
  </variables>

  <sequence name="main">
    <receive name="receiveInput" partnerLink="incDec" portType="tns:incDecPT"
             operation="init" variable="x" createInstance="yes"/>

    <assign>
      <copy> <from expression="false()"/> <to variable="terminated"/> </copy>
    </assign>

    <while condition="bpws:getVariableData('terminated') = 'false'">
      <pick>
        <onMessage partnerLink="incDec" portType="tns:incDecPT" operation="inc" variable="x">
          <empty/>
        </onMessage>

        <onMessage partnerLink="incDec" portType="tns:incDecPT" operation="dec" variable="x">
          <sequence>
            <assign>
              <copy> <from opaque="yes"/> <to variable="invalidValue"/> </copy>
            </assign>
            <switch>
              <case condition="bpws:getVariableData('invalidValue') = 'true'">
                <empty/>
              </case>
              <otherwise>
                <throw faultName="invalid_dec" faultVariable="decErr"/>
              </otherwise>
            </switch>
          </sequence>
        </onMessage>

        <onMessage partnerLink="incDec" portType="tns:incDecPT" operation="demandVal" variable="dummy">
          <assign>
            <copy> <from expression="true()"/> <to variable="terminated"/> </copy>
          </assign>
        </onMessage>
      </pick>
    </while>

    <invoke name="callbackClient" partnerLink="incDec" portType="tns:incDecPTCallback"
            operation="receiveVal" inputVariable="r"/>
  </sequence>
</process>
```

Fig. 7. Abstract protocol described in BPEL4WS for `intVarMachine`

```
<definitions name="intVarMachine" ...>
  <message name="x_nat">
    <part name="idpart"
          element="s1:x_natElement"/>
  </message>

  <message name="t_boolean">
    <part name="idpart"
          element="s1:t_booleanElement"/>
  </message>

  <message name="invalid_dec">
    <part name="idpart"
          element="s1:invalid_decElement"/>
  </message>

  <message name="dummy"/>

  <portType name="incDecPT">
    <operation name="init">
      <input message="tns:x_nat"/>
    </operation>

    <operation name="inc">
      <input message="tns:x_nat"/>
    </operation>

    <operation name="dec">
      <input message="tns:x_nat"/>
      <fault message="tns:invalid_dec"/>
    </operation>

    <operation name="demandVal">
      <input message="tns:dummy"/>
    </operation>

  </portType>

  <portType name="incDecPTCallback">
    <operation name="receiveVal">
      <input message="tns:x_nat"/>
    </operation>
  </portType>


  <plnk:partnerLinkType name="incDecLT">
    <plnk:role name="incDecService">
      <plnk:portType name="tns:incDecPT"/>
    </plnk:role>
    <plnk:role name="incDecServiceRequester">
       <plnk:portType name="tns:incDecPTCallback"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Fig. 6. Excerpts from a WSDL description for a simple Web service for managing an integer variable

In order to write concise and precise descriptions of protocols, simpler, and more abstract, notations should be used. In the following, we examine one possible such notation (and variants of it).

C. Path Expressions

Path expressions were introduced to express synchronization of concurrent processes [33], [34]. A path expression can be used to describe, and restrict, the allowable sequences of operations on an entity, thus can be interpreted as a specification of the allowable "transactions" associated with a system [35].

Path expressions are a kind of *regular expressions* where basic path expressions have the following form [34]—parenthesis can be used, as necessary, to create sub-expressions:

- `A;B`: `A` can be executed followed by `B`.
- `A+B`: either `A` or `B` can be executed.
- `A`*: 0, 1 or more copies of `A` can be executed consecutively.

The protocol for invocation of the various operations of our integer variable Web service presented earlier in Figure 6 could thus be represented by the following path expression :

```
init; (inc + dec)*; demandVal
```

D. ω-Regular Languages and Temporal Logic Specifications

Regular expressions in various forms can also be found in other languages for specifying properties of processes' behavior. For instance, it is possible to extend modal and temporal logic operators with such regular expressions. This is done, for example, in the *regular alternation-free mu-calculus* [36], where various useful properties of processes can be expressed quite intuitively and concisely using those regular expressions, instead of using the more complex fixed point operators.

Of course, a reactive systems generally exhibits a non terminating behavior. In this case, regular expressions, which correspond to finite automata and finite sequences of operations, can be generalized to ω-regular expressions, which correspond to ω-automata that can deal with infinite sequences of computation. Figure 8 presents a (simplified) Buchi automata for our protocol, one possible form of ω-automata–here, the single state `s1`, appearing in a double circle, is an *accepting* state (instead of a final one).[1]

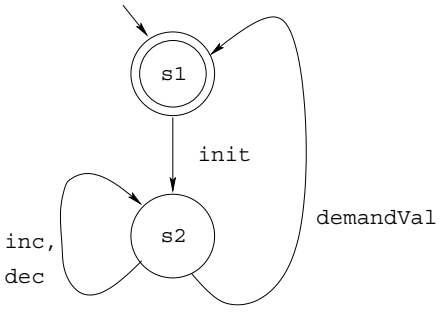

Fig. 8. A Buchi automata for specifying the protocol of the `intVarMachine`

Appropriate mappings between such automata and equivalent (linear) temporal logic exist, for example, between LTL and generalized Buchi automata [6]. Thus, specifying protocols using appropriate (ω-)regular expressions or linear temporal logic can be considered equivalent.

[1]Note that this is a simplified ω-automata, since some transitions have been omitted, namely, the non legal ones.

E. Generalizing Regular Expressions for Web Services

To better describe the behavior of Web services as allowed in BPEL4WS and WSCI, it is useful to describe not only the incoming messages but also the outgoing ones—the responses as well as the requests. Concurrent execution of operations can also be expressed in BPEL4WS or WSCI. Furthermore, specifying the possible faults, distinct from the regular messages/operations, can be useful. One possible way of better reflecting these aspects of a Web service behavior is to extend the language of path expressions with the following constructs:

- `A | B`: execution of `A` and `B` can proceed in parallel.
- `[A]`: execution of `A` is optional.
- `^A`: invocation, by the service, of an operation `A` that must be provided by the client, e.g., a callback operation.
- `!A`: signaling a fault `A`.

The specification of the protocol for our earlier example could thus be refined as follows:

```
init;
( inc + (dec; [!invalid_dec]) )*;
demandVal;
^receiveVal
```

F. Verifications Based on Protocol Specifications

Specifying a protocol can play different roles. Of course, first and foremost, it can be used for documentation purpose, stating the precise manner in which a Web service can and must be used. Additionally, once such protocol specifications are provided, a number of verifications can then be performed.

Static verification of protocol usage: Given a protocol description expressed as an appropriate path or (ω-)regular expression, we are interested in checking whether a specific executable business process that uses this web service do obey the indicated protocol. We are currently in the process of developing a tool that will perform such analysis. This tool will be built around the CADP toolbox (Construction and Analysis of Distributed Processes, formerly known as "CAESAR/ALDEBARAN Development Package") [37].

More precisely, the goal is to analyze an executable process description written in BPEL4WS and check whether the process do conform to the protocol of the various web services used by the process. This tool will work as follows:

- Using the `BPWS4J` tool and API [38], a business processes P expressed in BPEL4WS will be parsed and analyzed and the various Web services used by P will be identified.
- For each key service, the appropriate (slice of) flow of control within P will be analyzed and an algebraic process model representing this flow will be generated—LOTOS [28], one of the key language supported by the CADP toolbox, will be used as our target algebraic process model.
- The path expression describing the protocol of a service will be translated into an appropriate regular alternation-free mu-calculus expression [36], one of the logic notation supported by the CAPD toolbox.
- Conformance of the abstract model of P with the service's protocol will then be verified using the `evaluator` model-checking tool of CADP.

Dynamic verification of protocol usage: A static analysis cannot, in general, be exact and complete. Whenever static analysis cannot ensure that a process does conform to a protocol, it could be interesting to introduce a form of *run-time monitoring*. Such dynamic monitoring would be the equivalent, for protocol description, of the dynamic verification of pre/post-conditions and assertions in the case of contracts. How to integrate such dynamic monitoring into executable process is still under investigation. Work done in the context of security checks, e.g., using Schneider's security automata [39], should be useful in this regard.

G. Related Work

Nakajima [40] applies model-checking to Web services described in WSFL (Web Services Flow Language), a net-oriented specification language based on a workflow description language. The verification is performed by translating WSFL descriptions into Promela, where properties to be checked are expressed in LTL. One problem appears to be that WSFL's operational semantics does not correctly handle some types of dataflow, a problem which has been corrected in BPEL4WS using a form of dead path elimination (DPE).

Narayanan and McIlraith [41] start with the DAML-S ontology for Web services, for which they provide a formal semantics defined by mapping DAML-S into PSL (Process Specification Language). This PSL semantics is then translated into Petri nets, on which various analysis are performed, e.g., reachability, liveness, and deadlock detection. Model-checking of temporal properties, however, is not addressed.

Foster *et al.* [42] describe an approach to the model-based verification of Web services compositions using BPEL4WS. More precisely, LTSA-MSC (Labelled Transition System Analyzer extended with Message Sequence Chart) is first used to describe the intended behavior of the workflow using scenario-based specifications, which are then compiled into FSP (Finite State Processes), a textual notation for process calculus. A BPEL4WS implementation is then written and translated into FSP. Verifying that the implementation satisfies the specification is done through trace equivalence analysis of the resulting FSP representations (a form of bi-simulation checking). The types of properties that can be checked using this approach are not stated.

Koshkina and van Breugel [43] introduce the BPE-calculus, which targets the essence of BPEL4WS, focusing on the flow of control but abstracting from data and ignoring fault and compensation handlers. The formal syntax and (structural operational) semantics of the BPE-calculus are defined using the notation supported by PAC (Process Algebra Compiler), which then interfaces with the CWB (Concurrency Work-Bench) where model-checking is performed.

V. Conclusion

In this paper, we have presented some proposals toward adding specification of contracts and protocols to Web service descriptions, showing how these specifications could also be used for verification purposes. Admittedly, the work we have been presenting is still *in progress*, so that no result are available yet. However, we think the questions raised are important, since a number of standards or proposals for describing Web services have recently been, and still are, emerging. Understanding the strengths, and limits, of these proposals is important. Furthermore, we think that the development of Web services should also benefit from the same kind of conceptual tools, such as contracts, now available in other areas of software development.

Acknowledgment

Support for this research was provided by an NSERC research grant (# RGPIN183776).

References

[1] R. Chinnici, M. Gudgin, J.-J. Moreau, C. Schlimmer, and S. Weerawarana, "Web services description language (WSDL) version 2.0 part 1: Core language," August 2004, http://www.w3.org/TR/2004/WD-wsdl20-20040803.

[2] N. Kavantzas, D. Burdett, and G. Ritzinger, "Web services choreography description language version 1.0," April 2004, http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427.

[3] C. Peltz, "Web services orchestration and choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, October 2003.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, A. Trickovic, and S. Weerawarana, "Business process execution language for web services (BPEL4WS) version 1.1," May 2004, http://www-128.ibm.com/developerworks/library/ws-bpel.

[5] J. Bowen and M. Hinchey, "Formal models and the specification process," in *The computer science and engineering handbook*. CRC Press, 1997.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.

[7] C. Jones, *Systematic Software Development using VDM*. Prentice-Hall, 1986.

[8] J. Spivey, *Understanding Z: A specification language and its formal semantics*. Cambridge University Press, 1988.

[9] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[10] H. Mili, G. Bou Jaoude, E. Lefebvre, G. Tremblay, and A. Petrenko, "Business process modeling languages: Sorting through the alphabet soup," Dépt. d'Informatique, UQAM," Rapport de Recherche, Janvier 2004.

[11] J. Mendling, G. Neumann, and M. Nuttgens, "A comparison of XML interchange formats for business process modelling," in *Proc. of EMISA 2004 "Informationssysteme im E-Business und E-Government"*, ser. Lecture Notes in Informatics (LNI), F. Feltz, A. Oberweis, and B. Otjacques, Eds., vol. 56, Oct. 2004, pp. 129–140.

[12] Business Process Management Institute, "Business process modeling language," 2003, http://www.bpmi.org/bpml-spec.htm.

[13] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek, "Web service choreography interface (WSCI) 1.0," August 2002, http://www.w3.org/TR/wsci.

[14] J. Warmer and A. Kleppe, *The Object Constraint Language—Precise Modeling with UML*. Addison-Wesley, 1999.

[15] B. Meyer, *Object-Oriented Software Construction (Second edition)*. Prentice-Hall, 1997.

[16] O. Enseling, "iContract: Design by contract in Java," *Java World*, February 2001, http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

[17] A. Hunt and D. Thomas, *The Pragmatic Programmer—From journeyman to master*. Addison-Wesley, 2000.

[18] S. McConnell, *Code Complete — A Practical Handbook of Software Construction (Second Edition)*. Redmond, WA: Microsoft Press, 2004.

[19] S. McIlraith and H. Z. T.C. Son, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, March 2001.

[20] T. D. S. Coalition, "DAML-S: Semantic markup for web services," in *Semantic Web Working Symposium*, Stanford, CA, 2001, pp. 411–430.

[21] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[22] Y. Cheon and G. Leavens, "A quick overview of Larch/C++," Iowa State University, Dept. of Comp. Sci., Tech. Report 93-18, 1993.

[23] I. Curcio, "ASAP—a simple assertion pre-processor," *SIGPLAN Notices*, vol. 33, no. 12, pp. 44–51, 1998.

[24] R. Plosch, "Design by contract for Python," in *Joint Asia Pacific Software Engineering Conference*, Hong Kong, Dec. 1997, http://citeseer.nj.nec.com/257710.html.

[25] J. Rumbauch, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

[26] "Wikipedia, the free encyclopedia," http://en.wikipedia.org/wiki.

[27] C. Hoare, *Communicating sequential processes*. Prentice Hall, 1985.

[28] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1987.

[29] R. Milner, *Communication and concurrency*. Prentice-Hall, 1989.

[30] U. Furbach, "Formal specification methods for reactive systems," *J. of Systems and Software*, vol. 21, no. 2, pp. 129–139, Feb. 1993.

[31] M. Ardis, J. Chaves, L. Jagadeesan, P. Mataga, C. Puchol, M. Staskauskas, and J. Von Olnhausen, "A framework for evaluating specification methods for reactive systems—experience report," *IEEE Trans. on Soft. Eng.*, vol. 22, no. 6, pp. 378–389, 1996.

[32] K. Schneider, *Verification of Reactive Systems—Formal Methods and Algorithms*. Springer, 2004.

[33] R. Campbell and A. Habermann, "The specification of process synchronization by path expressions," in *Operating Systems*, E. Gelenbe and C. Kaiser, Eds. Springer-Verlag, LNCS-16, 1974, pp. 89–102.

[34] S. Andler, "Predicate path expressions," in *Annual Symposium on Principles of Programming Languages, 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Association for Computing Machinery. San Antonio, Texas: ACM Press, 1979, pp. 226–236.

[35] F. Lustman, "Specifying transaction-based information systems with regular expressions," *IEEE Trans. on Soft. Eng.*, vol. 20, no. 3, pp. 207–217, March 1994.

[36] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," INRIA, Rocquencourt, Tech. Rep. 2899, Mars 2000.

[37] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "CADP: A protocol validation and verification toolbox," in *Computer Aided Verification*. Springer-Verlag, LNCS-1102, 1996, pp. 437–440.

[38] IBM, "IBM business process execution language for web services Java run time (BPWS4J)," 2004, http://www.alphaworks.ibm.com/tech/bpws4j.

[39] B. Alpern and F. Schneider, "Verifying temporal properties without temporal logic," *ACM Transactions on Programming Languages*, vol. 11, no. 1, pp. 147–167, January 1989.

[40] S. Nakajima, "Model-checking verification for reliable web service," in *OOPSLA Workshop on Object-Oriented Web Services*, 2002.

[41] S. Narayanan and McIlraith, S.A., "Simulation, verification and automated composition of web services," in *Proceedings of the eleventh international conference on World Wide Web*. Honolulu, Hawaii, USA: ACM Press, 2002, pp. 77–88.

[42] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions," in *18th IEEE International Conference on Automated Software Engineering*. Montreal, QC, Canada: IEEE Computer Society Press, 2003, pp. 152–163.

[43] M. Koshkina and F. van Breugel, "Verification of business processes for web services," Deparment of Computer Science, York University, Toronto, Ontario, Tech. Rep. CS-2003-11, 2003.