Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1215-1233
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

PEWS: A New Language for Building Web Service Interfaces

Cheikh Ba
(Université François Rabelais - LI/Campus de Blois - France
cheikh.ba@etu.univ-tours.fr)

Marcos Aurélio Carrero
(Federal University of Paraná, Brazil
carrero@inf.ufpr.br)

Mírian Halfeld Ferrari
(Université François Rabelais - LI/Campus de Blois - France
mirian@univ-tours.fr)

Martin A. Musicante
(Federal University of Paraná, Brazil
mam@inf.ufpr.br)

Abstract: Recent proposals in the domain of interface description languages for web services stress the importance of specifying the dynamic, behavioral aspects of the services. The goal of this paper is to introduce a new interface description language, called PEWS, that uses predicate path expressions to define web service behaviours. Our proposal represents a simple but expressive way to describe order and conditional constraints over web service operations. PEWS aims to be used not only to the specification of simple web services but also to be a tool for describing service composition.

In this paper, we use the Action Semantics framework to present the syntax and semantics of the most significant parts of PEWS and we introduce XPEWS, the XML-based version of PEWS used to publish service behaviours for future searches and composition. The definition of XPEWS is done by giving the XML Schema that defines the syntax of XPEWS programs.

Key Words: Programming languages, web services, formal semantics.
Category: D.3.m, C.2.4, D.3.1

1 Introduction

Web[1] services are software applications accessible via Internet. They are typically designed to participate to composed applications where different web services should interact. The composition of new web services from existing ones is a

[1] This paper is an extended version of [Ba et al., 2005a]. Some details on the definition of XPEWS (the XML-based version of PEWS) and its implementation given in this work were not present in that paper.

challenging problem since it is necessary to ensure the correct interaction of independent, communicating software pieces.

Web service interfaces are usually specified by using XML-based languages. Interface languages, such as WSDL, address only static interface specifications and they do not describe the *observable behaviour* of the web service (*i.e.*, they do not specify the order between operations). WSDL interfaces do not inform how interaction with the service can be built, they do not allow the verification of the system dynamics and do not fulfill the recent requirements of W3C choreography working group [Austin et al., 2004].

Indeed, in order to address the challenging issues of web service composition (such as synchronization, coordination, composition maintenance and manipulation), different formalisms [Berardi et al., 2003a, Hamadi and Benatallah, 2003, Hull et al., 2003, Salaün et al., 2004] have already been proposed.

In this paper, we use predicate path expressions [Andler, 1979] to define web service behaviour and we introduce a new language, called PEWS (Path Expressions for Web Services), as a complement to other interface description languages (such as WSDL). PEWS aims to be used not only in the *specification* of simple or composite web services, but also as an implementation language for them. A PEWS program acts as an upper layer for the program implementing the actions involved in the data processing of the web service.

PEWS specifies the implementation of a web service interface. Indeed, PEWS is used to specify a system and serves as a guide to the implementation of each operation involved in this system. Then, given a set of operations (or methods) already implemented, a web service can be implemented by following a behaviour description presented by PEWS programs.

Our approach brings predicate path expressions to the context of web services. Predicate path-expressions [Andler, 1979] were introduced as a tool to express the synchronization of operations on data objects. Path expressions are programming language constructs used to restrict the allowable sequences of operations on an object. For instance, given the operations a, b and c, the path expression $a^*.(b \mid\mid c)$ defines that the parallel execution of operations b and c should be preceded by zero or more executions of a.

As noted in [Andler, 1979], the use of predicates in path expressions allow a finer control of the access to the object being manipulated. For instance, the predicate path expression $a^*.([P]b + [\mathsf{not}\ P]c)$ indicates that b or c would be executed according to the truth-value of predicate P.

In [Berardi et al., 2003b], the authors stress the importance of taking the notion of time into account when dealing with web services. Indeed, time is needed to understand the dynamics of transactions and compositions. It also plays a role when we want to impose some time-out constraints to the service. In this paper, we extend the proposal in [Andler, 1979] by taking the notion of time into account.

The paper is structured as follows. In Section 2 we present a motivating example of the use of predicate path expressions for web services. Section 3 provides some background notions used in this paper. The syntax and semantics of PEWS are given in Section 4 using the action semantics framework [Mosses, 1992]. Section 5 describes XPEWS, the XML-based version of PEWS, and its project. Finally, Section 6 discusses some related work and concludes our paper.

2 Overview of our Proposal

Interface description for web services is a crucial point, having consequences for service discovery, compatibility, verification and composition [Hull et al., 2003]. Recent proposals in this domain stress the dynamic aspects of the service, describing the behaviour of web service interfaces over several interactions. Our approach can be placed in this new context, proposing a simple but expressive way to describe order and conditional constraints over web service operations.

To illustrate our approach, we adapt an example from [Hull et al., 2003] where the interface of a web service for a warehouse is considered. The warehouse service interacts with other services to perform the following operations: receive an *order*, send a *bill*, receive a *payment* and send a *receipt*.

We suppose different kinds of warehouses[2] (applying different procedures for dealing with payments and receipts) and we consider their descriptions using PEWS. Indeed, our proposal allows us to describe different web service behaviours, each one corresponding to a different kind of warehouse.

As a first behaviour, we consider the "cautious" warehouse of [Hull et al., 2003]. In this case, the warehouse behaviour is described by the following path expression that constraints a receipt to be sent only if a payment has already been received.

$$(\mathsf{order.bill.payment.receipt})^*$$

This path expression specifies how a warehouse deals with the orders it receives: one order is treated at a time and the operations are performed in a sequence. A different situation can be described by the expression:

$$\{\mathsf{order.bill.payment.receipt}\}$$

where each received order is treated sequentially, but an unbounded amount of orders can be treated in parallel.

The "trusting" warehouse of [Hull et al., 2003] allows for two activities to proceed in an interleaved fashion: the sending of a receipt and the sequence

[2] In this way, one can choose which policy is more adequate for his own warehouse.

of sending a bill and receiving a payment. This behaviour is described by the following path expression

$$\{\mathsf{order}.(\mathsf{bill}.\mathsf{payment} \mid\mid \mathsf{receipt})\}$$

The precise syntax and semantics of PEWS is defined in Section 4. The path expression model gives us the possibility to reason over web service properties (*e.g.*, by adapting the verification techniques proposed in [Andler, 1979, Musicante, 1999]). However, for storing service description for advertising, searching, composing, etc. we need to translate PEWS into a concrete, standard-compliant representation. To this end, we introduce XPEWS in Section 5. XPEWS is an XML-based language that extends WSDL interface descriptions by adding behavioural constraints.

3 Background

This section summarizes two topics that will be used in our work, namely the Action Semantics formalism and WSDL. The first one is a formal description framework used to specify the syntax and s semantics of PEWS. WSDL represents the traditional interface description languages that PEWS intends to extend, *i.e.*, those that focused on defining the messages that could be passed in and out of a web service in a single interaction.

3.1 Action Semantics

Action semantics [Mosses, 1992, Watt, 1991] is a formal framework for semantic specification, developed to provide "readable" descriptions of real-life languages [Doh and Mosses, 2003, Menezes and Moura, 2001, Duarte Jr. and Musicante, 1999]. Action semantic descriptions are *compositional*, *i.e.*, they define semantic functions to map abstract syntax objects to semantic entities. Semantic functions are defined inductively using equations. The semantic entities are *actions*, ad-hoc entities which provide a natural way to describe computations.

Action semantics uses a special notation to describe actions. This notation is called *action notation*, and it is used in action semantic descriptions very much in the same way as the λ-notation is used in denotational semantics. The symbols used in action notation are intentionally verbose, so that English-like phrases can be used—completely formally—to express most of the concepts present in computing.

Action semantics has features that are similar to other semantic formalisms. It is similar to denotational semantics which uses semantic functions to describe the meaning of objects. However, actions have a more operational flavor than functions. In this sense, action semantics bridges the gap between denotational and operational semantics [Winskel, 1993].

Actions are used to describe the meaning of computation. Actions can be *performed* to process *information*, with various possible outcomes: normal termination (performance of the action *completes*), exceptional termination (it *escapes*), unsuccessful termination (it *fails*) or non-termination (it *diverges*). Action notation provides some primitive actions, and various *combinators* for forming complex actions, corresponding to the main fundamental concepts of programming languages.

A *data notation* is used to describe the information processed by actions. The standard data notation (included in action notation) provides a collection of algebraically defined abstract data types, including numbers, characters, strings, sets, tuples, maps, etc.; further data may be specified *ad hoc*.

There is also a third class of entities in action notation, called *yielders*. A yielder represents unevaluated data, whose value depends on the current information available to the primitive action in which it occurs. Yielders are *evaluated* to yield data. An example of a standard yielder is the data bound to I, which depends on the current bindings that are received by the enclosing primitive action.

Actions can represent pure control, or can process different types of information. The so-called 'facets' of an action represent the behaviour of the action. Each facet deals with one aspect of the information processed by the action. There exists five facets of each action:

Basic: This facet deals with pure control flow, without reference to information processing issues.

Functional: This facet deals with *transient* data, which is given to or by an action. For example, when the primitive action give the successor of the given natural is given a natural number n as transient data, it completes, giving $n + 1$ as a transient. The compound action A_1 then A_2 performs the action A_1 first; all transient data given by A_1 is passed on to A_2, which is performed after A_1 completes. The primitive action choose D, where D is a sort of data, makes a non-deterministic choice of an individual of sort D, giving the chosen datum as a transient.

Declarative: This facet deals with the manipulation of *scoped* information, represented by associations of *tokens* to bindable data. For example, performance of the primitive action bind "max-length" to 256 completes, producing a binding of the token "max-length" to the natural number 256.

Imperative: This facet is concerned with *storage* handling. A storage in action notation is simply a mapping from memory cells to storable data. For example, consider the action allocate a cell then store 26 in the given cell, which combines features of the functional and imperative facets.

Communicative: This facet provides a system of *agents*, which can each be 'contracted' to perform particular actions. Initially only a special 'user' agent is active. Agents can communicate using asynchronous message passing. Each agent has its own *communication buffer*, in which all the messages sent to the agent are placed. Arbitrary data can be contained in messages.

3.2 WSDL

WSDL (Web Service Description Language) [Curbera et al., 2002, Christensen et al., 2001] is an XML-based language developed by IBM and Microsoft to describe Web service interfaces. The goal of a WSDL service description is twofold: (*i*) to provide an application-level service description, *i.e.*, to indicate which operations (or methods) the service exposes and (*ii*) to give protocol-dependent details necessary to access the service. In this section we just consider how WSDL specifies an abstract interface (*i.e.*, just the first goal is discussed).

A WSDL abstract description specifies the messages taking part in a service interaction. External type systems (*e.g.*, the one of XML Schema) are used to provide data type definitions for the information exchange. Messages provide an abstract, typed data definition sent to and from the services. Operations are lists of messages, defining the interactions the Web service supports. An *operation* combines messages labeled as input, output, or fault to indicate what part a particular message plays in the interaction.

A *portType* is a collection of operations that are collectively supported by an end point. In this way, it describes a set of messages that a service sends and/or receives. A portType element can be compared to a function library (or a module, or a class) in a traditional programming language.

The following example shows an WSDL abstract description.

Example 3.1 The WSDL fragment given in figure 1 shows data type definitions (with types integer, string, etc from XML Schema) and the portType element that groups four different operations, namely, `order, bill, payment` and `receipt`.
The `order` operation is a request-response operation since it expects the `productOrderIn` as input and returns a `productOrderOut` as the response. On the other hand, the `bill` operation is a solicit-response operation having `sendBill` as a request and `ackBill` as a response. The `payment` operation is a one-way operation which takes message `getPayment` as input while the `receipt` operation is a notification operation which just sends an output message. □

From Example 3.1 we understand that WSDL defines four types of operations. In a *request-response* operation, a client makes a request, and the web

```
<message name="productOrderIn">
  <part name= "prodCode"
        type="xsd:string"/>
  <part name= "quantity"
        type="xsd:integer"/>
</message>
<message name="productOrderOut">
  <part name= "price"
        type="xsd:real"/> ...
</message>
<message name="sendReceiptClient">
  <part name= "prodCode"
        type="xsd:string"/>
  <part name= "datePay"
        type="xsd:date"/>  ...
</message>
<message name="sendBill">
  <part name= "prodCode"
        type="xsd:string"/>
  <part name= "total"
        type="xsd:real"/>
...
</message>
<message name="ackBill">
  <part name= "prodCode"
        type="xsd:string"/>
...
</message>
...
```

```
...
<portType name="WarehousePortType">
   <operation name="order">
        <input message="productOrderIn"/>
        <output message="productOrderOut"/>
   </operation>

  <operation name="bill">
        <output message="sendBill"/>
        <input message="ackBill"/>
   </operation>

  <operation name="payment">
        <input message="getPayment"/>
   </operation>

   <operation name="receipt">
        <output message="sendReceiptClient"/>
   </operation>
</portType>
```

Figure 1: Operations defined in WSDL.

service responds to it while in a *solicit-response* operation, the web service sends a message to the client (output before input) and the client responds. *One-way* operations indicates that a client sends a message to the web service but expects no response. Finally, in a *notification* operations a web service sends a message to the client but expects no response.

4 The PEWS Language

In this paper we use Predicate Path Expressions (PPE) [Andler, 1979] to describe the behaviour of web service interfaces.

Path expressions are programming language constructs used to restrict the allowable sequences of operations on an object. They were introduced as a technique for specifying process synchronization [Campbell and Habermann, 1974, Campbell, 1977]. They propose a static description of the order in which operations are performed, allowing the operation's code to be written without any explicit reference to synchronization primitives. Many versions of path expressions, defining different kinds of operation combinations, have been proposed [Campbell and Habermann, 1974, Flon and Habermann, 1976, Habermann, 1975, Bruegge and Hibbard, 1983, Andler, 1979]. Most proposals include sequential, parallel and non deterministic choice for defining the combination of operations.

Predicate Path Expressions [Andler, 1979] extend basic path expressions by adding predicates. Predicate path expressions preserve the advantages of original path expressions, but are more powerful since they may contain history variables and predicates.

In this paper we introduce *predicate path-expression for web services* (PEWS), a language that allows to specify the order in which the operations of a given web service can be performed. In our proposal we use a notation based on that of [Andler, 1979].

In PEWS, a path expression combines web service operation names using the operators sequencing (.), exclusive choice (+), parallel (||), sequential repetition (*), parallel repetition ({...}) and predicate prefixing ([...] ...) as defined below.

PEWS Syntax:

grammar:

```
(1)  interface  = ⟦ portType def* path ⟧

(2)  path       = ⟦ opname ⟧ | ⟦ path "." path ⟧ | ⟦ path "+" path ⟧ |
                  ⟦ path "||" path ⟧ | ⟦ path "*" ⟧ | ⟦ "{" path "}" ⟧ |
                  ⟦ "[" pred "]" path ⟧

(3)  pred       = ⟦ "true" ⟧ | ⟦ "false" ⟧ | ⟦ "not" pred ⟧ |
                  ⟦ pred boolOp pred ⟧ | ⟦ arith-expr relOp arith-expr ⟧

(4)  def        = ⟦ "def" var "=" arith-expr ⟧

(5)  portType   = ⟦ operation+ ⟧

(6)  operation  = ⟦ opname "("opArg ")"⟧

(7)  opArg      = ⟦"in:" msgName⟧ | ⟦"out:" msgName⟧ |
                  ⟦"in-out:" msgName ", " msgName ⟧ |
                  ⟦"out-in:" msgName ", " msgName ⟧

(8)  msgName    = □

(9)  opname     = □

(10) arith-expr = ⟦ var ⟧ | ⟦ arith-expr arithOp arith-expr ⟧ | ⟦ "now()" ⟧ |
                  ⟦ "act(" opname ").val" ⟧ | ⟦ "act(" opname ").time" ⟧ |
                  ⟦ "term(" opname ").val" ⟧ | ⟦ "term(" opname ").time" ⟧

(11) boolOp     = □

(12) relOp      = □
```

(13) arithOp $= \Box$

(14) var $= \Box$

In the grammar above, the notation $\Box$ means that the right-hand side of the rule is not defined here (all the cases above are straightforwardly specified.

An interface is defined as sequence of definitions (def) followed by a path expression. Each definition declares integer variables to be used in predicates. The value of variables is obtained by the evaluation of an arithmetic expression which involves predefined counters and library functions. Each counter is assumed to be a pair of integers $(val, time)$. The val component represents the counter itself while the $time$ component indicates the moment the counter was last modified. We suppose the existence of three counters for each web service operation O:

req(f): The val component describes the number of times a caller has attempted to perform the operation f. The $time$ component indicates the moment of the last request.

act(O): The val component describes the number of times a caller has started to perform the operation O. The $time$ component indicates the moment of the last activation of the service.

term(O): The val component describes the number of times a caller has terminated to perform the operation O. The $time$ component indicates the moment of the last conclusion of the service.

Example 4.1 In our warehouse example, suppose that a payment should be done within 48 hours after the bill has been sent. If the deadline is not respected the whole process is aborted. A path expression modeling this situation is:

def t^{pay} = now() - term(bill).$time$

(order.bill.([$t^{pay} \leq 48h$] payment.receipt + [$t^{pay} > 48h$] abortOperation))*

The value of the variable t^{pay} is computed by evaluating the expression now() - term(bill).$time$ which involves function now() (that computes the current time). The predicates appearing in the above expression represent guards, giving to the expression the semantics of a conditional construct. Guards are evaluated until one of them is true. Notice that the value of the variable t^{pay} changes at each evaluation, since it depends on the current time. $\Box$

PEWS Semantics:

- execute _ : interface → Action

```
(1)  execute ⟦ $T$: portType $D$: def* $P$: path ⟧ =
        |elaborate ⟦$T$⟧
        before elaborate ⟦$D$⟧
        before execute ⟦$P$⟧
```

We define the semantic function execute over an interface. This function is responsible for the definitions of bindings for the service operations and for the integer variables to be used inside predicates. Moreover, execute also performs the actions established by the path expression of the interface.

- execute _ : path → Action

```
(1)  execute ⟦ $O$:opname ⟧ =
        |indivisibly
        |||give the natural stored in the cell bound to act($O$).$val$
        |||then store successor of it in the cell bound to act($O$).$val$
        ||and store current-time in the cell bound to act($O$).$time$
        and then
        |enact the service bound to $O$
        and then
        |indivisibly
        |||give the natural stored in the cell bound to term($O$).$val$
        |||then store successor of it in the cell bound to term($O$).$val$
        ||and store current-time in the cell bound to term($O$).$time$
```

The execution of the service operation is preceded by the update of the counter act, and it is followed by the update of the counter term. These updates consist on the increment of the component *val* and on the use of the yielder current-time for setting the component *time* of each counter. The yielder current-time is not part of the standard action semantics framework. We suppose that it maintains the actual, absolute, global time of the system. This yielder represents a call to a system function in a real-life implementation.

(2) execute ⟦ P_1:path "." P_2:path ⟧ = execute ⟦ P_1 ⟧ and then execute ⟦ P_2 ⟧

(3) execute ⟦ P_1:path "+" P_2:path ⟧ = execute ⟦ P_1 ⟧ or execute ⟦ P_2 ⟧

(4) execute ⟦ P_1:path "||" P_2:path ⟧ = execute ⟦ P_1 ⟧ and execute ⟦ P_2 ⟧

The equations above define different composition operations over path expressions; namely, sequential composition, non deterministic choice and parallel composition.

```
(5)  execute ⟦ $P_1$:path "*" ⟧ =
        unfolding
        ||complete
        |or
        ||execute ⟦ $P_1$ ⟧ and then unfold
```

```
(6)  execute ⟦ "{" $P_1$:path "}" ⟧ =
         unfolding
         |execute ⟦ $P_1$ ⟧ and unfold
```

The star operator defines the unbounded, sequential, repetitive execution of a path expression P_1. The $\{\_\}$ operator specifies the unbounded parallel execution of a path expression P_1.

```
(7)  execute ⟦ "[" $B$:pred "]" $P_1$:path ⟧ =
         unfolding
         ||evaluate ⟦ $B$ ⟧ and enabled ⟦$P_1$⟧
         |then
         |||check (the given tuple is ⟨ true, true ⟩)
         ||and then commit and then execute ⟦ $P_1$ ⟧
         ||or
         |||check not (the given tuple is ⟨ true, true ⟩)
         ||and then unfold
```

The above definition considers the case where the execution of a path expression P_1 depends on the result of evaluating a given predicate B. In order to execute P_1, the action waits until the following two conditions are simultaneously verified: (i) the evaluation of B yields $true$ and (ii) P_1 is ready to be executed. This second condition, specified by enabled _, consists in verifying whether input messages are available to be read by P_1 (if P_1 starts with a response-request or a one-way operation).

- elaborate _ : portType $\rightarrow$ Action

```
(1)  elaborate ⟦$Q_1$:operation+ $Q_2$:operation+ ⟧ =
         elaborate ⟦$Q_1$⟧ and elaborate ⟦$Q_2$⟧

(2)  elaborate ⟦ $O$:opname "(" $A$:opArg ")" ⟧ =
         |allocate a cell then
         ||store 0 in it and bind it to act($O$).$val$
         and
         |allocate a cell then bind it to act($O$).$time$
         and
         |allocate a cell then
         ||store 0 in it and bind it to term($O$).$val$
         and
         |allocate a cell then
         ||store never in it and bind it to term($O$).$time$
         and
         |bind kindOf ⟦ $A$ ⟧ to kind($O$)
         and
         |GetAbstractionForService($O$) then bind it to $O$
```

The elaboration of the portType definitions creates the counters for each operation (each counter is formed by a pair of cells). The action*GetAbstractionForService_* represents the identification of an (external) abstraction, to be associated to the service operation. In a real-life implementation, this function is performed by the HTTP server.

- elaborate _ : def* $\rightarrow$ Action

(1) elaborate $[\![\ ]\!]$ = complete

(2) elaborate $[\![$ G_1: def* G_2: def* $]\!]$ = elaborate $[\![$ G_1 $]\!]$ and elaborate $[\![$ G_2 $]\!]$

(3) elaborate $[\![$ "def" I:var "=" E:arith-expr $]\!]$ =
bind I to closure abstraction of evaluate $[\![$ E $]\!]$

Variable declarations in PEWS are a simple form of function definition. The expression that defines a variable will be evaluated each time the variable is needed during the execution of a path expression.

- evaluate _ : arith-expr $\rightarrow$ Action[giving an (integer | time)]
- evaluate _ : pred $\rightarrow$ Action[giving a truth-value]
- enabled _ : path Action[giving a truth-value]

```
(1)  enabled ⟦ O:opname ⟧ =
         indivisibly
           | check (the datum bound to kind(O) is a (out | out-in)) and then give true
           or
           | check (the datum bound to kind(O) is a (in | in-out))
           | and then
           |   | choose a message[containing ⟨ O, data ⟩]
           |   |                   [in set of items of the current buffer]
           |   then
           |   | check (it is a message) and then give true
           |   | or
           |   | check (it is nothing) and then give false
```

(2) enabled $[\![$ P_1:path "." P_2:path $]\!]$ = enabled $[\![$ P_1 $]\!]$

(3) enabled $[\![$ P_1:path "*" "." P_2:path $]\!]$ = enabled $[\![$ P_1 $]\!]$ or enabled $[\![$ P_2 $]\!]$

(4) enabled $[\![$ P_1:path "+" P_2:path $]\!]$ = enabled $[\![$ P_1 $]\!]$ or enabled $[\![$ P_2 $]\!]$

(5) enabled $[\![$ P_1:path "||" P_2:path $]\!]$ = enabled $[\![$ P_1 $]\!]$ or enabled $[\![$ P_2 $]\!]$

(6) enabled $[\![$ P_1:path "*" $]\!]$ = enabled $[\![$ P_1 $]\!]$

(7) enabled $[\![$ "{" P_1:path "}" $]\!]$ = enabled $[\![$ P_1 $]\!]$

(8) $\textsf{enabled} \; [\![\; \text{``[''} \; B\textsf{:pred} \; \text{``]''} \; P_1\textsf{:path} \;]\!] = \textsf{enabled} \; [\![\; P_1 \;]\!]$

The semantic function enabled _ returns a truth-value, being true iff the leading operation in the expression is enabled to be executed, *i.e.*, if it is an output or output/input operation or if there is a message waiting for it in the communications buffer.

The semantic equations given in this section present the most significant part of the action semantics definition of PEWS. The complete set of equations can be found in [Ba et al., 2005b]. Notice that although we consider that a web service can have different behaviours (as illustrated in Section 2), in this section, we have just presented how to define a service with just one behaviour. This extension is straightforward.

5 XPEWS

So far in this paper we used a user-friendly syntax for PEWS programs. This section presents XPEWS the XML, machine-readable version of the language. XPEWS programs can be generated from PEWS in an intuitive way. For instance, the PEWS program

def t^{pay} = now() - term(bill).$time$

(order.bill.([$t^{pay} \leq 48h$] payment.receipt + [$t^{pay} > 48h$] abortOperation))*

is written in XPEWS as shown in Figure 2.

The root element of a XPEWS document is the `<envelope>`. It can contain the definition of the possible behaviours of the web service. For instance, when defining the access to a certain piece of data, we can specify different orders in which the instances of these data can be accessed. Figure 2 shows just one behaviour for our warehouse service. It takes into account a 48 hour timeout condition between the placement of an order and the payment reception.

The `<behaviour>` construct specifies the way in which the client can interact with the web service. This is done by a predicate path expression involving the operations of the service whose interface is being defined. Notice that element `<behaviour>` has two attributes. The `name` attribute identifies the element. The `target` attribute refers to the portType (in an WSDL file) whose operations are involved in the behaviour being defined (see Section 3.2).

The PEWS implementation is in its early stage of developement. The system is formed by a front-end and a back-end.

The front-end is a WSDL-aware, syntax-directed editor for PEWS programs, which is being implemented as a plugin extension for the the Eclipse

```
<envelope xmlns="http://aquarius.inf.ufpr.br"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://aquarius.inf.ufpr.br pews.xsd">

<behaviour name="timeOutEx"
           xmlns:ns1="http://www.aquarius.inf.ufpr.br/warehouse">
  <operations>
    <operation name="bill"
               portType="WarehousePortType"
               refersTo="ns1:bill"/>
    <operation name="order"
               portType="WarehousePortType"
               refersTo="ns1:order"/>
    <operation name="payement"
               portType="WarehousePortType"
               refersTo="ns1:payement"/>
    <operation name="receipt"
               portType="WarehousePortType"
               refersTo="ns1:receipt"/>
  </operations>
  <varDef>
    <minus>
      <libFunction name="now"
                   unit="hours"/>
      <pewsCounter opname="bill"
                   name="term"
                   component="time"
                   unit="hours"/>
    </minus>
  </varDef>
  <pathExp>
    <star>
      <seq>
        <operation name="order"/>
        <operation name="bill"/>
        <choice>
          <seq>
            <pred>
              <leq>
                <var name="tpay"/>
                <const value="48"/>
              </leq>
              <operation name="payement"/>
            </pred>
            <operation name="receipt"/>
          </seq>
          <pred>
            <gt>
              <var name="tpay"/>
              <const value="48"/>
            </gt>
            <operation name="abortOperation"/>
          </pred>
        </choice>
      </seq>
    </star>
  </pathExp>
</behaviour>
</envelope>
```

Figure 2: Translation of the warehouse service into XPEWS.

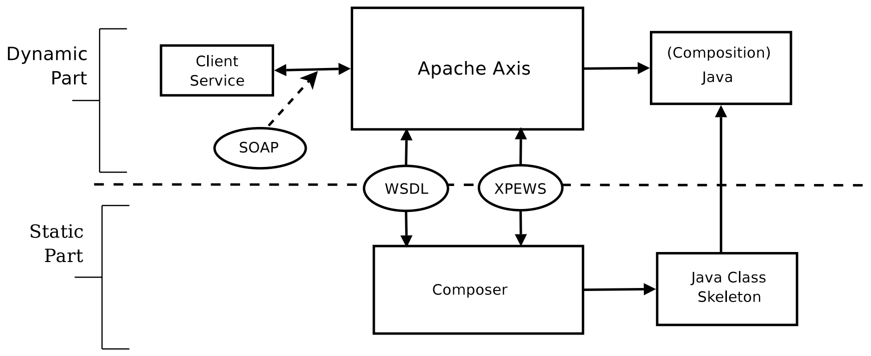

Figure 3: XPEWS back-end tool.

Platform [ecl, 2005]. The font-end tool includes a type checker and a generator of XPEWS programs. The translation from PEWS to XPEWS follows the XMLSchema definition of XPEWS, as given in Figure 2.

The back-end tool generates java classes from XPEWS programs. This tool is schematized in figure 3. The static part of the tool is formed by a composer, which reads an interface description and generates a java class skeleton for the service. The interface description is formed by WSDL and XPEWS documents.

The class skeleton generated by the composer uses the JSCP library [jsc, 2004] to implement the synchronization part of the service (*i.e.*, the path expressions). The generated skeleton is the basis for the implementation of the service. The java program (composition) in figure 3 represents the user extension of this eskeleton to implement the service.

The dynamic part of the back-end is the runtime system of the language. The program is registered as a service at the http server, and communicates with clients (and other services) using SOAP.

6 Related Work and Concluding Remarks

In order to correctly place our work in the domain of service description languages, we adopt the general framework proposed in [Salaün et al., 2004] which allows to classify service languages into three layers, namely:

1. The *abstract layer*, used to reasoning. On this level we can put together different formal proposals such as process algebra, finite state automata, etc.

2. The *public level* or *interface layer*, containing XML-based languages that allow the description of the service behaviour.
3. The *private* or *implementation level*, containing the languages used to implement services.

In this context, we place PEWS at the most abstract level while XPEWS can be placed at the public (interface) level.

In the abstract level, several works propose formal tools to reason over web services and to ensure some properties. The main reason is that the use of a formal model allows the verification of properties and the detection of inconsistencies and deadlocks, both within and between services. Their goal is usually to define service descriptions formally. Thus, we cannot expect for web services specifications to succeed based on implementation languages only.

Several models have appeared trying to formalize synchronization of concurrent processes (such as in services composition). In [Hamadi and Benatallah, 2003] the authors propose Petri Nets for modeling composition. In [Meredith and Bjorg, 2003] services are modeled as mobile processes and their composition is verified using π-calculus. In [Salaün et al., 2004] the authors advocate the use of process algebra to describe and compose web services at an abstract level while [Berardi et al., 2003b] use finite state automata as a conceptual model.

In this paper, we propose the use of predicate path expressions to restrict the allowable sequences of operations on a web service. To this end, we define an interface description language called PEWS. We use the Action Semantics framework to define the semantics of PEWS. Similarly to other abstract models, the path expression model gives us the possibility to reason over web service properties.

When we consider the interface layer, we notice that, besides WSDL, some web service description languages have already been proposed. For instance, WSCL (Web Services Conversation Language) [Banerji et al., 2002] is an XML-based language which models the conversation supported by a service; WSCI (Web Service Choreography Interface) [Arkin et al., 2002] is an evolution of WSCL. It is an XML-based interface description language that describes the flow of messages exchanged by a web Service participating in choreographed interactions with other services. To the best of our knowledge, the semantics for these languages are not defined formally.

Our proposal allows an automatic translation between the user-friendly syntax of PEWS to its XML-based version XPEWS. Moreover, given an XPEWS service description and the implementation of its individual operations, our proposal includes the possibility of an automatic implementation of the service. We are currently working on sound mappings between the abstract layer, the interface layer and the concrete layer for our proposal.

In this paper, we present PEWS as a language for describing web service behaviour, but we are currently considering its generalization in order to specify web service composition. In fact, since the development of web service composition languages have been mainly driven by software vendors, too many standards have been proposed, usually having overlapping functionality [van der Aalst, 2003]. These propositions include XLANG [Thatte, 2001], from Microsoft, which specifies message exchange behaviour among the participating web services for automation and composition of new business processes; WSFL (Web Services Flow Language) [Leyman, 2001], from IBM, an XML language for the description of Web Services compositions; BPML (Business Process Modeling Language) [Intalio and BPMI.org., 2002] which is a meta-language for modeling and an abstracted execution model for collaborative and transactional business process based on the concept of transactional finite-state machine; BPEL4WS (Business Process Execution Language for Web Services) [Andrews et al., 2003] combines the graph oriented process representation of WSFL and the structural construct based processes of XLANG into a unified standard for web services composition and ebXML [ebXML Team, 2001], a specification that enables enterprises to conduct business over the Internet using an open XML-based infrastructure. It is also important to notice the work on a web service markup language called DAML-S [Coalition, 2004] aiming at providing service providers with a core set of markup language constructs for describing the properties and capabilities of their services in unambiguous, computer-interpretable (machine-understandable) form.

References

[jsc, 2004] (2004). Communicating sequential processes for java. http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[ecl, 2005] (2005). The eclipse project. http://www.eclipse.org.

[Andler, 1979] Andler, S. (1979). Predicate path expressions. In *Sixth Annual ACM Symposium on Principles of Programming Languages (6th POPL'79).*

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weeranwarana, S. (2003). Bussiness process execution language for web services. Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

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

[Austin et al., 2004] Austin, D., Barbir, A., Peters, E., and Ross-Talbot, S. (2004). Web services choreography requirements. Available at http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/. W3C Working Draft.

[Ba et al., 2005a] Ba, C., Ferrari, M. H., and Musicante, M. (2005a). Building web services interfaces using predicate path expressions. In *Proceedings of SBLP 2005.*

IX Brazilian Symposium on Programming Languages, pages 147–160, Recife - Brazil. Brasilian Computer Science Society, University of Pernambuco.

[Ba et al., 2005b] Ba, C., Halfeld Ferrari Alves, M., and Musicante, M. A. (2005b). PEWS: Predicate path expressions for web services. Technical Report LI (to appear), Université François Rabelais de Tours.

[Banerji et al., 2002] Banerji, A., Bartolini, C., Beringer, D., Chopella, V., Govindarajan, K., Karp, A., Kuno, H., Lemon, M., Pogossiants, G., Sharma, S., and Williams, S. (2002). Web services conversation language (wscl) 1.0. Available at http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[Berardi et al., 2003a] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Mecella, M. (2003a). Automatic composition of e-services. Technical Report 22-2003, Dipartimento di Informatica e Sistemistica, Universita di Roma La Sapienza, Roma, Italy.

[Berardi et al., 2003b] Berardi, D., de Rosa, F., de Santis, L., and Mecella, M. (2003b). Finite state automata as conceptual model for e-services. In *Integrated Design and Process Technology (IDPT)*.

[Bruegge and Hibbard, 1983] Bruegge, B. and Hibbard, P. (1983). Generalized path expressions: A high-level debugging mechanism. *Journal of Systems and Software*.

[Campbell, 1977] Campbell, R. H. (1977). Path expressions: A technique for specifying process synchronization. Report UIUCDCS-R-77-863, Dept. Comp. Sci., Univ. Illinois at Urbana-Champaign.

[Campbell and Habermann, 1974] Campbell, R. N. and Habermann, A. N. (1974). The specification of process synchronization by path expressions. *Lecture Notes in Computer Science*, 16.

[Christensen et al., 2001] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web services description language (WSDL) 1.1. Availabre at http://www.w3.org/TR/wsdl.

[Coalition, 2004] Coalition, D. S. (2004). Daml-s: Semantic markup for web services. Available at http://www.daml.org/services/.

[Curbera et al., 2002] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. (2002). Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*.

[Doh and Mosses, 2003] Doh, K.-G. and Mosses, P. D. (2003). Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36. Elsevier Science Publishers.

[Duarte Jr. and Musicante, 1999] Duarte Jr., E. P. and Musicante, M. A. (1999). Formal specification of SNMP mib's using action semantics: The routing proxy case study. In Publishing, I., editor, *Proc. of the Sixth IFIP/IEEE Int'l Symp. on Integrated Network Management*, Boston, USA.

[ebXML Team, 2001] ebXML Team (2001). ebxml requirements specification, version 1.06. Available at http://www.ebxml.org/specs/ebREQ.pdf.

[Flon and Habermann, 1976] Flon, L. and Habermann, A. N. (1976). Toward the construction of verifiable software systems. *Sigplan Notices*.

[Habermann, 1975] Habermann, A. N. (1975). Path expressions. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA.

[Hamadi and Benatallah, 2003] Hamadi, R. and Benatallah, B. (2003). A petri net-based model for web service composition. In Schewe, K.-D. and Zhou, X., editors, *Fourteenth Australasian Database Conference (ADC2003)*, volume 17 of *CRPIT*, pages 191–200, Adelaide, Australia. ACS.

[Hull et al., 2003] Hull, R., Benedikt, M., Christophides, V., and Su, J. (2003). E-services: a look behind the curtain. In ACM, editor, *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2003: San Diego, Calif., June 9–11, 2003.*

[Intalio and BPMI.org., 2002] Intalio and BPMI.org. (2002). Bussiness process modeling language. Available at http://www.bpmi.org/bpmi-downloads/BPML-SPEC-1.0.zip.

[Leyman, 2001] Leyman, F. (2001). Web services flow language (wsfl) 1.0. Availabre at http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf/.

[Menezes and Moura, 2001] Menezes, L. C. and Moura, H. (2001). Component-based action semantics: A new approach for programming language specifications. In *SBLP 2001 - V Brazilian Symposium on Programming Languages*, pages 152–163, Curitiba, Brazil. Universidade Federal do ParanÃ¡.

[Meredith and Bjorg, 2003] Meredith, L. and Bjorg, S. (2003). Contracts and types. *Communications of the ACM*, 46(10).

[Mosses, 1992] Mosses, P. D. (1992). *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[Musicante, 1999] Musicante, M. A. (1999). Formal semantics of interleaving. In *SBLP 99 - Proceedings of the III Brazilian Symposium on Programming Languages.*

[Salaün et al., 2004] Salaün, G., Bordeaux, L., and Schaerf, M. (2004). Describing and reasoning on web services using process algebra. In *Proceeding of the 2nd International Conference on Web Services, IEEE.*

[Thatte, 2001] Thatte, S. (2001). XLANG: Web services for business process design. Available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[van der Aalst, 2003] van der Aalst, W. M. P. (2003). Don't go with the flow: Web services compositions standards exposed. *Issue of IEEE Inteligent System.*

[Watt, 1991] Watt, D. A. (1991). *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice Hall.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing Series. MIT Press.