*Journal of Universal Computer Science, vol. 12, no. 7 (2006), 903-921*
*submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06* © *J.UCS*

# Expressing Workflow Patterns for Web Services: The Case of PEWS

**Martin A. Musicante**
(Federal University of Rio Grande do Norte, Natal, Brazil
mam@dimap.ufrn.br)

**Edinardo Potrich**
(Federal University of Paraná, Curitiba, Brazil
edinardo@inf.ufpr.br)

**Abstract:** PEWS is a language for the implementation of web service interfaces. PEWS programs can be used for the description of both individual and composed web services. Individual web services can be built up from Java programs. Composed web services are built from simpler services. PEWS operators describe the allowed workflow of the web service, *i.e.*the *order* in which the operations of the web service will be executed. In this paper we analyze the expressiveness of PEWS programs. This is done by the systematic evaluation of the language. Our evaluation is based on a framework composed by workflow patterns. We also compare PEWS with other interface description languages. This comparison is based on the workflow behavior of the languages.
**Key Words:** workflow patterns, programming languages, web services.
**Category:** C.2.4, D.3.m.

## 1 Introduction

Web services are software systems accessible via Internet. They provide mechanisms for both human and software users to submit data for processing and to receive the results.

The implementation of web services is based on industry standards such as SOAP and WSDL [Curbera et al., 2002]. These two standards are XML languages: SOAP is an application-layer communication protocol, used to code messages. WSDL is a language for the description of web service interfaces. WSDL documents state the name and input/output behavior of the operations of a service, as well as which parameters (messages) are allowed by each operation.

WSDL documents can be registered on a web server to provide a point of access to the service. It is assumed that the programs that implement the service have the input/output behavior specified by the WSDL description.

During a typical web service execution, the web server performs the following sequence of actions:

1. The reception of a SOAP-encoded message (the service call);

2. The unwrapping of the message to obtain the data carried by it;
3. The call to the corresponding operation, providing the received data as parameters;
4. The wrapping of the result into a SOAP-encoded message (the service answer);
5. The sending of the SOAP-encoded message.

WSDL documents deal just with the name, type and parameters of each individual operation of the web service. WSDL documents do not specify the *behavior* of the service interface, *i.e.*the order in which operations of the service can be performed is not defined by the WSDL documents.

Solutions to this problem have been proposed in several forms, ranging from the definition of new implementation languages to formal specifications. On the practical side, we can find the definition of new languages to specify the behavior (or workflow) of web services. Among these languages, we can cite WSCI, XLANG, WSFL, BPML, BPEL4WS, LCWS and PEWS [Arkin et al., 2002, BPML.org, 2002, Andrews et al., 2003, Maciel and Yano, 2005, Ba et al., 2005]. In these proposals, the behavior of the service is defined by using workflow directives, which define the relative order in which individual operations are called.

On the theoretical side, we can survey the use of concurrency models for the specification of web services workflow. These models include process calculi [Salaün et al., 2004], finite state machines [Berardi et al., 2003], Petri nets [Hamadi and Benatallah, 2003] and path expressions [Andler, 1979].

The above mentioned languages and formalisms deal with the definition of *web service compositions*. A compound web service is a new web service formed by the combination of existing ones. The composition of web services is a challenging problem since it is necessary to ensure the correct interaction of software services that are implemented in different software and hardware platforms.

The composition of web services and the definition of (business) process workflow are areas which have many common characteristics [Aalst et al., 2003]. They share, for instance, the necessity of dealing with independent, communicating pieces of software.

In [Wohed et al., 2003] it is proposed the analysis of several web service composition languages. The analysis is based on a framework composed by *workflow patterns*: abstracted forms of common situations found at the organization of business process workflow.

In this paper, we are interested in the use of PEWS, a language for the definition of web services workflow. PEWS is based on Predicate Path Expressions [Andler, 1979]. The language represents a simple but expressive way to describe order and conditional constraints over web service operations.

The main purpose of this paper is to study the use of PEWS to express the workflow patterns presented in [Aalst et al., 2003]. This study will allow us to compare PEWS with other (more popular) languages for web service interfaces.

In the next section we briefly introduce PEWS. Section 3 discuss each of the workflow patterns that compose the framework and their implementation in PEWS. In section 5 we present a comparison of our analysis and the results in [Wohed et al., 2003].

## 2 PEWS

PEWS is a language for the definition of web services. A PEWS program defines the interface to a web service. The operations of the service are implemented by a program written in another programming language[1].

A PEWS program acts as an upper layer for the program implementing the data processing of the web service. A PEWS program is also an extension to the web service interface defined as a WSDL document.

PEWS 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 $a * .(b||c)$ 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 allows a finer control of the access to the object being manipulated. For instance, the predicate path expression $a * .([P]b + [notP]c)$ indicates that either $b$ or $c$ would be executed according to the truth-value of predicate $P$ (the execution of $b$ or of $c$ will be preceded by zero or more executions of $a$).

PEWS extends the proposal in [Andler, 1979] by taking the notion of time into account. Indeed, we follow [Berardi et al., 2003], where the authors stress the importance of taking the notion of time into account when dealing with web services. The notion of 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 the following section we present some examples of PEWS programs. The structure of the programs as well as each construct if the language are explained there.

[1] The only language supported by todays implementation of PEWS is Java.

## 3 Workflow Patterns in PEWS

We have already noticed that the composition of web services and the modeling of business processes deal with similar problems: Both are concerned with the organization of process execution. In this section, we consider the 20 standards for workflow presented in [Aalst et al., 2003], in order to discuss their implementation PEWS.

### 3.1 Basic control patterns

#### 3.1.1 WP1 Sequence:

Several operations will be executed, one at a time, in a predefined, sequential order.
*Implementation of WP1:* This pattern corresponds to the semantics of the (sequential) operator "." of PEWS, exemplified in Figure 1.

The PEWS program in Figure 1 defines a web service containing two operations. The first line of the program refers to a WSDL document (where the service operations and their parameters are defined. The names `op1` and `op2` are *local* names for the operations `opA` and `opB` (which are defined in the referred WSDL document). The expressions `portType/opA` and `portType/opB` serve to identify the path to the operation inside the WSDL document[2].

The last line of the program in Figure 1 defines the behavior of the web service. In this case, the operations `op1` and `op2` are executed sequentially. The sequencing operator is associative.

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace

op1 . op2
```

**Figure 1:** WP1 Sequence.

[2] In the future versions of PEWS, these expressions will be substituted by XPath expressions.

### 3.1.2 WP2 Parallel Split:

Some new control threads are spawn at a point of the program. The new threads can be executed in parallel.
*Implementation of WP2:* The semantics of this pattern corresponds to the parallel constructor in PEWS (Figure 2).

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

(op1 || op2 || op3) . op4
```

**Figure 2:** WP2 Parallel Split and WP3 Synchronization.

The last line of the PEWS program in Figure 2 states that the operations `op1`, `op2` and `op3` will be executed in parallel (or with arbitrary interleaving of their atomic actions, when only one processor is executing the program).

### 3.1.3 WP3 Synchronization:

A point of the program where several threads of control joins into a single one. It is assumed that once a branch finishes its execution it waits until the other branches complete. An example of this situation is "*The car is released to the buyer only after a general quality control is performed on it and the payment is confirmed*".
*Implementation of WP3:* This pattern is supported by PEWS, as shown in Figure 2. The operation `op3` will be executed only after `op1`, `op2` and `op3` have completed.

Notice that only the structured versions of patterns WP2 and WP3 are directly supported by PEWS. The semantics of unstructured workflow for parallel split and synchronization can be obtained by the use of more elaborate constructors in PEWS (or by the repetition of the operations name, if needed). An example of unstructured workflow is shown in Figure 3. The arrows define order restrictions for the operations. For instance, operation $D$ should be executed only after $A$, but it can be done in parallel with $B$. Operation $D$ must be complete before operation $C$ can be performed.

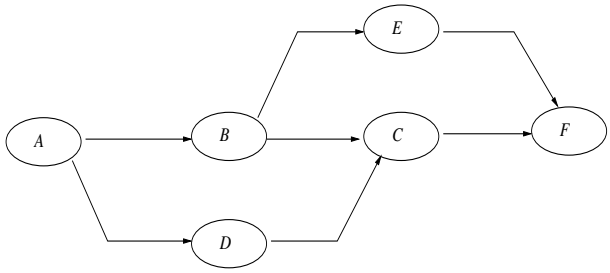



**Figure 3:** Unstructured workflow.

The execution order among operations is imposed (in PEWS) by the use of *predicates on counters*. For each operation in the program, the PEWS run-time system implements three counters. Each of these counters is composed by a pair: natural number (the counter itself) and a timestamp, corresponding to the last time in which the counter was modified. The `val` component of each counter represents the counters value while the `time` component indicates the moment the counter was last modified. Each counter for an operation `O` is described as follows:

`req(O):` The `val` component describes the number of times a caller has attempted to perform the operation `O`. 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.

The control behavior described in Figure 3 can be expressed in PEWS by using the following path expression:

```
A . (( B . [term(D).val > term(C).val] C
         . [term(E).val > term(F).val] F
     )
     || D
     || [term(B).val > term(E).val] E
    )
```

Notice that in this program fragment, the operation `A` is performed first. After `A` completes, a three-branch parallel segment is executed. The first branch represents the main line of the workflow in Figure 3 the operation `B` can be performed in parallel with `D`. The execution of `C`, `F` and `E` will not begin until the predicates preceding these operations become true. For instance, the execution of `E` will be delayed until the number of completions of `B` is greater than the number of times that `E` has been executed.

#### 3.1.4 WP4 Exclusive Choice:

A point in the program where A point in the process where one of several branches is chosen. The decision of which branch is the one which will be executed can be based on data.

#### 3.1.5 WP5 Simple Merge:

A point in the workflow process where two or more alternative branches join (without synchronization). This is the most simple merge of alternatives. It is assumed that none of the alternative branches is ever executed in parallel (that case will be covered by the patterns Multi- Merge and Discriminator).
*Implementation of WP4 and WP5:* Patterns WP4 and WP5 are easily implemented in PEWS using the choice constructor "`|`". This is described by the program in Figure 4.

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

(op1 | op2 | op3) . op4
```

**Figure 4:** WP4 Exclusive Choice and WP5 Simple Merge.

Pattern WP4 corresponds to the choice between operations presented in the last line of Figure 2. When executed, the choice operator will select one branch among those which are *enabled*. One branch is enabled if it has an operation available to be executed and, case it is guarded by a predicate, that predicate is evaluated to true [Ba et al., 2005].

Pattern WP5 is also implemented in Figure 2: The operation `op4` will be performed only after the chosen branch has finished.

#### 3.1.6 WP6 Multi-Choice:

This pattern consists on choosing one or more branches, with base on control data. The chosen branches can be executed in parallel.

#### 3.1.7 WP7 Synchronizing Merge:

This pattern is presented in [Wohed et al., 2003] as "A point in the process where multiple paths converge into one single thread. Some of these paths are active (i.e. they are being executed) and some are not. If only one path is active, the activity after the merge is triggered as soon as this path completes. If more than one path is active, synchronization of all active paths needs to take place before the next activity is triggered. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete."
*Implementation of WP6 and WP7:* Pews programs can contain predicates, which will act as guards on parallel threads. These predicates are defined on the value of counters or other variables of the program. Patterns WP6 and WP7 can be implemented in PEWS using a combination of the parallel and choice constructors. An example of the implementation of these patterns is the program sketch in Figure 5. In this figure, the first sequence operator defines the expected behavior of pattern WP6: The execution of `op2` is conditioned to the truth-value of the predicate `[x < v1]`. The predicate will be evaluated after `op1` is completed. In the case this predicate evaluates to false, the primitive, null operation `nop` is executed (and the same reasoning applies to the branch containing `op3`).

The behavior of WP7 is achieved by the program in Figure 5, since the operation `op4` will be executed at most once, and only if either operation `op2` or `op3` was executed.

#### 3.1.8 WP8 Multi-Merge:

This pattern consists of a point in a process where two or more branches reconverge without synchronization. The activity that follows the merge must be activated for each activated branch. An example of this pattern is: Two different companies are used to emit fly tickets. After all tickets were emitted each ticket must pass a validation process. According to [Wohed et al., 2003], BPEL4WS does not offers direct support to this pattern.
*Implementation of WP8:* The program in Figure 6 uses the PEWS counters to execute `op3` as many times as required.

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

def x = ...an expression...
def y = ...another expression...
def v1 = ...a value...
def v2 = ...another value...

op1.(  ([x < v1]op2 | [x >= v1]nop)
    || ([y > v2]op3 | [y <= v2]nop)
    )
   .
   ( [term(op2).time > term(op1).time
      or term(op3).time > term(op1).time] op4
   | [term(op2).time <= term(op1).time
      and term(op3).time <= term(op1).time] nop
   )
```

**Figure 5:** WP6 Multi-Choice and WP7 Synchronizing Merge.

#### 3.1.9 WP9 Discriminator:

This pattern describes a situation in which several, possibly parallel branches are activated. The pattern is concerned with a point in the workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. All the other (parallel) branches that are still being executed will be ignored (the program will wait until all of them complete). A simple example of this situation is when we send several search commands on the Internet. Only the first response will be considered. This pattern can be generalized to wait for $k$ out of $n$ activated branches. According to [Wohed et al., 2003], BPEL4WS is not directly supported by BPEL4WS.

*Implementation of WP9:* The program in Figure 7 uses the PEWS counters to execute `op3` after `op1` or `op2` complete. Note that it is possible (and simple) to modify this program to express a more general discriminator of the form "$k$ out of $n$".

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

def x = ...an expression...
def y = ...another expression...
def v1 = ...a value...
def v2 = ...another value...

(  ([x < v1]op1 | [x >= v1]nop)
|| ([y > v2]op2 | [y <= v2]nop)
|| ([term(op1).val + term(op2).val
                  > 2*term(op3).val] op3)*
)
```

**Figure 6:** WP8 Multi-Merge.

#### 3.1.10 WP10 Arbitrary Cycles:

This pattern describes a point where a portion of the process (as complex as desired) needs to be performed repeatedly without imposing restrictions on the number, location, and nesting of these points. This pattern is not supported by BPEL4WS nor by PEWS. Both languages provide support for structured cycles only. In the case of PEWS, it provides the iteration operator "*".

#### 3.1.11 WP11 Implicit Termination:

A given activity is terminated when there is nothing left to do, i.e. termination does not require an explicit termination operation.
*Implementation of WP11:* The pattern is directly supported by PEWS. There is no need to explicitly specify a termination activity.

#### 3.1.12 WP12 Multiple Instances without Synchronization:

This pattern corresponds to the spawn of multiple threads from a point of the process. Each thread is a new instance of an activity. All the spawn threads are independent of each other.
*Implementation of WP12:* The pattern is directly supported by PEWS, by using the parallel repetition operator "{...}".

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

def x = ...an expression...
def y = ...another expression...
def v1 = ...a value...
def v2 = ...another value...

(  ([x < v1]op1 | [x >= v1]nop)
|| ([y > v2]op2 | [y <= v2]nop)
|| ([term(op1).val + term(op2).val
          >= 2*term(op3).val + 1] op3)
)
```

**Figure 7:** WP9 Discriminator.

#### 3.1.13 WP13-WP15 Multiple Instances with Synchronization:

These three patterns correspond to points in a workflow where a number of instances of a given activity are initiated. These instances are later synchronized. In WP13 the number of instances to be started/synchronized is known at the time when the program is written. In WP14 the number is known at some stage during run time, but before starting the initiation of the instances. In WP15 the number of instances to be created is not known in advance: new instances are created on demand, until no more instances are required.
*Implementation of WP13-WP15:* Combinations of the parallel repetition operator, with the evaluation of predicates in PEWS can be used to implement these patterns. In the case of WP13, the condition for spawning new threads is statically defined. WP14 is implemented by a condition that depends on a variable, whose value change as the program is executed. WP15 depends on no condition (its implementation is similar to that of WP12).

#### 3.1.14 WP16 Deferred Choice:

This pattern describes a point in a process where some information is used to choose one among several alternative branches. This information is not necessarily available when this point is reached. This pattern differs from the normal

exclusive choice in that the choice can be delayed until the occurrence of some event.
*Implementation of WP16:* The implementation of this pattern in PEWS is similar to that of WP4 (as described in Figure 4). The only difference to that case is that predicates containing the enabling conditions for activities should be used at each branch. According to [Ba et al., 2005], the program will wait until one of the conditions become true, and choose the path expression associated to it.

#### 3.1.15 WP17 Interleaved Parallel Routing

In this pattern, a collection of activities is executed in an arbitrary order. Each activity in the collection is executed once and the order between the activities is decided at run-time. In any case, no two activities in the collection can be active at the same time.
*Implementation of WP17:* This pattern is not directly supported by the present implementation of PEWS. It can be simulated by one big selector, where each branch contains the replication of the operations involved in the pattern. Another way to implement this pattern is to have an external, iterator operation which chooses among the different operations in the collection. This solution depends on an external Java program.

Future versions of PEWS will add support for this pattern, by introducing explicit data operations to the language.

#### 3.1.16 WP18 Milestone:

This pattern describes a situation in which a certain activity can only be enabled if a milestone has been reached which has not yet expired. A milestone is a point in the process where a given activity A has finished and a subsequent activity B has not yet started. For example, this is the case of a student which can cancel a subject at any time after the semester has begun and prior to the first exam. BPEL4WS does not offer support for this pattern.
*Implementation of WP18:* The example in Figure 8 exemplifies the milestone pattern. The execution of operations `op2` and `op3` must depend of an incoming message. Notice that after `op2` is requested for execution, the operation `op3` is not available.

#### 3.1.17 WP19 Cancel Activity and WP20 Cancel Case:

WP19 refers to the termination a running instance of an activity. WP20 leads to the removal of an entire workflow instance. The predefined operations `escape` and `abortOperation` in PEWS give support for these patterns.

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

( op1 . op2
|| ( [term(op1).val > req(op2).val] op3
   | [term(op1).val <= req(op2).val] nop
   )
)
```

**Figure 8:** WP18 Milestone.

## 4 Computational support: The front-end

Eclipse is an extensible programming environment. It permits the construction of new tools for the edition and compilation of programming languages. These tools are called plug-ins and would be written in Java.

In this section we present a plug-in, called PEWS Editor, that provide computational support for PEWS. Pews Editor helps the user to write new compositions, to verify the syntactical and type correctness of PEWS programs and to create XPEWS documents. The components of the plug-in are described as follows.

### 4.1 Editor

Is the main interface with the user, where the compositions will be written . The editor presents facilities like syntax highlight for reserved words, strings and comments (as is shown in item 2 on Figure 9). The editor includes error markers to show the errors found during the parsing (as shown in item 3 of Figure 9).

### 4.2 Context menu

The plug-in includes a XPEWS generation facility for the PEWS program being edited. A right click on the file name or in the editor window shows a context menu with an option called “Generate XPEWS” (item 1 on Figure 9). The execution of this option compiles the program in the editor by verifying the syntax and type semantics of the program. If no errors are detected, a XML document representing a XPEWS version of PEWS composition is generated.

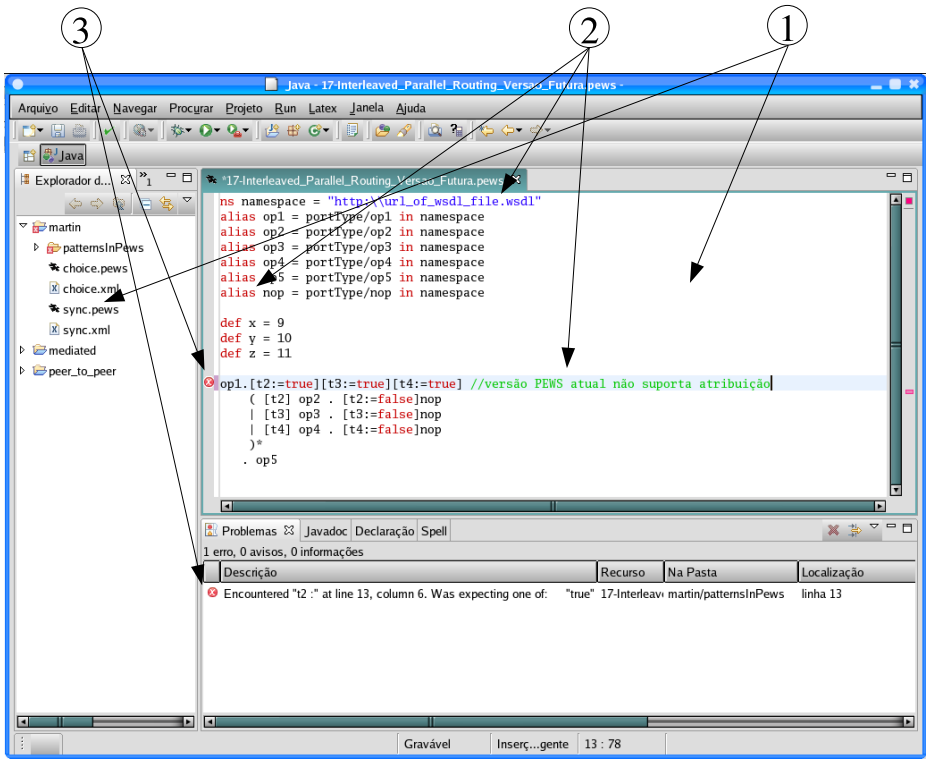



**Figure 9:** PEWS Editor.

### 4.3 Wizard

The wizard help to crate new PEWS programs, building a minimal code (an example document).

The wizard is called by the menu " File → News → Others. . . ". There is an option called "PEWS Composition" that shows the wizard (this wizard can be seen is Figure 10). In this wizard we can define some project parameters like the folder inside the project and the name of file of the PEWS program.

### 4.4 Compiler

The compiler is the main part of the plug-in, being responsible for the lexical, syntactic and semantic analysis, as well as the translation of the PEWS program into XPEWS code.

The compiler was built using JavaCC [Sun Microsystems, 2005]. This tool was used to build the lexical and syntactical analyzers [Aho et al., 1988]. The

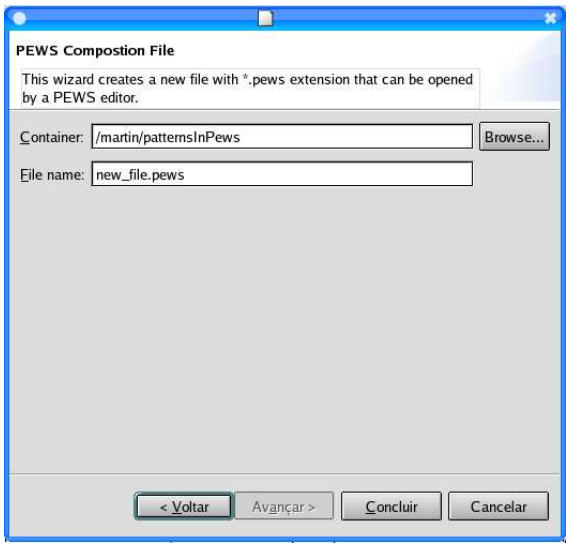



**Figure 10:** Wizard to create PEWS compositions.

JavaCC grammar of the PEWS language is shown in Figure 11. JavaCC generates a top-down parser. Our compiler builds a DOM tree [W3C, 2000] representation of the PEWS program, from which a XPEWS document can be generated.

## 5 Discussion

The preceding section shows the implementation in PEWS of most of the workflow patterns that compose the framework in [Aalst et al., 2003]. We have seen that PEWS gives support to all patterns except for WP10 (Arbitrary Cycles). This was expected, since PEWS supports only structured cycles.

Other web service composition languages have been tested and compared using this pattern framework. The results of the comparison for some of these languages are presented in [van der Aalst, 2003, Wohed et al., 2003]. The six first columns of table 1 are taken from those papers. The last column of the table corresponds to our study. Each row of the table represents one workflow pattern. The symbol "+" indicates that the pattern is supported. The symbol "-" indicates that the pattern is not supported. Partial support is signaled by "+/-".

Notice that PEWS is capable of expressing more patterns than all the other languages. This is an indication of the adequacy of path expressions for the expression of web service composition.

We claim that PEWS programs are, in general, more succinct than equivalent

```
program    ::= ( ns )+ ( alias )+ ( def )* path_expr "EOF"
ns         ::= "ns" namespace "=" file
alias      ::= "alias" opname "=" portType "/" operation "in" namespace
portType   ::= "ident"
operation  ::= "ident"
namespace  ::= ( "ident" )
file       ::= "string_constant"
def        ::= "def" var "=" arith_expr
var        ::= "ident"
pred_expr  ::= pred_term ( pred_expr2 )?
pred_expr2 ::= "or" pred_term
pred_term  ::= pred_factor ( pred_term2 )?
pred_term2 ::= "and" pred_factor
pred_factor::= ( "not" )? bool_expr
bool_expr  ::= "true"
             | "false"
             | arith_expr ( "<" | ">" | "<=" | ">=" | "==" | "! =" ) arith_expr
             | "(" pred_expr ")"
path_expr  ::= parallel ( "||" parallel )*
parallel   ::= choice ( "|" choice )*
choice     ::= sequence ( "." sequence )*
sequence   ::= "{" path_expr "}"
             | unarypath
unarypath  ::= path ( ( "*" | "+" ) )?
path       ::= opname
             | "[" pred_expr "]" path
             | "(" path_expr ")"
             | "abortOperation"
             | "escape"
arith_expr ::= term ( arith_expr2 )?
arith_expr2::= "+" term
             | "−" term
term       ::= unaryexpr ( term2 )?
term2      ::= "*" unaryexpr
             | "/" unaryexpr
unaryexpr  ::= ( "−" )? factor
factor     ::= "now" "(" ")"
             | "act" "(" opname ")" "." "val"
             | "act" "(" opname ")" "." "time"
             | "term" "(" opname ")" "." "val"
             | "term" "(" opname ")" "." "time"
             | "req" "(" opname ")" "." "val"
             | "req" "(" opname ")" "." "time"
             | var
             | "int_constant"
             | "(" arith_expr ")"
opname     ::= ( "ident" )
```

**Figure 11:** PEWS grammar.

| **Pattern** | **Product/Standard** | | | | | |
|---|---|---|---|---|---|---|
| | BPEL | XLANG | WSFL | BPML | WSCI | PEWS |
| Sequence | + | + | + | + | + | + |
| Parallel Split | + | + | + | + | + | + |
| Synchronization | + | + | + | + | + | + |
| Exclusive Choice | + | + | + | + | + | + |
| Simple Merge | + | + | + | + | + | + |
| Multi Choice | + | - | + | - | - | + |
| Synchronizing Merge | + | - | + | - | - | + |
| Multi-Merge | - | - | - | +/- | +/- | + |
| Discriminator | - | - | - | - | - | + |
| Arbitrary Cycles | - | - | - | - | - | - |
| Implicit Termination | + | - | + | + | + | + |
| MI W/O Synchronization | + | + | + | + | + | + |
| MI W/ Priori Design Time Knowledge | + | + | + | + | + | + |
| MI W/ Priori Runtime Knowledge | - | - | - | - | - | + |
| MI W/O Priori Runtime Knowledge | - | - | - | - | - | +/- |
| Deferred Choice | + | + | - | + | + | + |
| Interleaved Parallel Routing | +/- | - | - | - | - | + |
| Milestone | - | - | - | - | - | +/- |
| Cancel Activity | + | + | + | + | + | + |
| Cancel Case | + | + | + | + | + | + |

**Table 1:** Languages and their pattern support [van der Aalst, 2003].

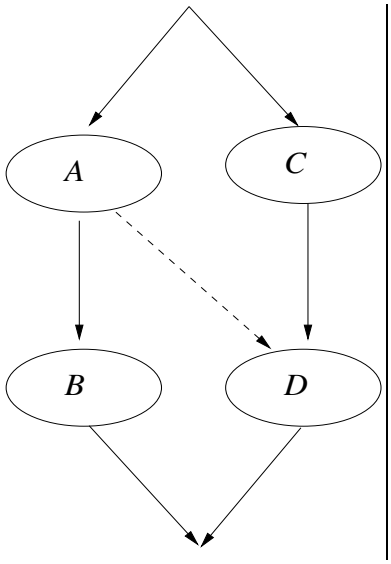
programs in other web service composition languages. This is not only because of the existence of a human-readable version of the language, but because of the use of conditions on the execution of service operations. For example, in Figure 12 is shown a workflow diagram, where the execution of operation `D` will only be initiated after `A` is completed (this is indicated by the dotted arrow). The BPEL4WS code for this workflow is also given in that figure.

A PEWS equivalent to the program in Figure 12 is given by:

```
( A . B ) || ( C . [term(B).val > term(D).val]) D
```

This program segment states that the operation `D` is enabled only after `B` completes its execution.

The implementation of PEWS is phase of alpha testing. The front-end is implemented as a plug-in to the Eclipse programming environment. The front-end has a syntax-aware text editor and type checker, as well as a simple code generator (which produces a XML version of the program, written in XPEWS). The back-end of the language reads the XPEWS program, the WSDL description of the service and the Java implementation of its operations. It produces a new Java program that can be registered to a web server. The new program implements the workflow defined by the PEWS program. The back-end also contains a run-time system to implement the languages operators and predefined operations.




```
<flow name="F">
 <links>
  <link name="L"/>
 </links>
  <sequence name="Sa">
   A
    <source linkName="L"/>
   B
  </sequence>
  <sequence name="Sb">
   C
   D
    <target linkName="L"/>
  </sequence>
</flow>
```

**Figure 12:** Unstructured workflow [Wohed et al., 2003].

## References

[Aalst et al., 2003] Aalst, W. M. P. V. D., Hofstede, A. H. M. T., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51.

[Aho et al., 1988] Aho, A. V., Sethi, R., and Ullman, J. D. (1988). *Compilers: principles, techniques, and tools*. Addison-Wesley.

[Andler, 1979] Andler, S. (1979). Predicate path expressions. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 226–236. ACM Press.

[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 Weerawarana, S. (2003). Specification: Business process execution language for web services, version 1.1. 2003. Available at http://www-106.ibm.com/developerworks/library/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/.

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

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

[BPML.org, 2002] BPML.org (2002). Business process modeling language. www.bpmi.org.

[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*.

[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.

[Maciel and Yano, 2005] Maciel, L. A. H. and Yano, E. T. (2005). Uma linguagem de workflow para composio de web services lcws. In *19th Brazilian Symposium on Software Engineering*. In portuguese.

[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*.

[Sun Microsystems, 2005] Sun Microsystems, I. (2005). Java compiler compiler$^{TM}$.

[van der Aalst, 2003] van der Aalst, W. (2003). Don't go with the flow: Web services composition standards exposed.

[W3C, 2000] W3C (2000). Document object model (dom) level 2 core specification.

[Wohed et al., 2003] Wohed, P., van der Aalst, W. M., Dumas, M., and ter Hofstede, A. H. (2003). Analysis of web services composition languages: The case of bpel4ws.