Workflow Patterns as Web Service Compositions: The case of PEWS

Marcelo Guerra[1], Regina Motz[1], Martin A. Musicante[2], and Alberto Pardo[1]

[1] Instituto de Computacion - Universidad de la Republica, Montevideo, Uruguay
[2] DIMAp - Universidade Federal do Rio Grande do Norte, Natal, Brazil

Abstract. In the past years, Web Services (WS) have become the standard for exposing services as application programming interfaces to be consumed from anywhere in the world. Since many operations require the collaboration between two or more WS, the need to have languages to express Web Service Compositions has emerged. Web Service Composition presents a problem very similar to Business Process Management (BPM), both disciplines aim to express complex combinations of operations to achieve broader goals. In BPM these combinations are generally known as Workflows. The current industry standard for representing Web Service Workflows is BPEL, a language developed by Microsoft, IBM and others. PEWS is another language proposed to describe Web Service Compositions. Simplicity and neatness of descriptions are two relevant features of this language. This work proposes some extensions to PEWS in order to support the most common workflow patterns. The extended version of PEWS is able to fully support 36 out of the 43 Workflow Control Patterns as defined in the literature. This new version creates the foundation for further studies on the language, especially future extensions to capture other patterns and features (such as data manipulation, error handling, etc.) as well as the addition of semantic information to compositions.

1 Introduction

Web services are software systems accessible via Internet. In a typical web service development setting, we have: *(i)* a description of the operations of the service and the data types they process; *(ii)* a specification of the service behavior (possibly written in a natural language) and *(iii)* a set of programs to implement each operation. The description of the service is given as a WSDL (Web Service Description Language) document [1]. It describes the names and interfaces (types of the arguments) of the operations of the service. WSDL addresses only static interface specifications and it does not describe the *observable behavior* of the web service. Moreover, WSDL is not equipped to describe compositions. Each operation is, normally, implemented as a method or procedure and its execution is supported by a web server.

The need for better-defined behavioral interfaces for software components and web services has motivated the definition of new languages and description techniques for these interfaces. PEWS (Path Expressions for Web Services) [2, 3] is an interface description language to define composed web services.

The composition of web services and the definition of (business) process workflows are areas which have many common characteristics [4]. They share, for instance, the necessity of dealing with independent, communicating pieces of software. In [5] 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 workflows.

In [6], new workflow patterns were added to those presented in [5]. The new patterns refine and specialize the patterns in [5].

The main goal of this work is to study the use of PEWS to express the workflow patterns presented in [6]. This study will allow us to compare PEWS with other (more popular) languages for web service interfaces. In particular, with BPEL [7], which is the *de-facto* standard for web service composition.

In the next section we briefly introduce PEWS. Section 3 discusses each of the workflow patterns that compose the framework presented in [6] and their implementation in PEWS. In Section 4 we present some extension to PEWS that are convenient for the practical use of the language as well as for the implementation of the new workflow patterns. In Section 5 we use the extensions to PEWS for the implementation of more workflow patterns. Section 6 presents a comparison of PEWS and BPEL in terms of the ability of both languages to express workflow patterns, as well as our analysis and this comparison.

2 PEWS

PEWS is a language for the definition of web services. The language is devised to help in the design of web services by specifying their behavior, *i.e*,. the relative order in which the operations of the service can be executed.

PEWS brings predicate path expressions to the context of web services. Predicate path-expressions [8] 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 [8], 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+[not P]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*).

The structure of a PEWS program is defined as:

```
P::=(var X = Exp)*(task T = S)* S
S::= O | T | S.S | S||S | S + S | if (Bexp) S |
while (Bexp) S

Exp::= Aexp | Bexp
```

According to the grammar above, a PEWS program *P* is composed by a sequence of:

- ***Macro definitions:*** An identifier *X* is bound to an expression (*Exp*) that may be arithmetic (*Aexp*) or boolean (*Bexp*). The value of *X* will be computed each time the identifier is used during the run of the program. The expression may contain system identifiers that may change their value during execution. (This feature will be explained later on in this section.)
- ***Task definitions:*** An identifier *T* is bound to a workflow expression *S*. Task definitions may be recursive. For example, we can define the following tasks: **task** `T`$_1$`=A.(B||C)` and **task** `T`$_2$`=T`$_1$`.T`$_2$. Notice that the task T_2 is equivalent to the task `while (true) (A.(B||C))`.
- ***A workflow expression***: This expression defines the main task of the program. It represents the allowable order in which the operations of the service may be performed.

The grammar for *S* defines that a workflow expression can be a WSDL defined operation (*O*) or a task name (*T*); the sequential (`.`) or parallel (`||`) composition of workflow expressions; the exclusive choice between workflows (`+`); the conditional execution of a workflow (`if (Bexp) S`) and the (conditional) repetition of a workflow expression (`while (Bexp) S`). *Bexp* are PEWS predicates (boolean expressions that can contain PEWS counters and identifiers). PEWS also includes some predefined, primitive tasks, identified with the names `break` (used for the termination of loops), `exit` (successful, explicit termination of the current task) and `nop` (the no-operation).

The language associates *counters* to each operation and task: Given a task/operation identifier *Y* in the program, the PEWS run-time system associates three counters to it. They correspond, respectively, to the number of times *Y* has been called (`req(Y)`), started (`act(Y)`) or finished (`term(Y)`). Each of these counters is composed by a pair: a natural number (the counter itself, denoted by `val`) and a timestamp, corresponding to the last time in which the counter was modified (denoted by `time`). For instance, the expression `term(SellItem).time - act(SellItem).time` describes the time interval between the activation and termination of the opeation `SellItem`.

The language defined in this section will be referred to as *core* PEWS. This language is capable of describing an expressive number of workflow patterns (as it will be seen in the next section). However, the direct implementation of some other workflow patterns is problematic in core PEWS. The language will be extended with new features in section 4, in order to obtain a broader direct representation of the workflow patterns defined in [6].

3 Workflow Patterns

In this section we will focus on the control flow patterns supported by core PEWS. Control flow patterns correspond to a set of forty-three control structures, classified into eight collections, which usually arise during business process modeling [6].

Basic Control Flow Patterns: the Workflow Management Coalition defined a set of five patterns that represent the most basic combination of tasks. The list of patterns includes Sequence, Parallel Split, Synchronization, Exclusive Choice and Simple Merge. Given processes *A*, *B* and *C*, and a boolean expression *Cond*, these patterns can be defined in PEWS as:

Pattern	**Representation in PEWS**
Sequence	`A.B.C`
Parallel Split	`A.(B||C)`
Synchronization	`(A||B).C`
Exclusive Choice	`A.(if (Cond) B + if (Cond) C)`
Simple Merge	`(if (Cond) A + if (¬Cond) B).C`

Advanced Branching and Synchronization Patterns: the basic control flow patterns above turned out to be effective for business process automation and have widespread support, however, further requirements for workflows arise in practice, opening the possibility for extensions. In order to address this problem, a set of advanced branching and synchronization patterns were defined in [6].

Multiple Instance Patterns: in these patterns, the activation of several instances of the same task is needed. Each instance runs independently from their calling thread, which will not wait for their completion. Excepting for the Cancelling Partial Join for Multiple Instances, all of these patterns are representable in PEWS with the extensions that will be presented in section 4. We will give a representation, as well as an explanation, for these patterns in section 5.

State-based Patterns: state-based patterns represent situations where there is a need to support state. State represents the data and metadata related to the current execution, including the status of the various tasks being executed, variables and other data elements.

Deferred Choice: deferred choice patterns describe an external choice of tasks. This pattern can be described in PEWS using the choice operator, as in (A + B). One of the branches of the constructor will be chosen, in accordance with their availability.

Cancellation and Force Completion Patterns: some patterns describe situations where task cancellation is required. The simplest situation involving cancellation is captured by the Cancel Task pattern, where just a task is aborted. However, more advanced situations can arise. This group of patterns includes: Cancel Task, Cancel Case, Cancel Region, Cancel Multiple Instance Activity, Complete Multiple Instance Activity.

Iteration Patterns Iteration Patterns: define the repeated execution of tasks. The patterns in this group are analogous to loop and recursion constructs that are common in programming languages. These patterns include: Structured Loop, Recursion, and Arbitrary Cycles.

Termination Patterns Termination Patterns describe the ability to determine when a process is considered to be completed. Implicit and Explicit terminations are the two relevant patterns in this group: Implicit Termination and Explicit Termination.

Both patterns are (trivially) supported by core PEWS. Implicit termination is represented by the conclusion of a task, after its successful performance. Explicit termination of a task is represented in core PEWS by the exit primitive.

Trigger Patterns: the Trigger patterns define the ability of tasks to be started by external signals. There are two possible patterns, whose difference is on the persistence of the signal: Transient Trigger and Persistent Trigger.

4 Extending PEWS

Some of the patterns listed in the preceding section cannot be directly implemented in core PEWS. In this this section we will present a set of extensions that enable PEWS to express most of those patterns. These extensions add to the language new primitives for task cancellation, task instance creation and semaphores.

Task Instances There are situations where multiple instances of the same task need to be executed. These include:

- An activity is able to initiate multiple instances of itself.
- A given activity is initiated multiple times after receiving different triggers.
- Two or more activities share the same definition.

To support task instance creation, the $...$ operator is added to PEWS. This operator is defined as follows: Each time the control flow reaches the task A, a new instance of A will be spawned and the control flow continues without waiting for the termination of the spawned instance.

Task Cancellation The cancel operation is expressed by using the "!" operator, therefore A! means cancelling the execution of A, where A is a task identifier. For instance, the following expression first executes A and then initiates both tasks *B* and *C*. The first of these latter two tasks that finishes its execution will cancel the execution of the other one.

```
A.(B.C!||C.B!)
```

Notice that owing to the fact that the cancel operation can be used in combination with the task definition primitive, it is possible to cancel a composite task. In that case, all the tasks contained in the composite task are cancelled.

Semaphores Some situations require the synchronization of different threads of execution. These situations include scenarios that require accessing to critical regions or triggering a task (when a certain condition is verified or a point in the workflow is reached).

In order to support these cases, we harness the PEWS language with a library to implement semaphore primitives. Semaphore operations in PEWS are defined as:

`Sem<name>`: Defines a new semaphore of name `<name>`.

`set(<name>,<initial count>)`: Sets the value of the semaphore `<name>` with value `<initial count>`.

`wait(<name>)`: Decrements by one the counter of the semaphore, if its value is positive. If the value of the semaphore is zero, then the current thread suspends its execution until the value of the semaphore can be decremented.

`signal(<name>)`: Increments by one the value of the semaphore.

`free(<name>)`: Disposes the semaphore of name `<name>`.

5 Representing more Workflow Patterns in PEWS

In this section, we use the new constructors presented in section 4 to propose implementations in PEWS for most of the patterns that were not treated in section 3. The extensions not only improve the usability of PEWS, but also enable the direct support of most of the common workflow patterns. The rest of this section deals with the representation of the patterns in PEWS.

Structured Partial Join: this workflow pattern describes a situation in which a task is performed only after a choice of m out of n other tasks are completed. For instance, executing a task after two out of the three other tasks have finished. The PEWS representation of this situation is given as follows:

```
sem S . set(S, 0).A
.(B.signal(S)||C.signal(S)||D.signal(S)||wait(S).
wait(S).E)
.free(S)
```

In this example, the task A is performed first. After A's completion, the tasks B, C and D are initiated. The task E is performed only after two out of the three tasks B, C and D completed.

Cancelling Partial Join: this workflow pattern is similar to the previous one, in the sense that a task will be performed only after m out of n other tasks are completed. The only difference with the previous pattern is that the n – m unfinished tasks are cancelled. The case of this pattern where we should choose 2 out of 3 tasks can be described by the following PEWS expression:

```
sem S . set(S,0)
.A.(B.signal(S)||C.signal(S)||D.signal(S)
      ||wait(S).wait(S).(if(term(B).val+term(C).val=2)D!
                     +if(term(B).val+term(D).val=2)C!
                     +if(term(C).val+term(D).val=2)B!).E)
.free(S)
```

Blocking Partial Join: this pattern is similar to the *Structured Partial Join* pattern, where subsequent executions of the partial join require that previous executions of it are already completed. The PEWS representation of this pattern uses a globally defined semaphore G, initialized in 1. The case of this pattern where we should choose 2 out of 3 tasks can be described by the following PEWS expression:

```
wait(G)
.sem S.set(S, 0)
.A.(B.signal(S)||C.signal(S)||D.signal(S)||wait(S).
wait(S).E)
.free(S)
.signal(G)
```

Multiple Instances without Synchronization: this pattern is exactly described by the task instances primitive of PEWS.

Multiple Instances with a Priori Design-Time Knowledge: this pattern models a situation in which a statically known number (*n*) of instances of a task can be created. These instances are independent of each other and run concurrently. This pattern is implemented in PEWS as: `{$A$}`^{n}.

Multiple Instances with a Priori Run-Time Knowledge: this pattern describes a situation in which a number of task instances needs to be created. The number of instances is known before the instance creation begins. Given that the task *A* defines n, we can have this pattern described as: `A.{$B$}`^{n} .

Multiple Instances without a Priori Run-Time Knowledge: this pattern describes a situation in which the spawn of new instances is controlled by a condition in the program. This pattern is partially representable in PEWS, due to the nature of conditions in the language. In fact, PEWS conditions only define expressions on counters and constants. Data processed by operations (like input/output data) do not occur in PEWS programs.

The (partial) representation of this pattern is given by the task X, as follows:

```
task X = ((if(Cond)$A$+if(¬Cond)exit)||X)
```

Static Partial Join for Multiple Instances: this pattern corresponds to the execution of a task B after the completion of m instances (out of a total of n) of a task A. The PEWS representation of this pattern is:

`sem S.set(S,0).{$A.signal(S)}`^{n}`.{wait(S)}`^{m}`.B.free(S)`

Interleaved Parallel Routing: this pattern describes a situation in which a set of tasks is executed in a predefined (partial) order. No two tasks are to be executed at the same time. The partial order of tasks can be represented in PEWS by the combination of the sequential and parallel composition. The non-concomitant execution of tasks can be expressed in PEWS using semaphores. As an example of this pattern in PEWS, consider the case where a task B should be performed after A, while there is no defined order between the execution of these two tasks and another task C:

```
sem S . set(S,0).
(wait(S).A.signal(S).wait(S).B.signal(S))||
(wait(S).C.signal(S))
.free(S)
```

Milestone: this pattern describes a situation where a task is enabled only when the execution of another branch of the workflow is at a specific point (called a milestone). For instance, suppose that a task E, inside a loop, is enabled only after a task B has finished and before another task C started (B and C are in a sequence). This can be described by the following PEWS code:

```
var M = term(B).val = act(C).val + 1
task W = D.(if(M)(E.W+F.G)+if(¬M)F.G)
...
(A.B.C)||W
```

Critical Section: a critical section pattern describes a situation in which two or more parts of a composite task must be performed one at a time. The use of semaphores in PEWS implements this pattern. For instance, in the following PEWS expression,

```
sem S . set(S,0)
.A.(wait(S).(B.C).signal(S))||
(wait(S).(D||E).signal(S))
.free(S)
```

the sequential composition of B and C defines one critical section of the task and the parallel composition of D and E defines another critical section.

Interleaved Routing: this pattern describes a situation in which a pool of tasks is executed in mutual exclusion. This pattern is easily described using the semaphore primitive of PEWS.

```
        sem S . set(S,0)
        .((wait(S).A.signal(S))||
        (wait(S).B.signal(S))||(wait(S).C.signal(S)))
        .free(S)
```

Cancel Task: the cancellation of a task is a pattern that allows a part of the program to be cancelled by an authoritative agent. The preemption primitive of PEWS can be used for such a goal, as: (A : B!) k B . Notice that the task B can be terminated after A finishes.

Cancel Case: this pattern describes a situation in which a composite task must be preempted. Its representation in PEWS consists in defining a composite task by using the task directive, in order to identify it with a name. This task can, then, be cancelled in the same manner as in the previous pattern.

```
        task C = ...
        A.((B.C!)||C)
```

Transient Trigger: this pattern models the case in which the execution of a task is conditioned to the activation of a trigger. We will model the trigger using a semaphore. Notice that the activation of the trigger is transient, meaning that if there is no task ready to be performed once the trigger is enabled, then the activation has no efect. If there is a task waiting for the trigger's activation, then this task will be executed. This pattern can be described in PEWS as:

```
        sem(S).set(S,0).
        (while(true)(signal(S).set(S,0))||(A.wait(S).B))
```

Persistent Trigger: this pattern describes a situation that is similar to the previous one, except for the fact that the trigger is persistent (meaning that once it is activated, this condition remains during execution). This pattern can be described in PEWS as:

```
        sem(S).set(S,0).(signal(S)||(A.wait(S).B))
```

Local Synchronizing Merge: this pattern defines that one of the branches of a parallel task can take an independent path, exiting the constructor by creating a new thread. This is described in PEWS by using the task instance primitive, where the independent path is defined as the spawn of a task instance (D).

```
        task D = ...
        if(Cond1)A||if(Cond2)(B.(C+$D$))
```

General Synchronizing Merge: this pattern generalizes the previous one, adding a loop before the (local) synchronizing merge. This situation can be described in PEWS as follows:

```
        task D = ...
        task X = B.(C+$D$+X)
        if (Cond1) A || if (Cond2) X
```

Cancel Region: this pattern describes the ability of a task to cancel a set of other, possibly unrelated tasks. This capability is useful when alternative execution paths may be chosen, as a form of handling exceptional situations. In the following PEWS program, the region formed by tasks B and D will be cancelled whenever a trigger is activated.

```
        ((A.B)||(C.D))||...if(Trigger)(B!||D!)...
```

Cancel Multiple Instance Activity: this pattern describes a situation where multiple instances of a task must be terminated. The cancellation affects all the instances that are currently being executed as well as all future instances. The PEWS implementation of this situation can be described as follows.

```
        (...$A$...$A$...)||...if(Trigger) A!...
```

6 Concluding Remarks

Sections 3 and 5 show the implementation in PEWS of most of the workflow patterns that compose the framework in [6]. We have seen that PEWS gives support to most patterns, with the exception of those which are not structured. This was expected, since PEWS itself is a structured language.

We have compared the use of PEWS and BPEL for the expression of the workflow patterns presented in [6]. The result of the comparison appears in Table 1. In that table, the columns marked BPEL and PEWS correspond, respectively, to the ability of these languages to directly represent the patterns of the first column. Each row of the table represents a workflow pattern. The symbol "+" indicates that the pattern is supported, where the symbol "-" indicates the contrary. Partial support is signaled by "+/-".

References

1. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1. Availabre at http://www.w3.org/TR/wsdl (2001)
2. Ba, C., Carrero, M., Halfeld Ferrari, M., Musicante, M.: PEWS: A new language for building web service interfaces. Journal of Universal Computer Science 11 (2005) 1215-1233 http://www.jucs.org/jucs 11 7/pews a new language.
3. Musicante, M.A., Potrich, E., Carrero, M.A.: A programming environment for web services. In Wainwright, R.L., Haddad, H., eds.: SAC, ACM (2008) 2363-2367
4. Aalst, W.M.P.V.D., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14 (2003) 5-51
5. Wohed, P., van der Aalst, W.M., Dumas, M., ter Hofstede, A.H.: Analysis of web services composition languages: The case of BPEL4WS. In: Proceedings of the 22nd International Conference on Conceptual Modeling (ER), Chicago IL, USA, "citeseer.ist.psu.edu/659670.html" (2003)
6. N. Russell, A.H.M. ter Hofstede, W.v.d.A., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical Report BPM Center Report BPM-06-22, BPMcenter.org (2006)
7. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weeranwarana, S.: Bussiness process execution language for web services. Available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ (2003)
8. Andler, S.: Predicate path expressions. In: Sixth Annual ACM Symposium on Principles of Programming Languages (6th POPL'79). (1979) 226-236
9. Berardi, D., de Rosa, F., de Santis, L., Mecella, M.: Finite state automata as conceptual model for e-services. In: Integrated Design and Process Technology (IDPT). (2003)

Table 1. Workflow Patterns in BPEL and PEWS.

Group	Pattern	BPEL	PEWS
Basic	Sequence	+	+
	Parallel Split	+	+
	Synchronization	+	+
	Exclusive Choice	+	+
	Simple Merge	+	+
Advanced Branching and Sync	Multi-Choice	+	+
	Structured Synchronizing Merge	+	+
	Multi-Merge	-	+
	Structured Discriminator	-	+
	Blocking Discriminator	-	+
	Cancelling Discriminator	-	+
	Structured Partial Join	-	+
	Blocking Partial Join	-	+
	Cancelling Partial Join	-	+
	Generalized AND-Join	-	-
	Local Synchronizing Merge	+	+
	General Synchronizing Merge	-	+
	Thread Merge	+/-	+
	Thread Split	+/-	+
Mult. Instances	Multiple Instances without Synchronization	+	+
	Multiple Instances with a Priori Design-Time Knowledge	-	+
	Multiple Instances with a Priori Run-Time Knowledge	-	+
	Multiple Instances without a Priori Run-Time Knowledge	-	+/-
	Static Partial Join for Multiple Instances	-	+
	Cancelling Partial Join for Multiple Instances	-	-
	Dynamic Partial Join for Multiple Instances	-	-
State-based	Deferred Choice	+	+
	Interleaved Parallel Routing	+/-	+
	Milestone	-	+
	Critical Section	+	+
	Interleaved Routing	+	+
Cancellatio n	Cancel Task	+	+
	Cancel Case	+	+
	Cancel Region	+/-	+
	Cancel Multiple Instance Activity	-	+
	Complete Multiple Instance Activity	-	-
Iter.	Arbitrary Cycles	-	-
	Structured Loop	+	+
	Recursion	-	+
Te rm .	Implicit Termination	+	+
	Explicit Termination	-	+
Tri .	Transient Trigger	-	+
	Persistent Trigger	+	+