2010 2nd International Conference on Information and Multimedia Technology (ICIMT 2010)
IPCSIT vol. 42 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V42.4

Using OWL-S for Formalizing Workflow Patterns

Kui Yu[1], Hai Xu[1], Cuiqiong Jiang[2], Gang Xue[3], Shaowen Yao[3]

[1]School of Information Science and Engineering, Yunnan University, Kunming, China
[2]Student Affairs Department, Simao Teacher's College, Pu'er, China
[3]National Pilot School of Software, Yunnan University, Kunming, China
y_k_2000@163.com, cxtc2004@163.com, jcq_rose@163.com, mess@ynu.edu.cn, yaosw@ynu.edu.cn

Abstract—Workflow Patterns contain basic features of business process. The modeling languages and methods decide the execution of these patterns. Workflow Patterns include the original 20 patterns and 23 extended patterns. OWL-S is an ontology, within the OWL-based framework of the Semantic Web, for describing Semantic Web Services. The process model of the OWL-S consists of nine control constructs. The implementations of workflow patterns with the control constructs of the OWL-S process model are presented in this paper.

Keywords-Workflow Patterns; business process; OWL-S; process model; control constructs;

1. Introduction

Workflow Patterns describe activities and their execution ordering through different constructors, which permit flow of execution control. Reference [1] described the 20 workflow patterns frequently used in the business process. Reference [2] offered a modified version of the original 20 patterns, adding 23 extended workflow patterns.

OWL-S is short for Ontology Web Language for Services, the process model of which defines a set of processes and their execution order, and consists of nine control constructs. This paper will use these control constructors to formalize the Workflow Patterns.

The paper is organized as follows. Related work is discussed in section 2. Section 3 included brief introduction to Workflow Patterns and OWL-S. Workflow Patterns are represented using OWL-S in section 4. Finally, conclusion and future work are addressed in section 5.

2. Related Work

Reference [3] and [4] offered formal description of Workflow Patterns using Pi-Calculus. Another approach of giving a detailed representation of the workflow patterns has been made with YAWL[5]. Related research on workflow patterns is not limited to the workflow patterns itself, but also includes the workflow resource patterns, workflow data patterns.

There are many languages that can be used to describe the interaction protocols between web services, such as BPEL4WS [6], BPSS [7], WSCI [8], OWL-S [9]. But because these languages don't clearly define formal execution semantics, the combination of the implementation of web services is made more difficult.

3. Foundational Technologies

3.1 Workflow Patterns

Workflow Patterns consist of 43 patterns, including original 20 patterns and extended 23 patterns. There are eight categories of Workflow Patterns[1][2]: Basic Control Flow Patterns, Advanced Branching and Synchronization Patterns, Multiple Instance Patterns, State-based Patterns, Cancellation and Force Completion Patterns, Iteration Patterns, Termination Patterns, Trigger Patterns. This paper is mainly used to formalize Workflow Patterns using OWL-S.

3.2 OWL-S

OWL-S is an ontology, within the OWL-based framework of the Semantic Web, for describing Semantic Web Services. The OWL-S ontology has three main parts: the service profile, the process model and the grounding. The process model includes nine control constructs. They are as follows: Sequence, Split, Split+Join, Choice, Any-Order, If-Then-Else, Iterate, Repeat-While, and Repeat-Until. Iterate is ignored because it is an “abstract” class, in the sense that it’s not detailed enough to be instantiated in a process model [9].The specific meaning of control constructs of process model refers to reference [9]. OWL-S didn't consider the process execution, so the patterns relevant to the execution can’t be formalized using OWL-S.

4. Patterns Representation

Workflow Patterns capture the distinct modeling constructs that can be identified during process modeling[1]. A part of the Workflow Patterns will be formalized using OWL-S in the following sections.

4.1 Basic Control Flow Patterns

This section covers Basic Control Flow Patterns. There are five patterns, which are formalized below.

Sequence. Fig.1 illustrates the behavior of this pattern. The Sequence can directly formalize this pattern.

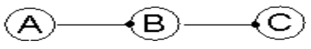

Figure 1. Sequence

```
<Sequence>
   <Process>A</Process>
   <Process>B</Process>
   <Process>C</Process>
</Sequence>
```

Parallel Split. Fig.2 can be used to represent this pattern. This pattern can be directly formalized by the Split.

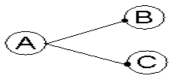

Figure 2. Parallel Split

```
<Sequence>
   <Process>A</Process>
   <Split>
      <Process>B</Process>
      <Process>C</Process>
   </Split>
</Sequence>
```

Synchronization. Fig.3 illustrates the behavior of this pattern. The Split-Join can directly formalize this pattern.

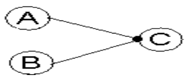

Figure 3. Synchronization

```
<Sequence>
   <Split-Join>
      <Process>A</Process>
```

```
    <Process>B</Process>
  </Split-Join>
  <Process>C</Process>
</Sequence>
```

Exclusive Choice. Fig.2 can also be used to represent this pattern. Either process B or process C is executed after the completion of process A according to condition *f1*.

```
<Sequence>
  <Process>A</Process>
  <If-then-else>
    <If-Condition> f1</If-Condition>
    <then><Process>B</Process></then>
    <else><Process>C</Process></else>
  </If-then-else>
</Sequence>
```

Simple Merge. Fig.3 can also illustrate the behavior of this pattern. The simple merge of two control flows from either processes A or B into C is achieved by condition *f1*. Per definition of this pattern, A and B will never be executed in parallel, so process C only needs to wait on one completion of process A or B.

```
<Sequence>
  <If-then-else>
    <If-Condition> f1</If-Condition>
    <then>
       <Process>A</Process>
    </then>
    <else>
       <Process>B</Process>
    </else>
  </If-then-else>
  <Process>C</Process>
</Sequence>
```

4.2 Advanced Branching and Synchronization Patterns

This section covers advanced branching and synchronization patterns. There are fourteen patterns. Because OWL-S didn't consider the process execution, the following patterns can’t be formalized using OWL-S: Structured Discriminator, Blocking Discriminator, Cancelling Discriminator, Blocking Partial Join, Cancelling Partial Join, Generalized AND-Join.

Multi-Choice. Fig.2 illustrates the behavior of this pattern, too. The choice between processes B or C or B and C after A is modeled by A having three possibilities of execution. Either process A triggers process B or C or both B and C according to condition *f1* and *f2*.

```
<Sequence>
  <Process>A</Process>
  <Split>
    <If-then-else>
      <If-Condition> f1</If-Condition>
      <then><Process>B</Process></then>
    </If-then-else>
    <If-then-else>
      <If-Condition> f2</If-Condition>
      <then><Process>C</Process></then>
    </If-then-else>
  </Split>
</Sequence>
```

Structured Synchronizing Merge. Fig.3 can also be used to illustrate the behavior of this pattern. The triggers for activating a process C can either come from process A or B as well as from process A and B under the condition *f1* and *f2*. If process A and B are executed in parallel, process C has to wait on the completion of A and B, otherwise only for the completion of process A or B.

```
<Sequence>
  <Split-Join>
    <If-then-else>
      <If-Condition> f1</If-Condition>
      <then><Process>A</Process></then>
    </If-then-else>
    <If-then-else>
      <If-Condition> f2</If-Condition>
      <then><Process>B</Process></then>
    </If-then-else>
  </Split-Join>
  <Process>C</Process>
</Sequence>
```

Multi-Merge. Fig.3 can be used to represent this pattern, too. Process C can be triggered arbitrary times by incoming triggers from process A or B. Each time process C gets triggered on the basis of the condition *f1* and *f2*, process C is repeatedly executed.

```
<Sequence>
  <Split-Join>
    <If-then-else>
      <If-Condition> f1</If-Condition>
      <then>
        <Sequence>
          <Process>A</Process><Process>C</Process>
        </Sequence>
      </then>
    </If-then-else>
    <If-then-else>
      <If-Condition> f2</If-Condition>
      <then>
        <Sequence>
          <Process>B</Process><Process>C</Process>
        </Sequence>
      </then>
    </If-then-else>
  </Split-Join>
</Sequence>
```

Structured Partial Join. Fig.4 can illustrate the behavior of this pattern. Process D can be triggered by incoming trigger from the completion of the two of process A, B or C. The Choice will choose a process from process A, B or C according to the condition *f1*, *f2*, or *f3*.

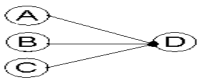

Figure 4. Structured Partial Join.

```
<Sequence>
  <Split-Join><Choice><If-then-else>
    <If-condition>f1</If-condition>
      <then><Process>A</Process></then>
   </If-then-else><If-then-else>
   <If-condition>f2</If-condition>
```

```
        <then><Process>B</Process></then>
      </If-then-else><If-then-else>
      <If-condition>f3</If-condition>
        <then><Process>C</Process></then>
      </If-then-else></Choice><Choice>
      <If-then-else><If-condition>f1</If-condition>
        <then><Process>A</Process></then>
      </If-then-else><If-then-else>
      <If-condition>f2</If-condition>
        <then><Process>B</Process></then>
      </If-then-else><If-then-else>
      <If-condition>f3</If-condition>
        <then><Process>C</Process></then>
     </If-then-else></Choice></Split-Join>
   <Process>D</Process>
</Sequence>
```

Local Synchronizing Merge. Fig.5 can be used to represent this pattern. The difference between this pattern and Structured Synchronizing Merge pattern is that this pattern introduces Deferred Choice pattern. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. Here, the condition *f1* is used to represent information locally available.

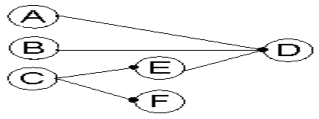

Figure 5. Local Synchronizing Merge.

```
<Sequence>
   <Split-Join><Process>A</Process><Process>B</Process>
      <Sequence><Process>C</Process><Split>
         <If-then-else><If-condition>f1</If-condition>
            <then><Process>E</Process></then>
            <Else><Process>F</Process></Else>
            </If-then-else></Split>
      </Sequence> </Split-Join>
   <Process>D</Process>
</Sequence>
```

General Synchronizing Merge. Fig.6 illustrates the behavior of this pattern. This pattern is much the same as Local Synchronizing Merge pattern, with the difference that process C is included in Deferred Choice pattern. The condition *f2* and *f3* is used to represent information locally available.

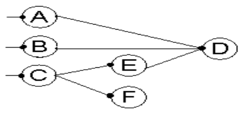

Figure 6. General Synchronizing Merge

```
<Sequence>
   <Split-Join><Process>A</Process> //set f1 to be false
      <Process>B</Process> <Sequence>
      <Process>C</Process> <Split>
      <Repeat-while><whileCondition>f1</whileCondition >
        <whileProcess><If-then-else>
         <If-condition>f2</If-condition>
         <then><Process>C</Process></then>
         <Else><If-then-else><If-condition>f3</If-condition>
         <then><Process>E</Process>//set f1 to be false
         </then><Else> <Process>F</Process>//set f1 to be false
        </Else></If-then-else></Else></If-then-else>
        </whileProcess></Repeat-while>
       </Split></Sequence></Split-Join>
  <Process>D</Process>
</Sequence>
```

Thread Merge. Fig.7 can be used to illustrate the behavior of this pattern. At process B, two execution threads that are represented by the process A in a single branch of the same process instance should be merged together into a single thread of execution which is represented by the process B.

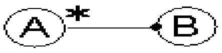

Figure 7. Thread Merge.

```
<Sequence>
  <Split-Join>
    <Process>A</Process>
    <Process>A</Process>
  </Split-Join>
  <Process>B</Process>
</Sequence>
```

Thread Split. Fig.8 can be used to represent the behavior of this pattern. After the completion of the process A, at process B, two execution threads can be initiated in a single branch of the same process instance.

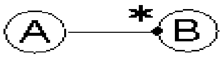

Figure 8. Thread Split.

```
<Sequence>
  <Process>A</Process>
  <Split >
    <Process>B</Process>
    <Process>B</Process>
  </Split >
</Sequence>
```

4.3 Multiple Instance Patterns.

Multiple instance patterns create several copies of workflow activities. There are seven patterns. Because OWL-S didn't consider the process execution, the following patterns can’t be formalized in term of OWL-S: Multiple Instances with a Priori Run-Time Knowledge, Multiple Instances without a Priori Run-Time Knowledge, Cancelling Partial Join for Multiple Instances, Dynamic Partial Join for Multiple Instances.

Multiple Instances without Synchronization. Fig.8 can be used to illustrate the behavior of this pattern, too. Any amount of multiple copies of a process B can easily spawn from a process A by judging the condition *f1*.

```
<Sequence>
  <Process>A</Process>
  <Repeat-while>
    <whileCondition>f1</whileCondition>
    <whileProcess>
      <Process>B</Process>
    </whileProcess>
  </Repeat-while>
</Sequence>
```

Multiple Instances with a Priori Design-Time Knowledge. Fig.9 can illustrate the behavior of this pattern. When the number of copies of process B is known at design time and the copies have to be synchronized before the execution of process C. The required number of instances of process B is three when the first task instance commences. Once all the task instances of process B have completed, the next task, process C, begins to be executed.

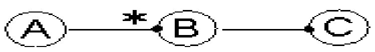

Figure 9. Multiple Instances with a Priori Design-Time Knowledge.

```
<Sequence><Process>A</Process>
  <Split ><Process>B</Process>
    <Process>B</Process><Process>B</Process>
  </Split ><Split-Join><Process>B</Process>
    <Process>B</Process><Process>B</Process>
  </Split-Join><Process>C</Process>
</Sequence>
```

Static Partial Join for Multiple Instances. Fig.9 can also be used to illustrate the behavior of this pattern. The required number of instances of process B is three when the first task instance commences. Once two of the task instances of process B have completed, the next task, process C, is triggered.

```
<Sequence><Process>A</Process>
  <Split ><Process>B</Process>
    <Process>B</Process><Process>B</Process>
  </Split ><Split-Join >
    <Choice><Process>B</Process>
      <Process>B</Process><Process>B</Process>
    </Choice><Choice><Process>B</Process>
      <Process>B</Process><Process>B</Process>
    </Choice></Split-Join ><Process>C</Process>
</Sequence>
```

4.4 State-based Patterns

State based patterns capture implicit behavior of processes that is not based on the current case rather than the environment or other parts of the process. There are five patterns, which are formalized below.

Deferred Choice. Fig.10 represents the behavior of this pattern. This pattern is much like the exclusive choice with the distinction that if process B, C or D get executed the choice is not made explicit in A rather than by the environment. The Choice can directly formalize this pattern.

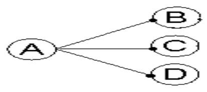

Figure 10. Deferred Choice

```
<Sequence>
  <Process>A</Process>
  <Choice>
    <Process>B</Process>
    <Process>C</Process>
    <Process>D</Process>
  </Choice >
</Sequence>
```

Interleaved Parallel Routing. Fig.11 can be used to illustrate the behavior of this pattern, which represents each process run only once, and the order of the implementation depends on the runtime, and at any one time two or more processes would not be parallelly executed. The pattern can be directly formalized by the Any-Order.

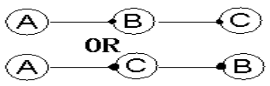

Figure 11. Interleaved Parallel Routing.

```
<Sequence>
   <Process>A</Process>
   <AnyOrder>
      <Process>B</Process>
      <Process>C</Process>
   </AnyOrder >
</Sequence>
```

Milestone. Fig.12 can illustrate the behavior of this pattern. Process B must be executed after the completion of process A and before the beginning of the process C.

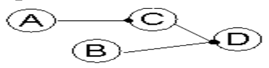

Figure 12. Milestone

```
<Sequence>
   <Process>A</Process>
   <Repeat-while>
      <whileCondition>f1</whileCondition>
      <whileProcess>
         <Choice>
            <Process>B</Process>
            <Process>C</Process> //set f1 to be false
         </Choice>
      </whileProcess>
   </Repeat-while>
   <Process>D</Process>
</Sequence>
```

Critical Section. Fig.13 can be used to illustrate the behavior of this pattern. At runtime for a given process instance, only tasks in one of the critical section can be active at any given time. Once execution of the tasks in one critical section commences, it must complete before another critical section can commence.

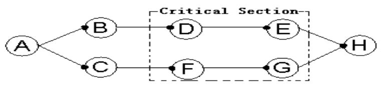

Figure 13. Critical Section.

```
<Sequence><Process>A</Process>
   <Split >
      <Process>B</Process><Process>C</Process>
   </Split ><Choice>
      <Sequence><Process>B</Process>
         <Process>D</Process><Process>E</Process>
      </Sequence><Sequence><Process>C</Process>
         <Process>F</Process><Process>G</Process>
      </Sequence></Choice><Process>H</Process>
</Sequence>
```

Interleaved Routing. Fig.4 can be used to illustrate the behavior of this pattern, too. Process A, B and C are executed in any order, but process D must be executed after the completion of process A, B and C. The Any-Order can directly formalize the pattern.

```
<Sequence>
   <AnyOrder><Process>A</Process>
      <Process>B</Process><Process>C</Process>
   </AnyOrder><Process>D</Process>
</Sequence>
```

4.5 Cancellation and Force Completion Patterns.

Cancellation and Force Completion Patterns describe the withdrawal of one or more processes that represent workflow activities. There are five patterns, which can't be formalized using OWL-S because OWL-S didn't consider the process execution.

4.6 Iteration Patterns

Iteration Patterns capture repetitive behavior in a workflow. There are three patterns. OWL-S doesn't have recursive structure and stack structure, so the Recursion Pattern can't be formalized in term of OWL-S.

Arbitrary Cycles. Fig.14 illustrates the behavior of this pattern. The only thing that must be taken care of is the re–instantiation of processes that execute repeatedly. The re–instantiation is modeled using the Repeat-Until for all processes that could be executed more than once (Process B, C, D). Process D must decide if the loop is called another time according to condition *f1*.

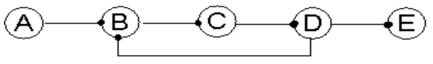

Figure 14. Arbitrary Cycles

```
<Sequence><Process>A</Process>
   <Repeat-until>
      <untilProcess ><Process>B</Process>
         <Process>C</Process><Process>D</Process>
      </untilProcess >
      <untilCondition>f1</untilCondition>
   </Repeat-until><Process>E</Process>
</Sequence>
```

Structured Loop. Fig.15 illustrates the behavior of this pattern. This pattern is much the same as Arbitrary Cycles pattern, with the difference that process E is included in the loop. There are two general forms of this pattern-the While Loop and the Repeat Loop.

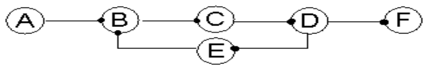

Figure 15. Structured Loop.

```
<Sequence>
   <Process>A</Process>//set f2 to be true
   <Repeat-until>
      <untilProcess >
         <If-then-else>
            <If-condition>f2</If-condition>
            <then>
               <Process>B</Process> //set f2 to be false
               <Process>C</Process>
               <Process>D</Process>
            </then>
            <Else>
               <Process>E</Process>
               <Process>B</Process> //set f2 to be false
               <Process>C</Process>
               <Process>D</Process>
            </Else>
         </If-then-else>
      </untilProcess >
      <untilCondition>f1</untilCondition>
   </Repeat-until>
   <Process>F</Process>
</Sequence>
```

The While Loop. Fig.16 can be used to illustrate the behavior of this pattern. This pattern is much the same as Arbitrary Cycles pattern, with the difference that process A must decide if the loop is called another time by judging the condition *f1*. The Repeat-While can directly formalize this pattern.

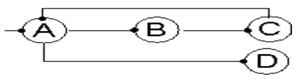

Figure 16. The While Loop.

```
<Sequence><Process>A</Process>
  <Repeat-while><whileCondition>f1</whileCondition>
    <whileProcess><Process>B</Process>
      <Process>C</Process><Process>A</Process>
    </whileProcess></Repeat-while><Process>D</Process>
</Sequence>
```

The Repeat Loop. Fig.17 illustrates the behavior of this pattern. This pattern is much the same as Arbitrary Cycles pattern, with the difference that process C must decide if the loop is called another time by judging the condition *f1*. This pattern can be directly formalized by the Repeat-Until.

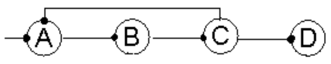

Figure 17. The Repeat Loop.

```
<Sequence><Repeat-Until>
    <UntilProcess><Process>A</Process>
      <Process>B</Process><Process>C</Process>
    </UntilProcess><UntilCondition>f1</UntilCondition>
  </Repeat-Until><Process>D</Process>
</Sequence>
```

4.7 Termination Patterns

Termination Patterns deal with the circumstances under which a workflow is considered to be completed. There are two patterns. The Explicit Termination patterns can't be formalized using OWL-S, because OWL-S didn't consider the process execution.

Implicit Termination. Fig.18 can be used to illustrate the behavior of this pattern. This pattern terminates a sub–process if no other activities can be made active. The Split provides the basis for implementation of this pattern. The process D or E, which terminates the workflow, must be random.

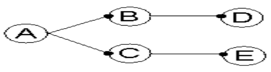

Figure 18. Implicit Termination.

```
<Sequence><Process>A</Process>
  <Split><Sequence >
      <Process>B</Process><Process>D</Process>
    </Sequence ><Sequence >
      <Process>C</Process><Process>E</Process>
    </Sequence ></Split >
</Sequence>
```

4.8 Trigger Patterns

Trigger Patterns deal with the external signals that may be required to start certain tasks. There are two patterns, which can' be formalized in term of OWL-S, because OWL-S didn't consider the process execution.

5. Conclusion

A part of workflow patterns are investigated and Formalized using the control constructs of the process model of OWL-S in this paper. OWL-S has stronger ability in semantics expression and the semantics of process model is accurate. Moreover, formalized workflows can open the door for reasoning on workflow process structures [3].

But, this paper does not describe all workflow patterns, such as Blocking Discriminator, Generalised AND-Join. This is because OWL-S didn't consider the process execution and the internal data and calculation of workflow instance of the process model, the patterns relevant to the execution can’t be formalized using OWL-S.

Further research has to be made based on the formalizations in this paper. Further resolve the error definition and the error handling issues of the implementation of the combination of Web services. For ensuring the correctness of process models formalized using OWL-S, a process model must be checked to satisfy two basic requirements[10]: (a)starting from initial state, final state will always be reachable; (b)any activity in process can be executed.

6. Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 60763008) and the Science Research Project of Yunnan University under Grant No. 2009F35Q.

7. References

[1] G Van Der Aalst, W, et al, “*Workflow patterns*,” Distributed and parallel databases, 2003. 14(1): p. 5-51.

[2] Russell, N., et al, “*Workflow control-flow patterns: A revised view*,” BPM Center Report BPM-06-22, BPMcenter. org, 2006: p. 06-22.

[3] Puhlmann, F. and M. Weske, “*Using the pi-calculus for formalizing workflow patterns*,” Lecture notes in computer science, 2005. 3649: p. 153.

[4] Kui Yu, Yanjun Qian., Gang Xue,Shaowen Yao, *Extended Workflow Patterns Description in Term of Pi-Calculus.* 4th International Conference on New Trends in Information Science and Service Science, 2010. Ⅱ: p. 654-659.

[5] Van der Aalst, W. and A. Ter Hofstede, *YAWL*, “*yet another workflow language*,” Information Systems, 2005. 30(4): p. 245-275.

[6] The OASIS Business Process TC. ebXML Business Process Specification Schema version 2. 0. 3. http :/ / www. oasis2open. org/ committees/ document s. php ? wg_abbrev = ebxml2bp , J un. 2006

[7] Andrews T , Curbera F , Dholakia H , et al . Business Process Execution Language for Web Services version 1. 1 [OL] . http :/ /www2128. ibm. com/ developerworks/ library/ specification/ ws-bpel/ , J un. 2006

[8] Arkin A , Askary S , Fordin S , et al . Web Service Choreography Interface version 1. 0 [OL] . ht tp :/ / www. w3. org/ TR/ wsci/ ,J un. 2006

[9] Martin D , Burstein M , Hobbs J , et al . OWL2S : Semantic Mark-up for Web Services [OL] . http :/ / www. w3. org/ Submission/OWL-S/ , J un. 2006

[10] Shen-sheng and YDZ, “*A pproach for workflow modeling using pi-calculus,*” Journal of Zhejiang University. Science, 2003. 4(6).