A Decentralized Framework for Executing Composite Services Based on BPMN

Yipeng Ji, Hailong Sun, Xudong Liu, Jin Zeng, Shangda Bai
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiyipeng, sunhl, liuxd, zengjin, baisd}@act.buaa.edu.cn

Abstract—**The execution of process-oriented composite services described with BPMN has become a hot research topic. In this paper, we present a decentralized execution engine for BPMN-based composite services, the basic idea of which is to make a BPMN model directly executable without converting to another executable model. An executable BPMN model named BPD is first defined. Second, a set of algorithms are proposed to execute the elements in a BPMN process. Third, we implement a composite service engine based on the proposed approach. Finally, a case study is performed to show its feasibility and effectiveness.**

Keywords-composite service; business process; workflow; execution engine

I. Introduction

With the rapid development of Web technologies, SOA (Service Oriented Architecture), as a new software pattern, has been widely used. Web service technology [1] is generally regarded as one of the key technologies to implement SOA, and its successful adoption conversely promotes the application of SOA in finance, telecommunication, ecommerce and other fields. However, an individual Web service usually cannot meet the real application requirements due to its relatively simple function. Therefore, composing individual Web services to construct advanced composite services is a straightforward approach for application development over Internet. Research on process-oriented (or process-aware) composite service, which borrows ideas from traditional workflow technique and BPM (business Process Management), has become a hot topic in SOA [1, 4, 14]. In this paper, we shall hence-forth not discriminate between the two terms of *composite service* and *business process*.

Modeling and execution are two major steps in the lifecycle of process-oriented service composition. The Business Process Modeling Notation (BPMN) [2] is a popular process modeling specification and it is not designed to be directly executable [6]. The execution of BPMN processes is generally done by mapping to a corresponding executable model that is commonly described with BPEL4WS (Business Process Execution Language for Web Services) [3]. However, some researches show that the conversion based method cannot work well in all situations [4-6]. The basic problem is the mapping of a graph to a block structure [15]. Although an extension to BPEL4WS can fulfill this purpose to a certain extent, changes applied on BPEL4WS cannot be automatically propagated back into the BPMN [4]. Since this problem, we turn to an alternative approach, the basic idea of which is to make a BPMN model directly executable. This can be achieved by defining the execution semantics and relevant attributes for each element in a BPMN model.

In this paper, we analyze the behavior semantics of all the core BPMN elements and propose an executable BPMN model named BPD (Business Process Diagram) conforming to BPMN specification version 1.1.Then we design a set of algorithms to execute the proposed model. Several workflow patterns [7] are used to compare the proposed model with BPEL and we show that some workflow patterns that are not considered in BPEL are well supported in our work. Then we implement an engine supporting decentralized execution of composite services based on BPMN. Finally, a case study is provided to show the effectiveness of the proposed approach.

Major contributions of this paper are as follows:

- We define a BPD model and a set of extended attributes, which is a fundamental step to make a BPMN process directly executable.
- We provide a set of algorithms for executing composite services based on the BPD model without mapping to BPEL processes.
- We design and implement a BPMN engine based on a segmentation algorithm, which divides a BPMN process into a number of segments so as to support the decentralized execution.

The rest of the paper is organized as follows. Section II shows the related work. Section III introduces the preliminaries of this work. In Section IV, we present the BPD model and the corresponding execution algorithms. Section V provides the implementation of the engine and a case study. Finally, in Section VI, we conclude this work.

II. Related Work

Currently, the common method to execute BPMN is by mapping to BPEL4WS. The project BABEL [5] implements the mapping from BPMN to BPEL by translating some BPMN graphical elements and some typical flow structures to the basic activities of BPEL. However, White S.A [6] and Ouyang et al. [4] concluded that some BPMN structures like GOTO, complex loops and collaboration processes, cannot be represented with BPEL4WS. This means that, not all BPMN processes can be executed by converting to another executable model. In [8], the authors proposed xBPMN to enrich the BPMN model with execution relevant details.

Another important issue is verification of a composite services model. Some researches present that CPN (Colored Petri Net) can be used to model and verify the composite services. In paper [16], Kochut et al. present the approach of design and verification for composite services based on CPN. Hui Kang et al. propose a graphical and formal modeling tool for composite services based on CPN[17]. The idea of mapping BPMN to CPN is also already presented in [18].

As mentioned above, the conversion based method presented by some related works cannot work well in all situations. Although xBPMN supports the execution of BPMN in a way, it does not consider some important BPMN elements including *pool*, *lane*, *message flow* etc. Thus, it is necessary to present a framework to completely support a BPMN model directly executable. According to the researches in this section, CPN is also a useful tool to verify our work.

III. PRELIMINARIES

To exactly present our approach, we adopt CPN to present and validate the BPD model. CPN is an effective formal method to model a computer system, and it is widely used to describe business process models.

Definition 1 CPN [9]. A CPN=(Σ, $\mathcal{P}$, $\mathcal{T}$, $\mathcal{A}$, $\mathcal{N}$), where:

- Σ is a finite set of non-empty types, called color sets.
- $\mathcal{P}$ is a finite set of places.
- $\mathcal{T}$ is a finite set of transitions.
- $\mathcal{A}$ is a finite set of arcs
- $\mathcal{N}$: $\mathcal{P}\times\mathcal{T}\cup\mathcal{T}\times\mathcal{P}\rightarrow\mathcal{A}$, is a node function returning the arcs running between places and transitions.
- $\mathcal{P}\cap\mathcal{T}=\mathcal{P}\cap\mathcal{A}=\mathcal{T}\cap\mathcal{A}=\emptyset$.

CPNs can be executed completely, and the formal execution semantics and dynamic properties, such as initial and final marking can be found in [9]. According to [10], there are mainly four methods to analyze CPNs, including interactive simulation, automatic simulation, occurrence graph, and place invariants.

IV. BPMN EXECUTION FRAMEWORK

In this section, we define a BPMN model, which includes executable semantics and attributes of all core elements. Then, some algorithms which use these semantics and attributes are proposed to execute each BPMN elements.

A. A Formal Model of BPMN Processes

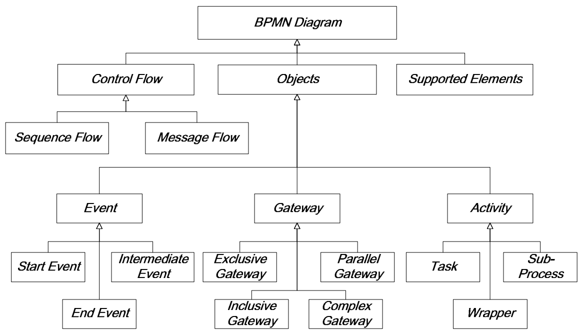


Figure 1. The hierarchical structure of BPMN model.

A composite service (or business process) modeled with BPMN is represented as a set of notations con-forming to the BPMN specification. In the following, we define a formal model of BPMN processes.

Definition 2 BPD (BPMN Process Diagram). A BPD is defined as a tuple (O, $\mathcal{A}$, $\mathcal{E}$, $\mathcal{G}$, $\mathcal{F}$, $S\mathcal{E}$), where:

- O is a set of *objects* of BPMN diagram, which includes *activities* $\mathcal{A}$, *events* $\mathcal{E}$, *gateways* $\mathcal{G}$.
- $\mathcal{A}$ is a set of *activities* that can be partitioned into disjoint sets of *tasks* $\mathcal{T}$, *sub-processes* S and *wrapper* $\mathcal{W}$.
- $\mathcal{E}$ is a set of *events* that can be partitioned into disjoint sets of *start events* $\mathcal{E}^S$, *intermediate events* $\mathcal{E}^I$ and *end events* $\mathcal{E}^E$
- $\mathcal{G}$ is a set of *gateways* that can be partitioned into disjoint sets of *parallel gateways* $\mathcal{G}^P$, *exclusive gateways* $\mathcal{G}^X$, *inclusive gateways* $\mathcal{G}^I$, and *complex gateways* $\mathcal{G}^C$
- $\mathcal{F}\subseteq O\times O$, which is a set of the control flows among *objects*
- $S\mathcal{E}$ is a set of *supported elements* in a BPMN diagram, e.g. *properties*, *expressions* and *messages*[2].

According to *Definition 2*, Fig. 1 shows the relationship among the elements of a BPD. An *object* has at least one preceding node or subsequent node. For $x\in O$ in a BPD, the preceding and subsequent nodes are given respectively by $pre(x)=\{y\in O|\ y\mathcal{F}x\}$ and $sub(x)=\{y\in O|\ x\mathcal{F}y\}$.

Additional criteria as follows should be considered to guarantee the correctness and integrity of a BPD.

Criteria 1. $|\mathcal{E}^S|\geq 1$, i.e. there is at least one *start event* in each BPD. But in a particular *pool*, there is exactly one *start event*.

Criteria 2. $|\mathcal{E}^E|\geq 1$, i.e. there is at least one *end event* in each BPD.

Criteria 3. $\forall e\in\mathcal{E}^S\ [pre(e)=\emptyset\wedge|sub(e)|\geq 1]$, i.e. a *start event* has no incoming flows while and at least one outgoing flows.

Criteria 4. $\forall e\in\mathcal{E}^E\ [|pre(e)|\geq 1\wedge sub(e)=\emptyset]$, i.e. an *end event* has at least one incoming flows while no outgoing flows.

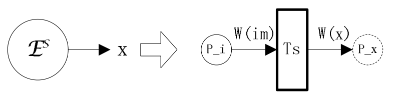
Criteria 5. $\forall o_1, o_2\in O\ [o_1\mathcal{F}o_2\rightarrow o_1\neq o_2]$, i.e. no *objects* can be in a direct flow relation with themselves.

Criteria 6. $\forall g_1, g_2\in\mathcal{G}\ [|pre(g_1)|\geq 1\wedge|pre(g_2)|\geq 1\wedge\exists g_1\mathcal{F}^* g_2\rightarrow\neg g_2\mathcal{F}^* g_1]$, i.e. no two Join *gateways* in a cycle.

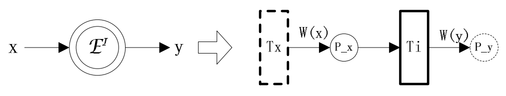
Criteria 7. $\forall o\in O, s\in\mathcal{E}^S, e\in\mathcal{E}^E\ [|pre(o)|\geq 1\wedge|sub(o)|\geq 1\wedge s\mathcal{F}^* o\wedge o\mathcal{F}^* e]$, i.e. all *objects* are on the path of a *start event* to an *end event*.

B. Characterizing A BPD with CPNs

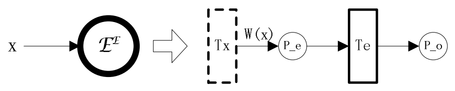
As discussed in Section III, CPNs are proved to be executable. According to [18], a model such as BPD based on BPMN v1.1 is proved that can be represented as a CPN. Due to the limitation of the space, Fig. 2(a) to (g) only illustrate the mapping of the basic structures. We assume that the arc weight (if not explicitly set) is one by default and the tokens carry true or false data. If the type of data is true, it means the transition which has this token should be invoked, or the



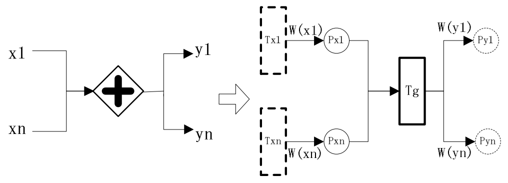
(a) start event.



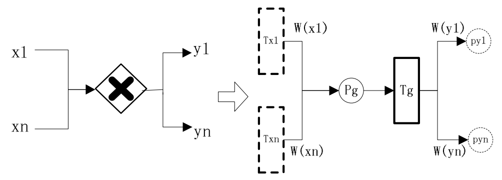
(b) intermediate event.



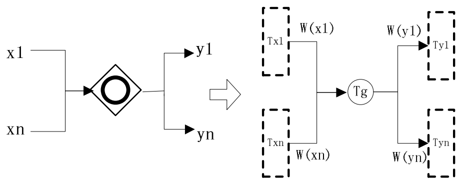
(c) end event.



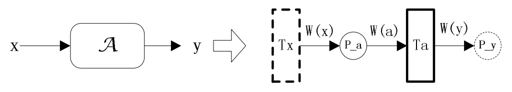
(d) parallel gateway.



(e) exclusive gateway.



(f) inclusive gateway.



(g) activity.

Figure 2. Basic mapping idea.

transition just passes the token to next places without invoking.

Any BPD can be obtained by composing the basic segments mentioned above. Therefore a BPD can be characterrized using a CPN on the basis of the mappings shown in Fig. 2(a) to (g). Then we can analyze CPNs instead of BPDs to validate the model.

Generally, behavioral properties including boundedness, liveness, home properties and dead markings are used to verify CPNs [9]. For our BPD model, there are inbound interface places (p_i) and outbound interface places (p_o) for all processes. An initial marking always has a true token in the inbound interface places. A final marking (dead marking) should therefore have token in the outbound interface places. There is no home marking which indicate that an *object* can get back to the previous state again or live transition which may lead to infinite occurrence sequences. A useless dead transition is also not involved by the model. Thus each *object* can be started and finished.

Identifying and investigating all dead markings for the model is another verification method. CPN Tools [11] offer some support for that and we manually use it to verify the model's validity that inbound interface places were cleared and tokens were placed into the outbound interface places.

C. Execution Algorithms of Core BPMN Elements

BPMN specification version 1.1 defines a set of attributes for each graphical element, but it is not enough to execute a BPMN process. We add some extended attributes (EA for short) to support the execution of BPMN.

1) Sequence Flow: sequence flow is used to order steps by connecting a source and a target *object*. Besides the attributes defined by BPMN specification, a *TOKEN* attribute is added to determine the status of downstream *object*. If a *sequence flow* has a condition expression, the value of *TOKEN* is the expression value, or else the value is true.

EA 1 Token. Determine the status of downstream *object*. Type: *Boolean*. Default Value: *true*.

2) Message Flow: In the BPMN collaboration process, different business entities are described as different pools. *message flow* is used to exchange data among *pools*. A *message flow* should know how to get remote target, so we add *URL* attribute.

EA 2 URL. Describe the target URL of a *message flow*. Type: *String*.

3) Object: object is the set of elements of BPD. Each type of *object*, such as *activity*, *event* and *gateway*, has its own attributes. However, in order to execute an *object*, some common attributes should be extended.

EA 3 Status. Determine the *object* should be executed or not. Type: *Boolean*. Default Value: *false*.

If $\forall o_1, o_2 \in \mathcal{O}$ which have a relationship ($o_1 \mathcal{F} o_2$), t(o_1,o_2) returns the token of the *sequence flow* between o_1 and o_2. The function to determine the Status is defined as follows:

$$Status[o_1] := \begin{cases} \text{true } o_1 \in \mathcal{A} \cup \mathcal{E} \cup \mathcal{G}^p\ \forall o_2 \in pre(o_1)\ t(o_2,o_1) = \text{true} \\ \text{true } o_1 \in \mathcal{G}^X \cup \mathcal{G}^I \cup \mathcal{G}^c\ \exists o_2 \in pre(o_1)\ t(o_2,o_1) = \text{true} \\ \text{false otherwise} \end{cases}$$

Executing a BPD is to execute every *object* with true *status* in the BPD. Fig. 3 shows the algorithm to execute a BPD.

As shown in Fig. 3, the input of *BPDExecution* is $\mathcal{O}$ and the output is the execution *result*. If there is no exception during execution, the *result* is true, or the value is false. The V_e is a set of executing *objects* and we set $\emptyset$ to initiate V_e (see line 1). Then the *BPDExecution* finds the start node which has no preceding nodes and adds the node to V_e (see lines 2 to 5). After that, the algorithm executes each node whose *status* is true in V_e. If a node execution is finished, it is removed from V_e and the subsequent nodes are added (see lines 6

```
BPDExecution()
input: $O$
output: result
1  $V_e \leftarrow \emptyset$
2  for each $o_i \in O$
3    if $pre[o_i] = \emptyset$
4      then $status[o_i] \leftarrow$ true
5      $V_e \leftarrow V_e \cup \{o_i\}$
6  while $V_e \neq \emptyset$
7    for each $o_j \in V_e$
8      for each $o_k \in sub[o_j]$
9        $status[o_k] \leftarrow t(o_k, o_j)$
10     if $status[o_j]$=true
11       then EXECUTE[$o_j$]
12   $V_e \leftarrow V_e - \{o_j\}$
13   $V_e \leftarrow V_e \cup sub[o_j]$
14 return result
```

Figure 3. Algorithm of BPD execution.

```
Split()
input: $g \in \mathcal{G}$
output: result
1  if $|sub[g]| \geq 1$
2    then for each $o_i \in sub[g]$
3      if $g \in \mathcal{G}^X$
4        then if $t(g,o_i)$=true
5          then $status[o_i] \leftarrow$ true
6          break
7        else if $t(g,o_i)$=true
8          then $status[o_i] \leftarrow$ true
9  return result
```

Figure 4. Algorithm of split gateway execution.

```
Join()
input: $g \in \mathcal{G}$
output: result
1  if $|pre[g]| \geq 1$
2    then for each $o_i \in pre[g]$
3      if $g \in \mathcal{G}^X$
4        then if $t(o_i,g)$=true
5          then $status[g] \leftarrow$ true
6          break
7        else if $t(o_i,g)$=true
8          then $status[g] \leftarrow$ true
9  SYNCHRONIZE[$g$]
10 return result
```

Figure 5. Algorithm of join gateway execution.

to 13). The algorithm ends when V_e becomes $\emptyset$.

4) Event: event $\mathcal{E}$ can be divided into *start event* $\mathcal{E}^S$, *intermediate event* $\mathcal{E}^I$ and *end event* $\mathcal{E}^E$. Each *event* has *TRIGGER* attribute attached *event details*. The key step is to get *details* from *trigger* of an *event* and execute them by their own semantics.

5) Activity: Different *activities* have different algorithms. *task* is a useful *activity* and in this section, we would like to present the algorithm to execute *service task*. The algorithms to execute *send task* and *receive task* will be particularized in Section IV.D.

The first step to execute a *service task* is to get the input set for Web service from *InMessageRef*. Then *service task* uses the input set to invoke the Web service and receives the result saved in output set to set the *OutMessageRef*.

6) Gateway: Generally, there are two kinds of *gateway*. One is *split gateway* and the other is *join gateway*. The core algorithms are designed as follows:

The *sequence flows* connecting the *split gateway* and its subsequence nodes have condition expressions to determine the value of tokens. As shown in Fig. 4, if g is an *exclusive gateway*, only one *sequence flow* which has a true token would be taken and the other subsequence nodes are ignored (see lines 3 to 6). Otherwise, all true *sequence flows* are considered (see lines 7 and 8).Fig. 5 shows an *exclusive gateway* only concerns about one preceding node which has a relationship with a true token and ignores other nodes (see lines 3 to 6). The other *gateways* will synchronize all preceding nodes with true *sequence flow* (see lines 7 to 9).

D. A Decentralized Execution Mechanism

In the collaboration process, participants are always not in the same location and want to interact with remote partner. It is necessary to provide a mechanism to make the decentralized execution of BPMN process feasible.

In order to implement the decentralized execution of BPMN, we adopt segmentation approach. This paper defines *INSTANCE* to denote the execution of a BPMN collaboration process. An instance includes some *SEGMENTATION TASKs*. Each task can be executed independently and uses

```
<parts jobId="jobId">
        <part id="id" poolId="poolId" >
                <node id="id"/>
                …
        </part>
        …
</parts>
```

Figure 6. Segmentation definition format.

```
ExecuteSendTask()
input: set∈Send Task
output: result
1  ASSIGNMENT[set]
2  message←GETMESSAGEFLOW[set]
3  record←SEND[message]
4  NOTIFYTASK[url[message]]
5  return result
```

Figure 7. Algorithm of send task execution.

```
ExecuteReceiveTask()
input: $rt \in$ ReceiveTask
output: $result$
1 ASSIGNMENT[$rt$]
2 $message \leftarrow$ GETMESSAGEFLOW[$rt$]
3 while $message \not\subseteq record$
4     WAIT[$rt$]
5 return $result$
```

Figure 8. Algorithm of receive task execution.

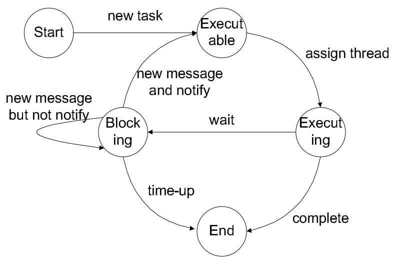


Figure 9. Segmentation tasks state diagram.

send task and *receive task* which are connected by *message flows* to exchange data and resources with other *tasks*.

In BPMN process, each *pool* indicates a participant. So it is reasonable to divide BPMN process into segments by *pools*. We use a XML file to describe the segmentation. Fig. 6 shows the format of the XML file.

In this way, a segmentation task is based on a complete pool in BPMN process. We can use the *BPDExecution* and other algorithms mentioned in Section IV.C to start execution of the segmentation task. When the task needs to communicate with other remote tasks of the same instance, *send task* and *receive task* should be involved to accomplish this function. Fig. 7 and 8 shows the algorithms to execute *send task* and *receive task*.

Fig. 7 shows that a *send task* first does the assignment to prepare properties for the *message* which it will send (see line 1). Then it receives *message* reference from *message flow* (see line 2). After that, the *message* attached properties is send to *record* which is a global variable and the *send task* notifies a segmentation task aimed by *URL* attribute of the *message* (see lines 3 and 4). *Receive task* is used to get messages from other tasks. Fig. 8 indicates it needs to gain the *message* reference from message *flow* and find that if the *message* is already in the *record* or not (see lines 2 and 3). If the *message* is not in record, the *receive task* will wait until a new *message* notifies it (see line 4).

A segmentation task will be blocked when all *receive tasks* are waiting. A blocked task cannot be executed sequentially until other tasks notify it. So there are four states of a segmentation task: Blocking, Executable, Executing and End. Fig. 9 shows the states switching of segmentation tasks.

E. Comparative Analysis of BPD

In the following, we provide a comparative functional analysis of our BPD model. As we have mentioned in Section I, process-oriented composite service research is based on workflow technologies to some extent. Therefore, as shown in TABLE I, we compare BPD with BPEL in terms of their respective support for typical workflow patterns [7].

We have analyzed 43 workflow patterns and the result show that BPD completely or partially supports 33 patterns while BPEL only supports 21 patterns. For example, BPD is based on graph structure so that it can also support Arbitrary

TABLE I. COMPARISON OF BPMN AND BPEL

Workflow Patterns	BPD Model	BPEL
Sequence	+	+
Parallel Split	+	+
Synchronized	+	+
Exclusive Choice	+	+
Simple Merge	+	+
Multi-Choice	+	+
Structured Synchronizing Merge	+	+
Multi-Merge	+	-
Structured Discriminator	+/-	-
Arbitrary Cycles	+	-
Implicit Termination	+	+
Multiple Instances without Synchronization	+	+
Multiple Instances with a Design-Time Knowledge	+	-
Multiple Instances with a Run-Time Knowledge	+	-
Multiple Instances without a Run-Time Knowledge	-	-
Deferred Choice	+	+
Interleaved Parallel Routing	-	+/-
Milestone	-	-
Cancel Activity	+	+
Cancel Case	+	+
Structured Loop	+	+
Recursion	-	-
Transient Trigger	-	-
Persistent Trigger	+	+
Cancel Region	+/-	+/-
Cancel Multiple Instance Activity	+	-
Complete Multiple Instance Activity	-	-
Blocking Discriminator	+/-	-
Cancelling Discriminator	+	-
Structured Partial Join	+/-	-
Blocking Partial Join	+/-	-
Cancelling Partial Join	+/-	-
Generalized AND-Join	+	-
Static Partial Join for Multiple Instances	+/-	-
Cancelling Partial Join for Multiple Instances	+/-	-
Dynamic Partial Join for Multiple Instances	-	-
Local Synchronizing Merge	-	+
General Synchronizing Merge	-	-
Critical Section	-	+
Interleaved Routing	+/-	+
Thread Merge	+	+/-
Thread Split	+	+/-
Explicit Termination	+	-

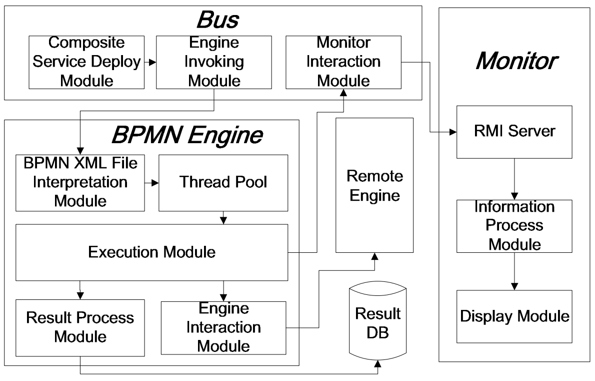

Bus
Composite Service Deploy Module
Engine Invoking Module
Monitor Interaction Module
Monitor
RMI Server
Information Process Module
Display Module
BPMN Engine
BPMN XML File Interpretation Module
Thread Pool
Execution Module
Remote Engine
Result Process Module
Engine Interaction Module
Result DB

Figure 10. BPMN engine structure.

Cycles which cannot be supported in BPEL. Thus, compare to BPEL, BPD is more complete.

V. System Implementation and a Case Study

A. Design and Implementation of a BPMN Engine

This paper presents a decentralized composite ser-vice engine based on the BPMN model and execution algorithms mentioned in Section IV. We have defined a BPMN schema in order to save a BPD as a XML file. The system structural is shown in Fig. 10.

First, we model a composite service by BPMN Model and save it as a XML file. Then the persistent file is deployed to Bus by Composite Service Deploy Module. When Bus receives a request, Engine Invoking Module creates an instance and divided it into segmentation tasks, then invokes BPMN Engines to execute every segmentation tasks. After interpreting the XML file by Interpretation Module, the segmentation task is pushed into Thread Pool and changes state to executable. If a thread is assigned to an executable task, the task is executed by Execution Module immediately. During the execution, a task uses Engine Inter action Module to communicate with other tasks executed in Remote Engine. Information of the execution is transferred to Monitor by Monitor Interaction Module and shown by Display Module. If a task has finished, Result Process Module will insert the result to Result Database.

In BPMN XML File Interpretation Module, we developed a toolkit -- BPMN4J to facilitate the persistent file to java objects of BPMN elements. BPMN4J is a tool based on EMF [13] which serializes the BPMN model to the java objects in memory. We implement all algorithms presented in Section IV and Execution Module uses these algorithms to execute each *object*. Monitor Interaction Module uses RMI to communicate with Monitor. Bus and BPMN engine are released as a Web service, and Monitor is an Eclipse RCP application. All programs are developed with JDK 6.0.

B. A Case Study

In this section, we analyze the effectiveness and performance of BPMN engine by some case studies. All case studies and experiments are performed in a PC (Intel core 2 1.86G CPU, 1G memory).

Fig. 11 shows the purchase book composite service which is a representative collaboration process with two remote participants, client and book store. First, a client uses *service task* login to transfer information to book store as a remote partner. If successful, the book store finds the book which the user wants to purchase. Otherwise, a failed message is replied to client. If the book is found, the book store calculates the discount and replies the price. Or the book store informs the client that the book is not found. As we mentioned above, it is difficult to describe a collaboration process by BPEL and participants here are decentralized. So the decentralized execution approach presented in this paper is necessary to make this process work. Fig. 12 is the view of the monitor for our engine. The *activity* with gray has been executed already and the pink *activity* is invoking at the moment. It indicates the process is executed correctly. Fig. 13 explains the relationship between amount of composite service live instance and response time for the purchase book process. We vary the number of concurrent live instances from 100 to 1000 to get various results respectively.

In order to evaluate this engine, it is necessary to compare our work with other systems. However, there is not a widely used BPMN engine currently. So we decide to compare the engine provided in this paper with ActiveBPEL, which is a representative BPEL engine.

As mentioned above, purchase book process is a representative collaboration process which cannot be described by BPEL. Thus, we define an echo process can be represented

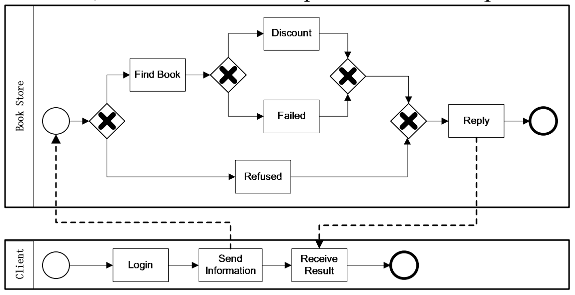
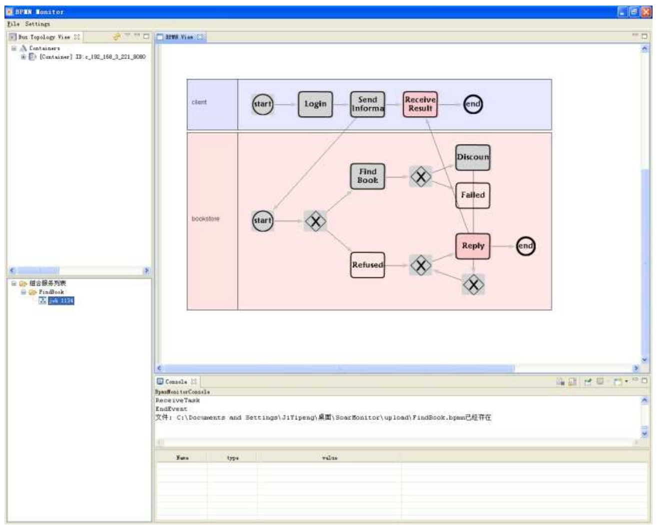


Figure 11. BPMN process of purchase book.



```
BpmMonitorConsole
ReceiveTask
EndEvent
文件: C:\Documents and Settings\JiYipeng\桌面\SoarMonitor\upload\FindBook.bpmn已经存在
```

Name	type	value

Figure 12. Monitor of executing the purchase book process.

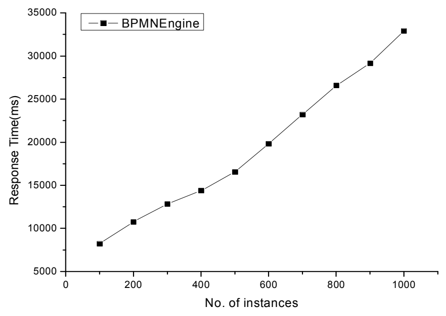


Figure 13. Purchase book process: Response time v.s. No. of instances.

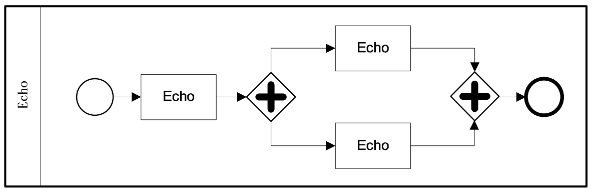


Figure 14. BPMN process of echo service.

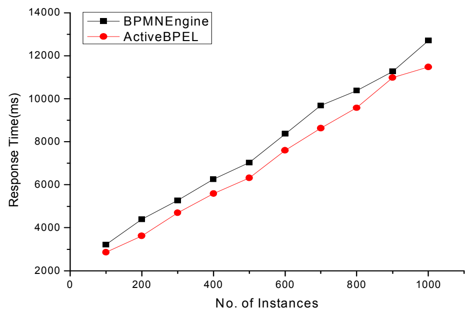


Figure 15. BPMN Engine v.s. ActiveBPEL.

with both BPMN and BPEL to explain the comparison of our work and ActiveBPEL. Fig. 14 shows a composite service which has three *service tasks* with the echo service which just invokes a remote service to return a string. The logic of the echo service is too simple so that the cost of performance is only from engine and network environment. We use BPMN engine and ActiveBPEL to execute it respectively in the same network environment and get various results which can indicate the difference of performance of BPMN engine and ActiveBPEL.

As shown in Fig. 15, the execution time of BPMN engine is about 800 ms more than ActiveBPEL. But for the better support to workflow patterns, this performance cost is acceptable.

VI. Conclusion and Future Work

In this paper, we introduce a decentralized execution framework for composite service based on BPMN. First, we give a BPD model and propose a set of algorithms to execute BPMN *objects*. Then this paper provides a decentralized execution mechanism to invoke collaboration process. We evaluate the proposed framework using workflow patterns and show a better support compared to BPEL. Finally, this paper describes the implementation of an engine based on the framework mentioned above. A set of case studies and experiments is performed to show feasibility and effectiveness of the framework. The future work is to support more workflow patterns and optimize the algorithms.

Acknowledgment

This work was supported by the National High Technology Research and Development Program of China (863 program) under grant 2007AA010301.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. Web Services Concepts, Architectures and Applications. Springer Verlag, 2004.

[2] BPMI.org, OMG. Business Process Modeling Notation 1.1. http://www.omg.org/spec/BPMN/1.1/, [June 25, 2008].

[3] Business Process Execution Language for Web Services version 1.1.http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, [September 25, 2005].

[4] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, and W.M.P. van der Aalst. From BPMN Process Models to BPEL Web Services. Proceedings of the IEEE International Conference on Web Services (ICWS'06)-Volume 00, pages 285–292, 2006.

[5] BABEL Project. Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages. http://www.bpm.fit.qut.edu.au/projects/babel/, [May 10, 2009].

[6] S.A. White. Using BPMN to Model a BPEL Process. BPTrends, pages 1-18, 2005.

[7] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(3), pages 5-51, 2003.

[8] A. Grosskopf. xBPMN: Formal Control Flow Specification of a BPMN based Process Execution Language. Master Thesis, Potsdam University, 2007.

[9] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1. Springer, 1996.

[10] K. Jensen. Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[11] CPNTOOLS. http://wiki.daimi.au.dk/cpntools, [May 15, 2009].

[12] J. DESEL and W. REISIG. Place/transition Petri nets. Lecture notes in computer science, pages 122–173, 1998.

[13] Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/emf/, [April 23, 2009].

[14] L.J. Zhang, J. Zhang, and H. Cai, Services Computing. Tsinghua University Press, Beijing, 2007.

[15] J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages.Technical report, Queensland University of Technology, 2006.

[16] X. Yi, and K. J. Kochut. Process Composition of Web Services with Complex Conversation Protocols: a Colored Petri Nets Based Approach. In Proc. DASD, pages 141-148, 2004.

[17] Hui Kang, Xiuli Yang, and Sinmiao Yuan. Modeling and Verification of Web Services Composition based on CPN. Network and Parallel Computing Workshops, 2007.

[18] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Formal Semantics and Automated Analysis of BPMN Process Models. QUT eprint, ID-5969, 2007.