2009 IEEE International Conference on Web Services

LiveMig: An Approach to Live Instance Migration in Composite Service Evolution

Jin Zeng, Jinpeng Huai, Hailong Sun, Ting Deng and Xiang Li
School of Computer Science and Engineering, Beihang University, Beijing, China
{zengjin, sunhl, dengting,lixiang }@act.buaa.eud.cn
huaijp@buaa.edu.cn

Abstract

Composite service evolution is one of the most important challenges to deal with in the field of service composition. In particular, this paper presents, LiveMig, an approach to live migration of composite service instance, which is a critical step for online composite service evolution. In LiveMig, a set of change operations preserving soundness is first defined. Second, a live instance state migration algorithm is proposed to determine if the state migration is allowed or not, and to compute the exact state after the migration. Finally, the correctness of LiveMig is theoretically proved, and an extensive set of simulations are performed to show its feasibility and effectiveness.

1. Introduction

Service composition is widely considered as an effective method to support the development of business applications using loosely-coupled and distributed web services over the Internet[1]. Currently, process-oriented (process-aware) service composition has become the research mainstream in services computing, which borrows the idea from traditional workflow technique and Business Process Management (BPM) methodology*.

Due to the highly dynamic changes of business requirements and Internet environments, composite service evolution is one of the most important challenges to deal with in the field of service composition. The dynamic evolution requires composite services to be capable of self-adapting according to the changes of business requirements and runtime environments. The main exhibition of dynamic composite service evolution are changeability of component services, adjustability and configurability of structural relations[2].

The realization of dynamic composite service evolution is faced with a lot of technical challenges[3-6]. In particular, there is an imperative need for an approach that supports the live instance migration when an old composite service definition is changed to a new one. When a composite service definition is experiencing dynamic evolution, its instances may still be running at the same time. For example, in order to satisfy new business requirements, a composite service for an e-commerce order-processing in a SCM (Supply Chain Management) system may need to partially adjust the business logic so as to support the new business pattern defined by RosettaNet PIP. Specifically, the original financial reporting function needs to be extended to support the external examination from a third-party auditing company. With this extension, the newly proposed Sarbanes-Oxley can be supported. In these scenarios, business services must run for 7*24 hours without any interruption, and the demand of dynamic evolution is more urgent. These critical scenarios neither allow composite service instances to be aborted and rolled back due to expensive time costs, nor allow the existence of multiple versions of composite service instances because the old versions are not compliant with new business constraints. Therefore we need an approach to live instance migration in composite service evolution.

For live instance migration in composite service evolution, early research[3, 7, 8] identifies a challenging problem "dynamic change bug", which means the states of composite service instances of an old process definition does not have any corresponding states that exist in the new one. The change regions are adopted to restrict the migration of some instances that are under certain states (*e.g.* invalid structural adjustment). However, the computational complexity of change regions is proved to be very high[8], thus process inheritance[9] and relaxed process projection[10] methods are proposed. But these methods do not support structural adjustments of business processes and provide no solutions to computing the new state after evolution.

Another important issue is the correctness guarantee during composite service evolution. In process-oriented composite service, the most important correctness criterion is structural soundness (or *soundness* for short)[3, 11]. For a complex business process it may be intractable to verify the soundness. Especially under the scenarios of online composite service evolution, there is not enough time to do the verification. In a word, it is highly important and challenging to find an effective method that can satisfy evolution requirements while preserve soundness after composite service evolution.

*We shall henceforth not discriminate between the two terms of composite service and business process. In this paper, composite services also refer to process-oriented composite services.

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.96

This paper presents an approach, LiveMig, to addressing the two problems mentioned above for live instance migration in composite service evolution. In LiveMig, a set of change operations preserving soundness is first defined to avoid complex verification. Then a live instance state migration algorithm based on WF-net[11] is proposed to achieve live instance migration in composite service evolution. The algorithm converts a sound WF-net to a reachability graph in which each vertex and edge of a graph represents a state and a transition respectively. On the basis of reachability graph and transition sequences (in WF-net, named as firing sequences), the algorithm can determine if a state migration is allowed or not, and to compute the state after the migration. Finally, the correctness of LiveMig is theoretically proved, and an extensive set of simulations are performed to show its feasibility and effectiveness.

Major contributions of this paper are as follows:

- We introduce a set of change operations preserving soundness. And we also prove that the soundness of new composite service can be satisfied after using the change operations. With this, complex verification process is avoided.
- We propose a live instance migration algorithm to determine if a state migration is allowed or not, and to compute the exact state after the migration. This algorithm removes the risk of incorporating dynamic change bugs.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. Section 3 and 4 introduce a specific scenario and the preliminaries of this paper. In Section 5 and 6, we give a set of change operations preserving structural soundness and an instance state migration algorithm in composite service evolution. Section 7 describes a case study and experiments. Finally, we wrap up this paper with some conclusions and future work in Section 8.

2. Related Work

Some surveys[3-5, 12] about composite service or business process evolution have analyzed the problems brought by evolution time, evolution impact, evolution operation categorizing and evolution management, however ensuring correctness of evolution and dealing with live instances is very challenging. Rinderle et al. [13] discussed the correctness criterion and thought completeness, correctness and change realization are three fundamental issues about business process dynamic change. Specifically to live instance state migration, Rinderle-Ma et al.[14] thought it is considered that evolution should realize number of migratable instances is maximized. About instance migration, Ellis identifies a notorious problem—dynamic change bug[7], that is the state in old instance has no corresponding state in new instance(incorrect state migration maybe cause that component service will be invoked again or be invalidly skipped). Ellis based on PetriNet model provided a concept "change region", that is the live instances inside change region can not be migrated (if migrated, result in dynamic change bug). Aalst[8] addressed change region proposed by Ellis is just static change region (SC) and SC is a neither sufficient nor necessary condition. So, Aalst proposed the definition and algorithm about "dynamic change region". This algorithm could accurately obtain sufficient change regions. Unfortunately, the complexity of the algorithm is factorial ($O(n^4(n!)^2)$) and dynamic change region only is sufficient and is not necessary. literature[9] presented a notion of process inheritance including protocol inheritance, projection inheritance, protocol/projection inheritance and life-cycle inheritance, and it can be proved that the live instance state can be validly migrated during the business process evolution under restriction of four inheritances. Essentially, different "inheritance" method is different projection operation of process, so that such evolution based on inheritance does not support service deletion and structural adjustments. To above weakness of process inheritance, a relaxed definition of projection operation[10] were proposed to support activity deletion in process and improved further the algorithm for change regions, but it has not provided how to computing new state after evolution.

3. Scenario

In this section, we give an evolution scenario for an order-processing in a SCM system. As shown in Figure 1, we use Petri nets to describe the instance migration problem. Figure 1a depicts a composite service for order-processing, which consists of five component services including Register, Order, Pay, Assemble and Supply. Since there exists a dependent relationship between Pay service and Assemble service, both services can be executed in parallel. In Figure 1b, for providing multiple payment methods, the composite service not only supports traditional Pay service (bank transfer) but also supports a new PayPal service (an online payment service offered by a third party). Adding PayPal is a choice relation with old Pay service and does not add any new places. Thus it is always feasible for an instance to migrate from Figure 1a to Figure 1b. Considering some mistakes may happen in assembling goods, a Check service is added after Assemble service in Figure 1c. Obviously, a new place is added and some possible states are also imported, but these changes do not cause any problems for instance migration from Figure 1b to Figure 1c. Moreover, some customers order goods, but may not intend to pay, which results in invalid goods assembling. Therefore customers are requested to assemble goods after payment in Figure 1d. Especially, this change can cause the notorious *dy-*

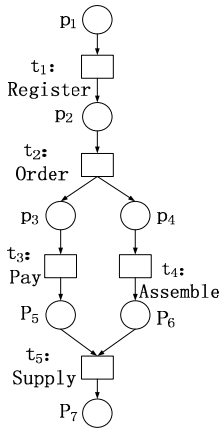

Figure 1a. Original Composite service

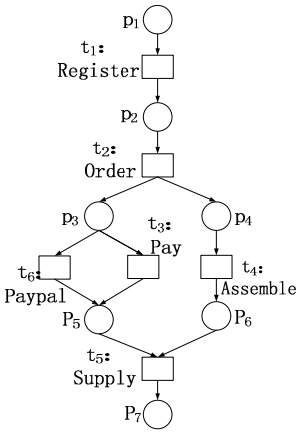

Figure 1b. Add a PayPal service

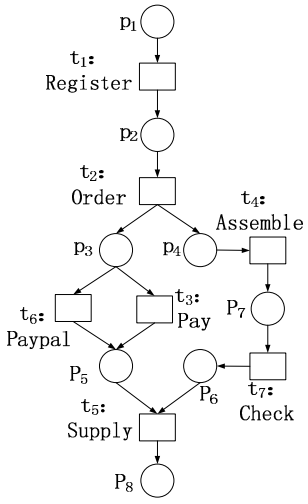

Figure 1c. Add a Check service

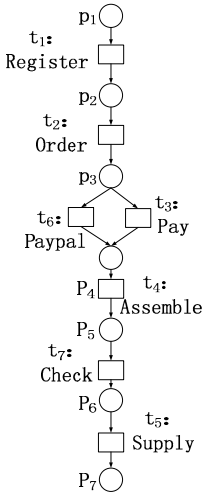

Figure 1d. Adjust the process structure

namic change bug problem, *i.e.*, the state with a token in both p_3 and p_6 (Assemble and Check have been executed and Pay or PayPal will be executed in Figure 1c) will cause problems because there is no corresponding state in the sequential process (Figure 1d). In section 7, we will further discuss how to apply our LiveMig to achieve valid instance migration and remove the risk of incorporating dynamic change bugs.

4. Preliminaries

To exactly present our approach, LiveMig, we adopt WF-net to formally describe a composite service. WF-net is a special Petri net and possesses the advantages of formal semantics definition, graphical nature and analysis techniques.

Definition 1 (WF-net)[11]*: A Petri net WFN=(P,T,F) is a WF-net (Workflow net) if and only if:*

- *There is one source place $i \in P$ such that $\bullet i = \Phi$;*
- *There is one sink place $o \in P$ such that $o\bullet = \Phi$; and*
- *Every node $x \in P \bigcup T$ is on a path from i to o;*

It should be noted that a WF-net specifies the dynamic behavior of a composite service instance. In WF-net, tasks (component services) are represented by transitions, conditions are represented by places, and flow relationships are used to specify the partial ordering of tasks. In addition, the *state* of a WF-net is indicated by the distribution of tokens amongst its places. We use is a $|P|$ dimension vector M to present the state of a live procedure in WF-net, where every element means the number of tokens in a place. We denote the number of tokens in place p in state M by M_p. Specifically, we use M_0 (M_{end}) to denote initial state (final state) which has only token in source (sink) place. A transition $t \in T$ is enabled in state M iff $M_p > 0$ for any place p *such that* $(p,t) \in F$. If t is enabled, t can fire leading to a new state M' such that $M'_p = M_p - 1$ and $M'_{p'} = M_{p'} + 1$ for each $(p,t) \in F$ and $(t,p') \in F$, which is denoted by $M\,[t>M'$.

Definition 2 (Sound)[11]*: A procedure modeled by a WF-net WFN=(P, T, F, i, o, M_0) is sound if and only if:*

- *For every state M reachable from initial state M_0, there exists a firing sequence leading from state M to final state M_{end}. Formally:*
$$\forall M (M_0 \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} M_{end})$$
- *State M_{end} is the only state reachable from state M_0 with at least one token in place o. Formally:*
$$\forall M (M_0 \xrightarrow{*} M \wedge M \geq M_{end}) \Rightarrow (M = M_{end})$$
- *There are no dead transitions in WFN. Formally:*
$$\forall t \in T \; \exists M, M' \text{ s.t. } M_0 \xrightarrow{*} M \xrightarrow{t} M'$$

Soundness is an important correctness criterion that guarantees proper termination of composite service. The first requirement in Definition 2 means that starting from the initial state M_0, it is always possible to reach the state only with one token in place o. The second requirement states that when a token is put in place o, all the other places should be empty. The last requirement states that there are no dead transitions in the initial state M_0. Generally speaking, for a complex WF-net it may be intractable to decide soundness[15]. In this paper we do not verify soundness of an evolved composite service, but preserve the one using a basic change operation set.

In addition, all of the discussed composite services are modeled by WF-net without any cyclical structures and a component service (transition) appears only once in a composite service (WFN). Essentially, the key issue in instance migration is to obtain the state of the migrated instance on the basis of the state before migration and the specific migration operations. We do not consider data flow and data constraints issues in this work.

5. Set of Change Operations

In the section a set of change operations is defined including replacement, addition, deletion and process structural adjustments. We show that a sound WF-net evolved with the set of change operations can preserve soundness all the same.

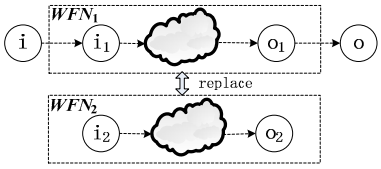

Figure 2. Replacement

Definition 3 (Replacement Operation): *Let $WFN_1=(P_1,T_1,F_1, i_1,o_1,M_{01})$ be a sound sub net of a sound WF-net $WFN=(P,T,F,i,o,M_0)$ and $WFN_2=(P_2,T_2,F_2,i_2,o_2, M_{02})$ be a sound WF-net such that $P_1\cap P_2=\Phi$, $T_1\cap T_2=\Phi$, $F_1\cap F_2=\Phi$, then $WFN'=(P', T', F', i', o', M_0')$ is the WF-net obtained by replacing WFN_1 by WFN_2, such that $P'=(P\backslash P_1)\cup P_2$, $T'=(T\backslash T_1)\cup T_2$, $F'=(F\backslash F_1)\cup F_2\cup F''$, where $F''=\{(x,i_2)\in P\times T_2 | (x, i_1)\in F_1\}\cup\{(o_2, y)\in T_2\times P | (o_1, y)\in F_1\}$, and initial state M_0' is a $|P'|$ dimension vector, where $S\backslash S'$ denotes the difference between set S and S', i.e. the set of elements which is in S and not in S'.*

We know if a transition in a sound WF-net is replaced by a other sound WF-net, then the resulting WF-net is also sound (Theorem 3 in literature [16]). Our replacement operation is an extension to above mentioned result, because a sound WF-net behaves like a transition. Obviously, we have the conclusion:

Proposition 1. *Replacement operation can preserve soundness of a sound WF-net.*

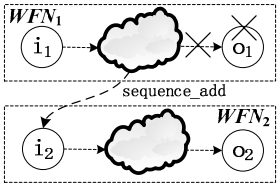

Figure 3a. Sequence_add

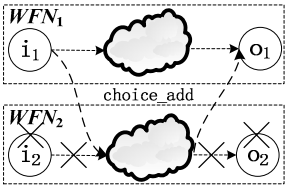

Figure 3c. Choice_add

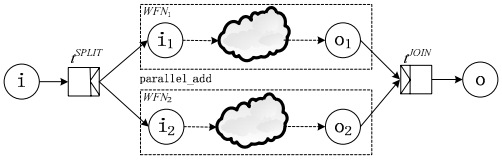

Figure 3b. Parallel_add

Definition 4 (Addition Operations): *Let $WFN_1=(P_1,T_1,F_1, i_1, o_1, M_{01})$ and $WFN_2=(P_2,T_2,F_2,i_2,o_2,M_{02})$ be two sound WF-nets, such that $P_1\cap P_2=\Phi$, $T_1\cap T_2=\Phi$, $F_1\cap F_2=\Phi$.*

- ***Sequence_add ($WFN_1\rightarrow WFN_2$)****: $WFN=(P, T, F, i_1, o_2, M_0)$ is the WF-net obtained by sequence_adding WFN_1 with WFN_2, where $P=(P_1\backslash\{o_1\})\cup P_2, T=T_1\cup T_2, F=\{(x,y)\in F_1|y\neq o_1\}\cup\{(x, i_2)\in T_2\times P_1|(x, o_1)\in F_1\}\cup F_2$ and M_0 is a $|P|$ dimension vector (where first element is 1 and rest ones are 0);*
- ***Parallel_add ($WFN_1||WFN_2$)****: $WFN=(P, T, F, i, o, M_0)$ is the WF-net obtained by parallel_adding WFN_1 with WFN_2, where $P= P_1\cup P_2\cup\{i, o\}$, $T= T_1\cup T_2\cup\{t^{SPLIT}, t^{JOIN}\}$, $F=F_1\cup F_2\cup\{< i , t^{SPLIT} >, < t^{SPLIT} , i_1>, < t^{SPLIT} , i_2>, < o_1, t^{JOIN} >, <o_2, t^{JOIN} >, < t^{JOIN} , o>\}$ and M_0 is a $|P|$ dimension vector (where first element is 1 and rest ones are 0);*
- ***Choice_add (WFN_1+WFN_2)****: $WFN=(P, T, F, i_1, o_1, M_0)$ is the WF-net obtained by choice_adding WFN_1 with WFN_2, where $P=P_1\cup(P_2\backslash\{i_2, o_2\})$, $T=T_1\cup T_2$, $F=F_1\cup\{(x, y)\in F_2| x\neq i_2\wedge y\neq o_2\}\cup\{(i_1, y)\in P_1\times T_2|(i_2, y)\in F_2\}\cup\{(x, o_1)\in T_2\times P_1|(x, o_2)\in F_2\}$ and M_0 is a $|P|$ dimension vector (where first element is 1 and rest ones are 0).*

Proposed add operations are sequential, parallel and choice adding a sound WF-net with another sound WF-net, obviously so obtained new WF-net is sound. Hence we have the conclusion:

Proposition 2. *Addition operations can preserve soundness of a sound WF-net.*

Delete operations are the reverse of addition operations; we do not discuss a series of delete operations due to the limitation of paper space. To composite service evolution，besides above replacement, addition and deletion operations, sometimes we need deal with structural adjustments of composite services process, for example, Figure 1d is adjusted from parallel to sequence based on Figure 1c. In this paper, five basic structural adjustment operations are presented to allow modifying some sound sub nets in a sound WF-net.

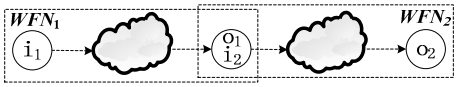

Figure 4a. Original sequence structure of two WF-net

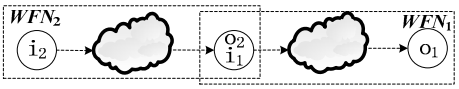

Figure 4b. Reversal sequence adjustment

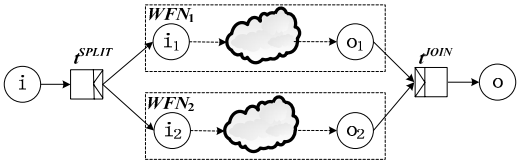

Figure 4c. Sequence_to_parallel adjustment

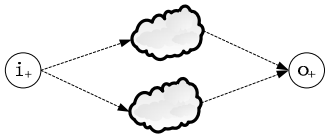

Figure 4d. Sequence_to_choice adjustment

Definition 5 (sequence structure adjustments)*: Let $WFN= (P, T, F, i, o, M_0)$ be a sound WF-net consisting of $WFN_1=(P_1,T_1,F_1,i_1,o_1, M_{01})$ and $WFN_2=(P_2,T_2,F_2,i_2,o_2,M_{02})$ in sequence, such that $P=P_1 \cup P_2$, $T=T_1 \cup T_2$, $F=F_1 \cup F_2$, $i=i_1$, $o_1=i_2$, $o=o_2$ (see Figure 4a).*

- ***Reversal_sequence adjustment****: $WFN' =(P, T , F, i',o' , M_0)$ is the WF-net obtained by reversal sequence adjustment between WFN_1 and WFN_2, where $i'=i_2$, $o'=o_1$, $o_2=i_1$; (see Figure 4b)*
- ***Sequence-to-parallel adjustment****: $WFN_{||}=(P_{||}, T_{||}, F_{||}, i_{||},o_{||},M_{0||})$ is the WF-net obtained by sequence_to_parallel adjustment between WFN_1 and WFN_2, where $P_{||}= P \cup\{i_{||}, o_{||}\}$, $T_{||}=T \cup\{ t^{SPLIT}, t^{JOIN} \}$, $F_{||}=F \cup\{< i_{||}, t^{SPLIT} >, < t^{SPLIT} , i_1>, < t^{SPLIT} , i_2>, <o_1, t^{JOIN} >, <o_2, t^{JOIN} >, < t^{JOIN}, o_{||}>\}$ and $M_{0||}$ is a $|P_{||}|$ dimension vector (where its first element is 1 and rest ones are 0). (see Figure 4c)*
- ***Sequence_to_choice adjustment****: $WFN_+=(P_+,T_+,F_+ i_+, o_+,M_{0+})$ is the WF-net obtained by sequence_to_choice adjustment between WFN_1 and WFN_2, where $P_+=(P\backslash\{i_x, i_y, o_x, o_y\}) \bigcup \{i_+, o_+\}$, $T_+=T$, $F_+=\{(p, q) \in F|p\neq i_1, i_2 \wedge q\neq o_1, o_2\} \cup\{(i_+, q)|(i_1, q) \in F_1 \vee (i_2, q) \in F_1 \} \cup\{(p, o_+)|(p, o_1) \in F_1 \vee (p, o_2) \in F_2\}$ and M_{0+} is a $|P_+|$ dimension vector (where its first element is 1 and rest ones are 0). (see Figure 4d)*

Similarly the "*parallel_to_sequence adjustment*" and "*choice_to_sequence adjustment*" could be defined. We omit them due to the limitation of paper space. Because proposed structural adjustments act on two sound sequential sub WF-net, clearly obtained new WF-net is sound. Hence we have the conclusion:

Proposition 3. *Structural adjustment operations can preserve soundness of a sound WF-net.*

6. Live Instance Migration

On the set of change operations preserving soundness, we discuss further the approach to live instance migration in composite service evolution. There are two serious challenges to live instance migration: the first is how to determine whether a state migration is allowed or not, and the next is how to compute the state after the instance migration. Research[8] given a fundamental definition for valid migration. Unfortunately, it is very complicated to determine if an instance state can be migrated. And all current works have not offered a method of computing exactly the state in new composite service instance. The difference between this paper and the other works is that our approach do not directly use WF-net to determine if a state migration is allowed or not. First the evolved composite service definition is transferred to a reachability graph. Then we use the reachability graph and transition sequence to determine if a state migration is allowed or not, and to compute the state after the migration. Here we give a definition about valid migration.

Definition 6 (Valid migration)*: Let $WFN^O=(P^O, T^O, F^O, i^O, o^O, M_0^O)$ and $WFN^N=(P^N, T^N, F^N, i^N, o^N, M_0^N)$ be two sound WF-net and M be the reachable state of WFN^O through the transition sequence seq^O. A migration from state M^O to state M^N of WFN^N is valid iff:*

- *M^N is a reachable state in WFN^N;*
- *There exists a transition sequence seq^N from the initial state M_0^N to M^N such that $[seq^O]\backslash(T^O\backslash T^N)=[seq^N] \backslash(T^N\backslash T^O)$, where [seq] is the set of all transitions in sequence seq; ($T^O\backslash T^N$ and $T^N\backslash T^O$ present deleted and added transitions during evolution separately)*
- *For every transition sequence seq^N of WFN^N from M^N to final state M_{end}^N, seq^N do not contain any transitions in seq^O, i.e., WFN^N do not repeat any transitions in seq^O.*

The first condition in Definition 6 requires the new state should be reachable and terminable in the new instance. And the second condition ensures that any necessary transitions in the old instance will be performed after migration. Finally, the last condition indicates that any executed transitions in the old instance will not be repeated after migration.

Now, we convert a sound WF-net to a *reachability graph* $G=(V,E)$, where each vertex represents a state and each edge with weight means a transition. Ye et al.[17] present an effective algorithm to construct a reachability graph of PetriNet. In this way, any live states of a composite service instance can be determined clearly and exactly. In addition, we can find a set of all possible transition sequences in a WF-net, as well as get easily the completed transition sequence which has been implemented by certain live instance.

***Definition** 7 The reachability graph of a WF-net WFN=(P, T, F, i, o, M_0) is a graph G=(V, E), where*

- *The node set V is the set of all states of WFN;*
- *There is a edge with weight t from node M to M' in E iff M [t>M' in WFN.*

After creating a reachability graph of a sound WF-net, we can easily obtain state sets and transition sequence set of the WF-net. The working procedure of LiveMig algorithm involves two steps including determining the validity of the live state migration, computing the corresponding exact state after the migration in the case of valid migration, otherwise executing a transition according to the old instance and re-entering the algorithm.

LiveMig Algorithm [Migrate a instance state from old process definition to the new one]

input: $WFN^O=(P^O, T^O, F^O, i^O, o^O, M_0{}^O)$, M^O, $ts^{completed}$
$WFN^N=(P^N, T^N, F^N, i^N, o^N, M_0{}^N)$
output: M^N

```
1 begin
2 G^N:=WFN2Graph(WFN^N)
3 TSSet^N:=TransitionSequence(G^N)
4 TSSet^N:= DELETE(TSSet^N, choice_add)
5 TSSet^N':= Φ
6 for each ts ∈ TSSet^N
7 {  ts':=ts-T^N\T^O
8   TSSet^N':= TSSet^N' ∪ts'  }
9 ts^completed:= ts^completed-T^O\T^N
10 k:=|ts^completed|
11 if ts^completed ∈ k· TSSet^N'
12 {  t:=LAST(ts^completed)
13   SubSet':=CONTAIN(TSSet^N', ts^completed)
14   SubSet:=EXTEND(SubSet')
15   ts^N:=SELECT(SubSet, t)
16   M^N:=POST(G^N, ts^N, t)
17   end  }
18 else
19 {  while M≠M_end do
20   {  SubSet':=COMPATIBLE(k· TSSet^N', ts^completed)
21     if SubSet'≠Φ
22     {  SubSet:=EXTEND(SubSet')
23       (ts^N, t):=SELECT(SubSet, ts^completed)
24       M^N:=POST(G^N, ts^N, ts_k)
25       end  }
26     (M, t^NEW):=NEXT(M)
27     ts^completed:= ts^completed +t^NEW
28     k:=k+1  }  }
29 end
```

LiveMig Algorithm shows a concrete process for instance state migration. Input of the algorithm is an old WF-net (WFN^O) before evolution and a new WF-net (WFN^N) after evolution, as well as the completed transition sequence $ts^{completed}$ under state M^O. Output of the algorithm is a state M^N of a new WF-net (WFN^N) which can be migrated to. First of all, a reachability graph of new WF-net is created and a available transition sequence set $TSSet^N$ of new workflow net are found out with Breadth-First-Search based on reachability graph (see lines 2-3). This transition sequence set is pretreated and transition sequences imported by choice_add operation are deleted (see line 4). Choice_add operation does not affect state migration, but that the operation and structure adjustments are together used will result in potential bug. We will explain this problem by an example in following case study. The newly added transitions in $TSSet^N$ during evolution are deleted to get the new transition sequence set $TSSet^N$'(see lines 5-8). At the same time, the deleted transitions in $ts^{completed}$ during evolution are deleted to find length K of $ts^{completed}$ (see lines 9 and 10). When $ts^{completed}$ belongs to a subset consisting of first k transitions in each transition sequence in $TSSet^N$', a state migration can be allowed. This shows process structure adjustments (if any) have not affected instance migration. (see line 11) Then last transition t in $ts^{complete}$ is found and subset(*SubSet'*) containing all $ts^{completed}$ in transition sequence set $TSSet^N$' is found. Newly added transitions is added to *SubSet'* to get a subset (*SubSet*) in actual new WF-net. Last, a transition sequence ts^N containing t is worked out in *SubSet*. Comparing with other transition sequences in *SubSet*, position of t in ts^N is leftmost. According to the reachability graph G^N of the new WF-net, the state represented by direct successive node of the edge with weight t in ts^N sequence is M^N which we want to find. Here, the algorithm is terminated (see lines 12-17). Otherwise, if $ts^{completed}$ do not belong to a subset consisting of first k transitions in each transition sequence in $TSSet^N$', this means the process structure has been adjusted to result in that current state M can not be validly migrated (see line 18). Here we judge if M equal to final state M_{end}, and if not, a loop is entered (see line 19). In the loop, the transition sequence subset(*SubSet'*) in $TSSet^N$' is found in which first k transitions in each transition sequence is adjusted at random to equal to $ts^{completed}$(see line 20). If *SubSet'* do not equal empty set, newly added transitions is added to *SubSet'* to get a subset (*SubSet*) in actual new WF-net. The shortest transition sequence ts^N compatible with $ts^{completed}$ and the last transition of ts^N is choosed from *SubSet*. According to the reachability graph G^N of the new WF-net, the state represented by direct successive node of the edge with weight for t in ts^N sequence is M^N which we want to find. Here, the algorithm is terminated (see lines 21-25). If *SubSet'* equals empty set, this shows M^O state can not be migrated. Here, a significant transition t^{NEW} only can be kept on executing according to old instance to get new M^O. then a loop is entered again (see line 26-28). The size of reachability graph of WF-net is exponential on the size of WF-net in the worst-case time, so the worst-case complexity of LiveMig Algorithm is exponential.

Theorem: *LiveMig Algorithm is terminable and correct, that is, it can return a valid migration.*

Proof Sketch: We give only sketch of proof due to the space limit. LiveMig Algorithm is terminable obviously. Suppose that an old state M^O via the transition sequence seq^O in WFN^O is migrated to new state M^N in WFN^N. Since LiveMig Algorithm is used on the reachability graph of WF-net, therefore M^N must be a reachable state of WFN^N. In addition, LiveMig Algorithm first selects a transition sequence seq^N such that $[seq^O]\backslash(T^O\backslash T^N)=[seq^N]\backslash(T^N\backslash T^O)$ and M^N is the state which WFN^N reach through seq^N. Lastly, for every transition sequence seq^N of WFN^N from M^N to final state M^N_{end}, seq^N do not contain any transitions in seq^O because we assume that all *WFN has no* cyclical structures and a component service (transition) appears only once in a composite service.

7. Case Study and Experiments

In this section, we analyze the scenario in section 3 to explain feasibility and correctness of LiveMig. First of all, we can see that an original order-processing composite service (Figure 1a) is evolved by *choice_add* (adding a new PayPal service t_6), *parallel_add* (adding a check service for goods t_7) and *parallel_to_sequence* adjustment (changing original parallel structure of payment and assemble to assemble after payment), finally to get a new order-processing composite service (Figure 1d). Then, we can get two reachability graphs of old and new composite service (Figure 5a and Figure 5b).

Provided that the current old instance has implemented Register, Order and Pay services, the completed transition sequence is $t_1t_2t_3$ and the current state is M_3. According to LiveMig Algorithm, the transition sequence set $\{t_1t_2t_3t_4t_7t_5, t_1t_2t_6t_4t_7t_5\}$ of the new composite service is pretreated to delete the newly added transition t_7 and the transition sequence $t_1t_2t_6t_4t_7t_5$ containing t_6 imported by choice_add operation and to get the transition sequence set $\{t_1t_2t_3t_4t_5\}$. Obviously, the $t_1t_2t_3$ is contained in the $t_1t_2t_3t_4t_5$, so the new state is t_3's direct successive state M_3' and this result satisfies completely valid migration definition. Under another condition, when the completed transition sequence is $t_1t_2t_4$ and the current state is M_4, the $t_1t_2t_4$ is not contained in the $t_1t_2t_3t_4t_5$ or is not compatible with the one. Thus the old instance can be only made to keep on executing a significant transition to get a new transition sequence $t_1t_2t_4t_3$ compatible with $t_1t_2t_3t_4t_5$. Therefore the new state is t_4's direct successive state M_4' and this migration is also valid. Here we will further explain actual significance about deleting the transition sequence containing a transition imported by choice_add operation in the algorithm. Provided that the completed transition sequence is $t_1t_2t_4$ and the state is M_4, if the transition sequence containing a transition imported by selection addition operation is not deleted, the transition sequence set is $\{t_1t_2t_3t_4t_5, t_1t_2t_4t_5\}$. We can easily see that the $t_1t_2t_4$ is contianed in the $t_1t_2t_4t_5$, so the new state after migration is M_4'. This will result in that payment service t_3 or t_6 can not be executed and do not accord with valid migration definition.

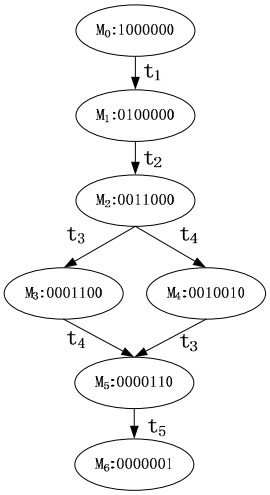

Figure 5a. Reachability graph of old composite service

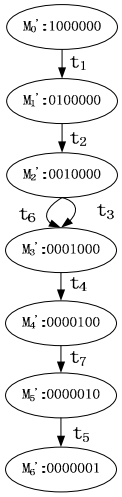

Figure 5b. Reachability graph of new composite service

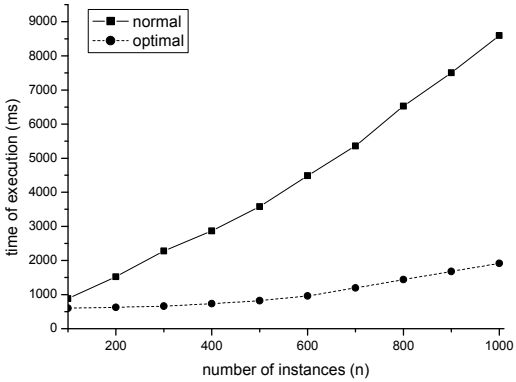

Figure 6a. Time VS. number of instances

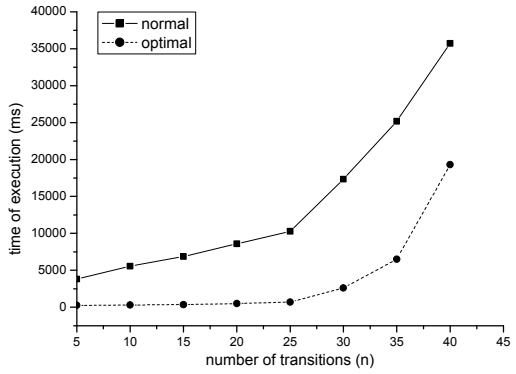

Figure 6b. Time VS. number of transitions

In addition, in order to evaluate further actual effectiveness of LiveMig, we have made simulation experiments. A simulation program is developed with JDK1.5 and runs in a PC (Intel core 2 1.86G CPU, 1G memory). All the experiments are repeated 10 times, and we report the average results. Simulation experiment 1 explains the relationship between amount of composite service live instance and migration time (solid line shown in Figure 6a). In the experiment, old composite service definition containing 20 component services is a sound and is created at random using set of building blocks (Chapter 4.3.3 in book [11]). Based on the proposed set of change operations, a new definition of composite service is obtained after implementing 10 changes at random. We vary the number of concurrent live instances from 100 to 1000 to get various results respectively. Simulation experiment 2 shows the relationship between size of composite service and migration time (solid lines shown in Figure 6b). In the experiment, currently there are 1000 live instances in the old component service definition. A new composite service definition is obtained after implementing 10 changes at random. We vary the number of the component services in old composite service definition from 5 to 40 (created at random) to get various results respectively. Our LiveMig just aims at migration of one live instance. When a large number of instances is simultaneously migrated, the algorithm can be appropriately optimized in two ways. On the one hand the reachability graph of evolved composite service may be created one time; on the other hand a lot of live instances at same state is not necessary to be computed repeatedly. After optimization, we have renewedly made experiment to get dasheds in Figure 6a and Figure 6b. It can be seen that performance of instances migration has been obviously improved.

8. Conclusion

In this paper, we introduce LiveMig, an approach to live instance migration in composite service evolution. First, we give a set of change operations preserving soundness and prove that the soundness of new composite services after evolution can be satisfied after using change operations in the basic operation set, which helps to avoid complex verifying process. Second, a live instance state migration algorithm is proposed to determine if a state migration is allowed or not, and to compute the exact state after the migration. Finally, the correctness of LiveMig is theoretically proved, and an extensive set of simulations are performed to show its feasibility and effectiveness. Our future work includes QoS-aware evolution process control and multi-version management.

Acknowledgment: This work was supported by the National High Technology Research and Development Program of China (863 program) under grant 2007AA010301, 2006AA01A106 and 2009AA01Z419. We would like to thank Yipeng Ji for the experiments in this paper.

References

[1] Zhang, L.-J., J. Zhang, and H. Cai, *Services Computing*. 2007: Beijing : Tsinghua University Press.

[2] Fu-Qing, Y., *Thinking on the Development of Software Engineering Technology.* Journal of Software, 2005. **16**(1): p. 1-7.

[3] Aalst, W.M.P.v.d. and S. Jablonski, *Dealing with workflow change: identification of issues and solutions.* International Journal of Computer Systems Science & Engineering, september 2000. **15**(5): p. 267-276.

[4] Papazoglou, M.P. *The Challenges of Service Evolution*. in *The 20th International Conference on Advanced Information Systems Engineering*. 2008.

[5] Andrikopoulos, V., S. Benbernou, and M.P. Papazoglou. *Managing the Evolution of Service Specifications*. in *The 20th International Conference on Advanced Information Systems Engineering*. 2008.

[6] RYU, S.H., et al., *Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures.* ACM Transactions on the Web, April 2008. **2**(2).

[7] Ellis, C. and G. Rozenberg. *Dynamic Change Within Workflow Systems*. in *Proceeding of the ACM Conference on Organizational Computing Systems (SIGOIS)*. 1995.

[8] Aalst, W.M.P.v.d., *Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change.* Information Systems Frontiers, 2001. **3**(3): p. 297-317.

[9] Aalst, W.M.P.v.d. and T. Basten, *Inheritance of workflows: an approach to tackling problems related to change.* Theoretical Computer Science, 2002: p. 125-203.

[10] Sun, P., C.-j. Jiang, and X.-m. Li, *Workflow Process Analysis Responding to Structural Changes.* Journal of System Simulation, Apr., 2008. **20**(7): p. 1856-1863.

[11] Aalst, W.v.d. and K.v. Hee, *Workflow Management Models, Methods, and Systems*. 2002, Massachusetts London, England: The MIT Press Cambridge.

[12] Aalst, W.M.P.v.d., et al. *Adaptive workflow On the interplay between flexibility and support*. in *Proceedings of the First International Conference on Enterprise Information Systems*. 2002.

[13] Rinderle, S., M. Reichert, and P. Dadam, *Correctness criteria for dynamic changes in workflow systems——a survey.* Data & Knowledge Engineering, 2004. **50**: p. 9-34.

[14] Rinderle-Ma, S., M. Reichert, and B. Weber. *Relaxed Compliance Notions in Adaptive Process Management Systems*. in *27th International Conference on Conceptual Modeling (ER)*. 2008.

[15] Cheng, A., J. Esparza, and J. Palsberg, *Complexity Results for 1-safe Nets.* Foundations of Software Technology and Theoretical computer Science, 1993. **761**: p. 326-337.

[16] Aalst, W.M.P.v.d., *Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques*, in *Business Process Management: Models, Techniques, and Empirical Studies*. 2000, Springer-Verlag. p. 161-183.

[17] Ye, X., J. Zhou, and X. Song, *On reachability graphs of Petri nets.* Computers & Electrical Engineering, 2003. **29**(2): p. 263-272.