

Failure Recovery Dependency Analysis for Web Composition Transactions Based on Extended Petri-Nets

Mei Xiaoyong^{1,2}, Huang Changqin^{1,3}, Tang Yong³, Zhao Gansen³ and Wang Huijin³

1. College of Educational Information and Technology, South China Normal University, Guangzhou 510631, China
2. School of Computer Science and Technology, Hunan University of Arts and Science, Changde 415000, China
3. Research Center for Information Services and Software Technology, South China Normal University, Guangzhou 510631, China
cdmxy@126.com

Abstract

For transactional Composition Web Services (CWS), traditional transaction theory in the database is introduced into the Long Running Transactions (LRTs) domain. In practical application, the typical LRTs employ forward or backward rollback. In order to improve the expansibility of LRTs, it is important to accurately specify the dependency between tasks in composition transaction; contemporary technologies usually statically specify dependency point and avoid implicit interaction in LRTs. In this paper, we propose a task dependency modeling method for LRTs, firstly we present formal description of atomic Web Services (WS) and discuss atomic WS such as pivot, compensable, retrievable and vital, then execution logs of LRTs data dependency and action dependency rules between tasks in LRTs are analyzed, specify, finally application of TRP shows that it is feasible to ensure consistent execution of reliable LRTs based dependencies.

Keywords: Transactional Composition Web Services, Task Dependency, Compensation, Execution log

1. Introduction

CWS composed a set of heterogeneous, autonomous and loosely coupled WS to offer new value-added services. Since the long-lived nature of CWS, it brings difficulty to transaction handling. Therefore, it needs to improve the failure handling capability to tolerate the occurrence of failure and finally coordinate it to a consistent acceptable state. In a loosely coupled LRTs environment, it is inevitable to take more relax transaction mechanism, which is called relaxed-ACID.

When using LRTs to composite WS, LRTs define compensation strategies and enclosed tasks in LRT will be compensated one by one from the innermost level until it terminates successfully. N. B. Lakhali and J. E. Haddad *et al.* discuss task dependency between nested layers in composition transaction, the failure in lower layer may propagate to a higher layer [1-2]. Since there are data dependency between composition transactions, failure of a task may lead to the inconsistency of other tasks. T.W. Chen and J. Miao discuss execution dependencies and compensation dependencies among tasks in business processes [3]. R.T. Khachana, A. Jamesa *et al.* introduce a new model for supporting transactions in a web environment [4]. More advanced data dependency approaches rely on comparing structured data such as parameters types and names [5]. This is generally not enough to identify the dependencies between tasks in terms of

functionality, and consequently makes it difficult, or even impossible, to use these methods to automate LRTs processing. L. Garcia-Banuelos proposes a executable transaction model based on ASML that allows seamless add/modify behavior, and extends failure handler in WS-BPEL [6-7]. R. Hamadi and B. Benatallan propose a Self-Adaptation Recovery Net (SARN) based on extended Petri nets model to specify fault or exceptional behavior in business process [7-8]. Due to the complexity of these kinds of transaction process, multiple dependencies such as temporal dependency, data flow dependency, control flow dependency, state dependency and behavior dependency often need to be constructed simultaneously to execute the transaction process in composition transaction environments. Hence the dependency problem between tasks in LRTs must be considered, which affects the effectiveness and efficiency of failure compensation of composition transaction.

2. Composition Transaction Dependency

How to ensure reliable and accurate execution of composition transaction is a important problem in transaction process. Based on the analysis of transaction execution logs, transaction dependency rules are constructed and misbehaviors that may cause unexpected results are detected, which handle failure, and ensure reliable and consistent execution of transaction.

2.1 Data Flow Dependency

The dependency of Web transaction relies on the control flow and the data flow of aggregation pattern. During the execution of composition business process, the data flow always follows the control flow. The explicit and implicit interaction in a data flow may violate atomicity of LRTs, explicit interaction in LRTs usually means the output of the previous task is the input of the current task and there is obvious partial order between two tasks, dependency rules are described as follows:

Rule 1

$(before(I_j, I_i) \vee contain(I_j, I_i) \vee meets(I_j, I_i) \vee starts(I_j, I_i)) \wedge I_i.precond(I_j.Activate()) \wedge (I_j.out \cap I_i.in \neq \emptyset) \Rightarrow I_i \xrightarrow{exd} I_j$.

Task I_i explicitly depends on I_j .

Rule 2 $I_i \xrightarrow{exd} I_k, I_k \xrightarrow{exd} I_j \Rightarrow I_i \xrightarrow{inexd}_{I_k} I_j$.

Task I_i explicitly depends on I_j indirectly. More generally, $I_i \xrightarrow{exd} I_1, I_1 \xrightarrow{exd} I_2, \dots, I_k \xrightarrow{exd} I_j \Rightarrow I_i \xrightarrow{inexd}_{\sigma} I_j$, where $\sigma = I_1 I_2 \dots I_k (k > 1)$.

Besides the mentioned explicit data dependency, there is implicit data interaction dependency between tasks of LRTs. For simplify analysis, *def-use* relationship [9-10] is introduced to locate this implicit data interaction. *def-use*(I_i, I_j) means, for two given tasks I_i and I_j , I_i is committed before I_j and I_j should use the data defined by I_i . The relaxed *def-use'*(I_i, I_j) means, for two given tasks I_i, I_j , I_i is committed before or together with I_j , and I_j should use the data defined by I_i .

Rule 3

$(equals(I_i, I_j) \vee contain(I_i, I_j) \vee starts(I_i, I_j) \vee finishes(I_i, I_j)) \wedge def-use'(t_i, t_j) \Rightarrow I_i \xrightarrow{imdl} I_j$.

Task I_i implicitly depends on I_j .

Rule 4 $I_i \xrightarrow{imdl} I_k, I_k \xrightarrow{imdl} I_j \Rightarrow I_i \xrightarrow{imdl}_{I_k} I_j$.

Task I_j implicitly depends on I_i indirectly. More generally, $I_i \xrightarrow{imdl} I_1, I_1 \xrightarrow{imdl} I_2, \dots, I_k \xrightarrow{imdl} I_j \Rightarrow I_i \xrightarrow{inimdl}_{\sigma} I_j$, where $\sigma = I_1 I_2 \dots I_k (k > 1)$.

2.2 Action Dependency

Rule 5 $I_i.commit() \wedge (before(I_i, I_j) \vee meets(I_i, I_j)) \rightarrow I_j.Activate() \Rightarrow I_j \xrightarrow{Act} I_i$.

Rule 6 $(I_i.Abort() \vee I_i.Fail() \vee I_i.Cancel()) \wedge ((before(I_i, I_j) \vee meets(I_i, I_j)) \wedge I_j.Activated) \rightarrow I_j.Abort() \Rightarrow I_j \xrightarrow{Abt} I_i$.

Rule 7 $(I_i.Fail() \vee I_i.Cancel()) \wedge (finishes(I_i, I_j) \vee equals(I_i, I_j)) \wedge I_j.Running \rightarrow I_j.Abort() \Rightarrow I_j \xrightarrow{Cnl} I_i$.

Rule 8 $I_i.Fail() \wedge (I_i \equiv I_i'') \rightarrow I_i''.Activate() \Rightarrow I_i'' \xrightarrow{Alt} I_i$.

$I_j'' \xrightarrow{Alt} I_j$ or $I_k'' \xrightarrow{Alt} I_k$ is satisfied in *AND-join*; $I_k \xrightarrow{Alt} I_k''$ is satisfied in *AND-split*.

According to the semantics of \xrightarrow{Abt} and \xrightarrow{Cnl} , \xrightarrow{Abt} should abort the tasks that haven't started and \xrightarrow{Cnl} should cancel the tasks under execution. Based on Rule 5- Rule 8, composition rules can be deduced as follows:

Rule 9 $I_i \xrightarrow{Cmt} I_j, I_j \xrightarrow{Cmt} I_k \Rightarrow I_i \xrightarrow{Cmt} I_k$;

Rule 10 $I_i \xrightarrow{Cmt} I_j, I_i \xrightarrow{Cmt} I_k \Rightarrow I_i \xrightarrow{Cmt} (I_j \wedge I_k)$;

Rule 11 $I_i \xrightarrow{Cmt} I_k, I_j \xrightarrow{Cmt} I_k \Rightarrow (I_i \vee I_j) \xrightarrow{Cmt} I_k$

Rule 12 $I_i \xrightarrow{Cmt} I_j, I_j \xrightarrow{Act} I_k \Rightarrow I_i \xrightarrow{Act} I_k$;

Rule 13 $I_i \xrightarrow{Act} I_j, I_i \xrightarrow{Act} I_k \Rightarrow I_i \xrightarrow{Act} (I_j \wedge I_k)$;

Rule 14 $I_i \xrightarrow{Cmt} I_j, (I_i \vee I_j) \xrightarrow{Act} I_k \Rightarrow (I_i \wedge I_j) \xrightarrow{Act} I_k$

Rule 15 $I_i \xrightarrow{Cmt} I_j, (I_j \wedge I_k) \xrightarrow{Abt} I_s \Rightarrow (I_i \wedge I_k) \xrightarrow{Abt} I_s$

Rule 16 $I_i \xrightarrow{Cmt} I_j, (I_j \vee I_k) \xrightarrow{Abt} I_s \Rightarrow (I_i \vee I_k) \xrightarrow{Abt} I_s$

Rule 17 $I_i \xrightarrow{Cnl} I_j, I_j \xrightarrow{Cnl} I_k \Rightarrow I_i \xrightarrow{Cnl} I_k$;

Rule 18 $I_i \xrightarrow{Cnl} I_j \Rightarrow (I_i \parallel I_1 \parallel I_2 \parallel \dots \parallel I_m) \xrightarrow{Cnl} I_j$;

If I_j sends $I_j.Fail()$ or $I_j.Compensated()$ to I_i , then there exist the following compensation rules between I_i and I_j :

Rule 19

$I_j \xrightarrow{exd} I_i \wedge (I_i.TBP \in \{Compensable, Vital\}) \wedge I_i.Committed \wedge (I_j.Fail() \vee I_j.Compensate()) \rightarrow I_i.Compensate() \Rightarrow I_i \xrightarrow{Cptexd} I_j$;

Rule 20

$(I_j \xrightarrow{inexd} I_i \vee I_j \xrightarrow{inexd} I_i) \wedge (I.TBP \in \{Vital, Compensable\}) \wedge I_i.Committed \wedge (I_j.Fail() \vee I_j.Compensate()) \rightarrow I_i.Compensate() \Rightarrow I_i \xrightarrow{Cptinexd} I_j$;

Rule 21

$(I_j \xrightarrow{inexd} I_i \vee I_j \xrightarrow{inexd} I_i) \wedge (I.TBP \in \{Vital, Compensable\}) \wedge I_i.Committed \wedge (I_j.Fail() \vee I_j.Compensate()) \rightarrow I_i.Compensate() \Rightarrow I_i \xrightarrow{Cptinmd} I_j$;

Rule 22

$I_j \xrightarrow{inimd} I_i \wedge I_j.TBP \in \{Compensable, Vital\} \wedge I_i.Committed \wedge (I_j.Fail() \vee I_j.Compensate()) \rightarrow I_i.Compensate() \Rightarrow I_i \xrightarrow{Cptinimd} I_j$;

Aggregation control patterns are used in LRTs, such as $\oplus, \sqcap, \otimes, \sqcup$, where \sqcap is composed of *And-Split* and *And-join*, \otimes is composed of *Or-split* and *Or-join*. Based on Rule 19- Rule 22, composition rules can be deduced as follows:

Rule 23 $I_i.TBP, I_j.TBP \in \{Compensable, Vital\} \wedge I_i \oplus I_j \Rightarrow I_j \xrightarrow{Cpt} I_i$.

Rule 24 $I_i.TBP, I_j.TBP, I_k.TBP \in \{Compensable, Vital\} \wedge I_i \oplus (I_j \sqcap I_k) \Rightarrow (I_j \sqcap I_k) \xrightarrow{Cpt} I_i$.

Rule 25 $I_i.TBP, I_j.TBP, I_k.TBP \in \{Compensable, Vital\} \wedge (I_i \sqcap I_j) \oplus I_k \Rightarrow (I_i \sqcap I_j) \xrightarrow{Cpt} I_k$.

Rule 26 $I_i.TBP, I_j.TBP, I_k.TBP \in \{Compensable, Vital\} \wedge I_i \oplus (I_j \otimes I_k) \Rightarrow (I_j \otimes I_k) \xrightarrow{Cpt} I_i$.

Rule 27 $I_i.TBP, I_j.TBP, I_k.TBP \in \{Compensable, Vital\} \wedge (I_i \otimes I_j) \oplus I_k \Rightarrow I_k \xrightarrow{Cpt} (I_i \otimes I_j)$.

Rule 28 $I_i \xrightarrow{Cpt} I_j, I_j \xrightarrow{Cpt} I_k \Rightarrow I_i \xrightarrow{Cpt} I_k$

Rule 29

$I_j \xrightarrow{Cpt} I_k, I_k \xrightarrow{Cpt} I_j, I_j \xrightarrow{Cnl} I_k, I_k \xrightarrow{Cnl} I_j, I_k \xrightarrow{Cpt} (I_i \sqcap I_j) \Rightarrow (I_j \sqcap I_k) \xrightarrow{Cpt} I_i$;

Rule 30

$I_i \xrightarrow{Cpt} I_j, I_j \xrightarrow{Cpt} I_i, I_i \xrightarrow{Cnl} I_j, I_j \xrightarrow{Cnl} I_i, I_i \xrightarrow{Abr} I_j, I_j \xrightarrow{Abr} I_i \Rightarrow I_k \xrightarrow{Cpt} (I_i \sqcap I_j)$;

Rule 10-Rule 22 can be deduced from Rule 1- Rule 9. Considerring the verification of flow structure of LRTs, if there exists *AND-Join*, but no *AND-Split* (or exists *OR-Join*, but no *OR-Split*), then there exist design defects in LRTs flow, deduced from Rule 9-Rule 30. Therefore, not all defects can be detected during execution according to rule deduction.

3. Generation and Test Algorithm of Dependency Rules for LRTs

For each task in LRTs, direct dependency between tasks is calculated and added to direct data dependency set DDs according to dependency rules. Then indirect data dependency set IDs is calculated between nonadjacent tasks according to DDs. Finally, DDs and IDs are merged together to calculate dependency set of LRTs. The generation dependency set algorithm is given as follows.

Algorithm 1 Generation algorithm of the set of compensation dependency

Input: task set T in business process BP;

Output: dependency set Ds

$DCD = \emptyset, ICD = \emptyset, CD = \emptyset$;

/* Calculate the direct data dependency set DDs */

for each $I_i \in T$

{ if $I_i \oplus I_j$ then

$DCD = DCD \cup \{I_i\} \xrightarrow{Cpt} I_i$;

if $I_i \oplus (I_m \parallel \dots \parallel I_n)$ then

$DCD = DCD \cup \{(I_j \parallel I_k)\} \xrightarrow{Cpt} I_i$;

if $(I_i \parallel I_j) \oplus I_k$ then

$DCD = DCD \cup \{I_k\} \xrightarrow{Cpt} (I_i \parallel I_j)$;

if $I_i \oplus (I_j \otimes I_k)$ then

$DCD \leftarrow DCD \cup \{(I_j \otimes I_k)\} \xrightarrow{Cpt} I_i$;

if $I_i \in$

$DCD \leftarrow DCD \cup \{I_k\} \xrightarrow{Cpt} (I_j \otimes I_i)$;

/* Calculate the set of indirect dependency IDs*/

for each $\{(I_m \parallel \dots \parallel I_n)\} \xrightarrow{Cpt} I_i \in DCD$

for each $I_j \in \{I_m, \dots, I_n\}$

for each $I_k \in \{I_m, \dots, I_n\}$

if $\{I_k\} \xrightarrow{Cpt} I_j \notin DCD$ then

$ICD = ICD \cup \{I_k\} \xrightarrow{Cpt} I_j \cup \{I_k\} \xrightarrow{Cnl} I_j$;

for each $\{I_i\} \xrightarrow{Cpt} (I_m \parallel \dots \parallel I_n) \in DCD$

for each $I_j \in \{I_m, \dots, I_n\}$

for each $I_k \in \{I_m, \dots, I_n\}$

if $\{I_k\} \xrightarrow{Cpt} I_j \notin DCD$ then

$ICD = ICD \cup \{I_k\} \xrightarrow{Cpt} I_j \cup \{I_k\} \xrightarrow{Cnl} I_j \cup \{I_k\} \xrightarrow{Abr} I_j$;

$CD = DCD \cup ICD$;

for each $\{I_j\} \xrightarrow{Cpt} I_i \in CD$

for each $\{I_k\} \xrightarrow{Cpt} I_j \in CD$

$CD \leftarrow CD \cup \{I_k\} \xrightarrow{Cpt} I_i$;

return CD

For given LRTs, if $|T| = n$, time complexity of generating compensation dependency set of adjacent tasks is $O(n^2)$, while for nonadjacent tasks it is $O(n^3)$, therefore, time complexity is $O(n^3)$.

DDG (Data Dependency Graph) and TDG (Task Dependency Graph) can be generated according to the exectuion logs and task dependencies in LRTs. For $x, y \in \text{Logitem}$ in DDG, there exist I_i such that $x \xrightarrow{I_i} y$; for $I_i, I_j \in \text{TDG}$, there exist x such that $I_i \xrightarrow{x} I_j$.

Weighted graph is an intuitive way to represent the dependency degree between tasks, complete dependency and partial dependency between tasks are represented in Figure 1 and Figure 2 respectively, where node denotes task, edge denotes dependency, and weight denotes dependency degree. If there exists $I_j \rightarrow I_i (I_j^{\alpha} I_i)$, then dependency degree of I_i and I_j is $\alpha = (n - ne) / n$, where n and ne are the number of dependency instance and nondependency instance respectively. Then we can get a $n \times n$ TDM (Task Dependency Matrix), where each element α_{ij} denotes dependency degree of I_i and I_j . For example, $I_i.out(par_1, par_2, \dots, par_m)$ and $I_j.in(par_1, par_2, \dots, par_k)$, if $n = m + k = 11$, $ne = 3$, then $\alpha = (11 - 3) / 11 = 72.7\%$, denoted as $I_j^{0.727} I_i$.

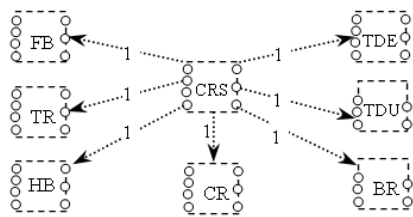


Figure 1. Complete Dependency

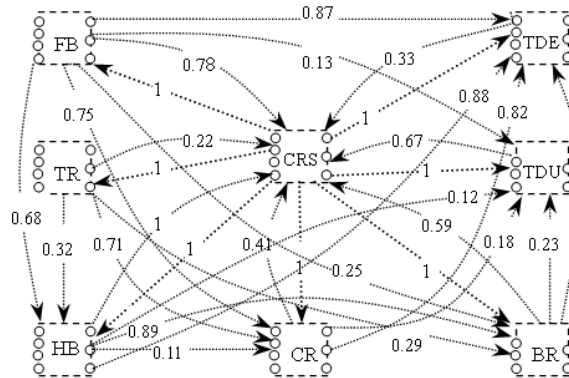


Figure 2. Partial Dependency

As shown in Figure 1 and Figure 2, it is obvious that partial dependency provide more information. For example, in Figure 2, there is a directed edge form FB to HB weighted 0.68, we call FB is the dependency point of HB, HB is the decision point of FB, and 0.68 is the partial dependency degree.

Algorithm 2 Generation algorithm of DDG and TDG

```

Input: Logfile and Ds
Output: DDG and TDG
{For each data x ∈ Ds /*datas that affectes transaction*/
  For each datat y ∈ Ii.data /*data item of transaction*/
    {If x  $\xrightarrow{I_i}$  y /*if there exist dependency*/
      InsertDDG(x,y); /*insert to DDG*/
      If  $I_i \xrightarrow{x} I_j$  /* if there exist dependency*/
        InsertTDG(Ii,Ij); /*insert to TDG*/
    }
}

```

It is very important to verify task dependency in LRTs. If all task dependencies in LRTs satisfy TD_1, TD_2, \dots, TD_n , then $I_1 \rightarrow I_n$, that is, there exist at least one rule set in rule base such that LRTs execute from I_1 to I_n . Henry *et al.* verify the logical expression correctness of this criterion with propositional logic [11]. This paper presents the verification algorithm of this criterion, which firstly verify basic dependency with Rule 5-Rule 9; then verify composition dependency with Rule 10-Rule 30; finally if execution of $I_1 \rightarrow I_n$ can be verified, then the execution of LRTs is correct.

Algorithm 3 Verify task dependency in LRTs.

```

Input: BP, TDG, DDG.
Output: Correctness of dependency.
Activated = {all the tasks whose state is Activated};
Committed = {all the tasks whose state is committed};
Aborted = {all the tasks whose state is aborted};
Cancelled = {all the tasks whose state is cancelled};
Compensated = {all the tasks whose state is compensated};
Failed = {all the tasks whose state is failed};
ExecutionTask = ∅; ActivationTask = ∅; CommittedTask = ∅;
AbortionTask = ∅; CancellationTask = ∅;
for each task I ∈ BP{
  for each task Ii.Activated {

```

```

if (li.Activate() is triggered) then
  if(IsVerifyRule({R6,R12,R13,R14}) is true)
    /*ExecutionTask means li verified through {R6,R12,R13,R14}*/
    { ExecutionTask=ExecutionTask ∪ {li};
      ActivationTask=ActivationTask ∪ {li};
    }
  For each rule r ∈ {R6,R12,R13,R14}
  if isLeft(r)=true ∧ isRight(r)=true then
    TDAct= TDAct ∪ Left(r)  $\xrightarrow{Abr}$  Right(r) ;
for each task li.Committed {
  if (li.Commit() is triggered) then
  if(IsVerifyRule({R9,R10,R11}) is true) then
    /*ExecutionTask means li verified through {R9,R10,R11}*/
    { ExecutionTask=ExecutionTask ∪ {li};
      CommitionTask=CommitionTask ∪ {li};
    }
  For each rule r ∈ {R9,R10,R11}
  if isLeft(r)=true ∧ isRight(r)=true then
    TDCmt= TDCmt ∪ Left(r)  $\xrightarrow{Cmt}$  Right(r) ;
for each task li.Failed {
  if (li.Abort() is triggered and li.FailureType=Attribute(Out(li)).Abortion) then
  if(IsVerifyRule({R6,R15,R16}) is true) then
    /*ExecutionTask means li verified through {R6,R15,R16}*/
    { ExecutionTask= ExecutionTask ∪ {li};
      AbortionTask=AbortionTask ∪ {li};
    }
  For each rule r ∈ {R6,R15,R16}
  if isLeft(r)=true ∧ isRight(r)=true then
    TDAbt= TDAbt ∪ Left(r)  $\xrightarrow{Abt}$  Right(r) ;
  if (li.Cancel() is triggered and li.FailureType=Attribute(Out(li)).Cancellation) then
  if(IsVerifyRule({R7,R17,R18}) is true) then
    /*ExecutionTask means li verified through {R7,R17,R18}*/
    { ExecutionTask= ExecutionTask ∪ {li};
      CancellationTask=CancellationTask ∪ {li};
    }
  For each rule r ∈ {R7,R17,R18}
  if isLeft(r)=true ∧ isRight(r)=true then
    TDCnl= TDCnl ∪ Left(r)  $\xrightarrow{Cnl}$  Right(r) ;
  if (li.Compensate() is triggered and li.FailureType=tribute(Out(li)).Compensation) then
  if(IsVerifyRule({R19,R20,R21,R22,R23,R24,R25,R26,R27,R28,R29,R30}) is true) then
    /*ExecutionTask means li verified through {R19,R20,R21,R22,R23,R24, R25,R26,R27,R28,R29,R30}*/
    { ExecutionTask= ExecutionTask ∪ {li};
      CompensationTask=CompensationTask ∪ {li};
    }
  For each rule r ∈ {R19,R20,R21,R22,R23,R24,R25,R26,R27,R28,R29,R30}
  if isLeft(r)=true ∧ isRight(r)=true then
    TDCpt= TDCpt ∪ Left(r)  $\xrightarrow{Cpt}$  Right(r) ;
}}
for each task li ∈ ExecutionTask {
  if (li.state=Activated and TDAct ≠ ∅ ) or (li.state=Committed and TDCmt ≠ ∅ ) then
  if ({I1 → In}) == TDCmt then
    return successful
  else if li.state= Failed then
    if (li.state= Aborted and TDAbt ≠ ∅ ) or (li.state= Cancelled and TDCnl ≠ ∅ ) or (li.state= Compensated and TDCpt ≠ ∅ ) then
      return Compensationssuccessful
}

```

For given LRTs, if $|T| = n$ and $|TP| = m$, time complexity of verification algorithm is $O(n^3)$. If LRTs failed, dependency requirements of tasks are verified with TD_{Act}, TD_{Cmt}, TD_{Abt}, TD_{Cnl}, TD_{Cpt}.

4. Application of Composition Transaction

We implement this case with the open source software ActiveBPEL [12], the experiment is implemented on a testbed consisting of 2 IBM x 3650 servers and 12 PCs connected by a 100Mbps Ethernet. Each server is equipped with 4 Intel 2930MHz processors and 1GB memory running SuSE 10.2 Linux. Each PCs is equipped with Intel Pentium Dual Core E5200 2.5 GHz, 2GB RAM, Windows XP SP2. the process execution engine construct execution log to help reconstruct compensation context when binding compensation service, so reverse compensation flow is constructed, as shown in Fig. 3. $t_1^f, t_2^f, \dots, t_{10}^f$ is introduced to construct the

mapping between I_1, I_2, \dots, I_{10} and their corresponding compensation tasks $I'_1, I'_2, \dots, I'_{10}$. In practical applications, it needs to consider compensation strategy. During the execution of TRP, the failure of task will trigger CH to abort the forward flow and turn to the backward compensation flow. Based on the compensation pair, the extended TRP is as follows: $(CRS \div CRS') \oplus (((FB \div FB') \otimes TR) \parallel (HB \div HB') \parallel ((CR \div CR') \otimes (BR \div BR'))) \oplus (OP \div OP') \oplus ((TDE \div TDE') \otimes (TDU \div TDU')) \oplus TC$.

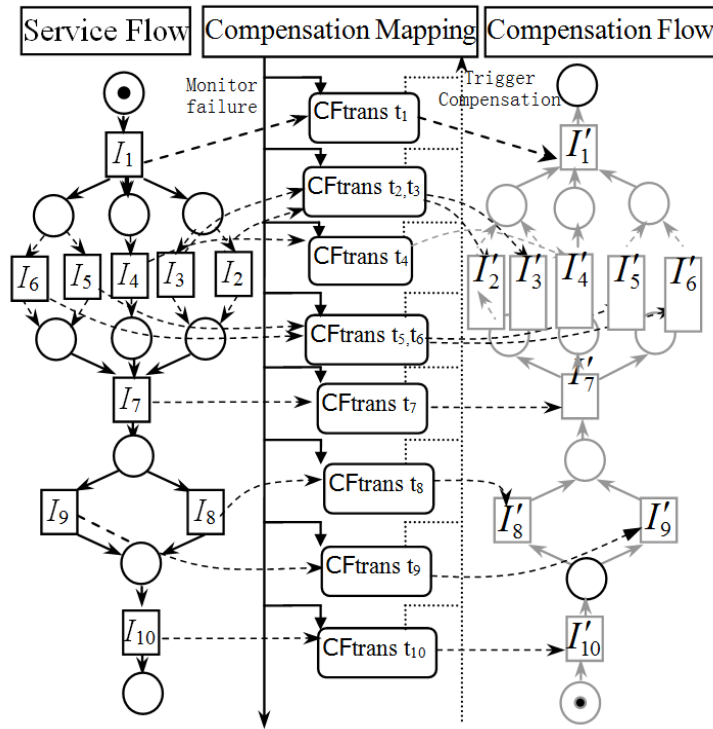


Figure 3. Multi-partner Cooperation Traveling Reservation Composition Transactions

To show the effectiveness of the proposed approach, we performed an experimental evaluation with a dependency-based WSC. Since the approach employed a generation algorithm of compensation dependency, it is feasible to execute it in a recovery environment. The result from the experiments that shows dependency rules based on recovery and the execution of TRP, as shown in Figure 4. The dependency-based recovery approach locates any conforming recovery service with the dependency rules and performs dependency reasoning. From these results, it is evident that the proposed approach is efficient and scalable than a nondependency-based approach which does not incorporate a facility to handle dependency.

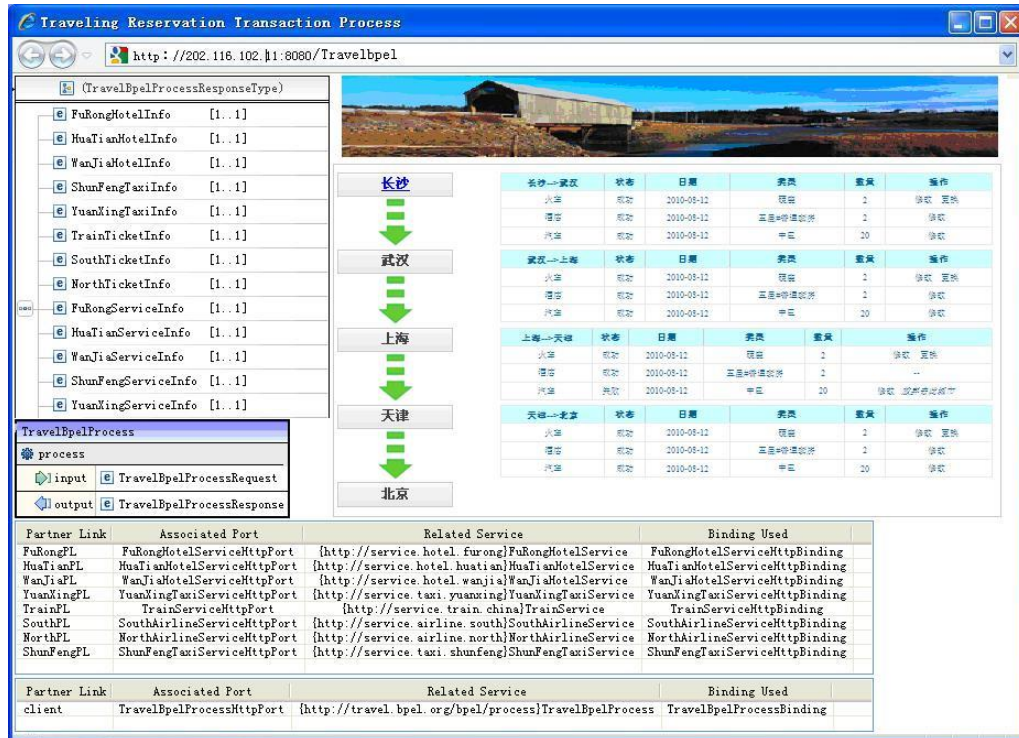


Figure 4. Execution Result of TRP

5. Conclusion

Current failure compensation strategies of composition transaction seldom take into consideration these dependencies. Therefore, in this paper, we propose a task dependency method based on extended Petri nets using transaction execution log, firstly we formalize atomic WS based on Petri nets and discuss atomic WS, then we analyze execution logs of LRTs, specify dataflow dependency and action dependency between tasks in LRTs, application of TRP based on LRTs shows that it is feasible to ensure reliable execution of LRTs and verify dependency of TRP.

Acknowledgements

This work is one of the projects supported by the National Key Technologies R&D Program of China (2013BAH72B01), the National Natural Science Foundation of China (60940033, 61370229), the Postdoctoral Science Foundation of China (20080440121), the Natural Science Foundation of Province (06017089, 60940033), the Science and Technology Planning Project of Hunan Province (2010GK3020), Youth Foundation of Educational Commission of Hubei Province of China (12B092).

References

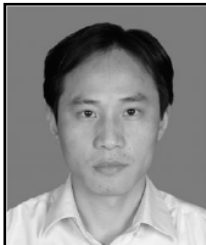
- [1] N. B. Lakhali, T. Kobayashi and H. Yokota, "FENECIA: failure endurable nested-transaction based execution of composite Web services with incorporated state analysis", International Journal on Very Large Data Bases, vol.18, no.1, (2009), pp. 1-56.
- [2] J. E. Haddad, M. Manouvrier and M. Rukoz, "TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition", IEEE Transactions on Services Computing, vol. 3, no. 1, (2010), pp. 73-85.
- [3] T. W. Chen and J. Miao, "Research on Exception Handling Mechanism Based on Directed Graph in Service Composition", Applied Mechanics and Materials, vol. 58, (2011), pp. 1088-1093.

- [4] R. T. Khachana, A. Jamesa and R. Iqbala, "Relaxation of ACID properties in AuTrA: The adaptive user-defined transaction relaxing approach", *Future Generation Computer Systems*, vol. 27, no. 1, (2011), pp. 58-66.
- [5] J. Ma, Y. Zhang and J. He, "Web Services Discovery Based on Latent Semantic Approach", *Proceedings of the 2008 IEEE International Conference on Web Services*, (2008), pp. 740-747.
- [6] L. Garcia-Banuelos, "An AsmL Executable Model for WS-BPEL with Orthogonal Transactional Behavior", *4th international conference Business process management, LNCS 4102*, (2006), pp. 401-406.
- [7] A. Varga, A. E. C. Basaveb, M. Rowe, F. Ciravegna and Y. Hed, "Linked knowledge sources for topic classification of microposts: A semantic graph-based approach", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 26, no. 4, (2014), pp. 36-57.
- [8] R. Hamadi, B. Benatallah and B. Medjahed, "Self-adapting recovery nets for policy-driven exception handling in business processes", *Distributed and Parallel Databases*, vol. 23, no. 1, (2008), pp. 1-44.
- [9] B. Medjahed and Z. Malik, "Bottom-Up Fault Management in Composite Web Services", *Lecture Notes in Computer Science*, vol. 6741, (2011), pp. 597-611.
- [10] [9] A. Tahir, D. Tosi and S. Morasca, "A systematic review on the functional testing of semantic web services", *Journal of Systems and Software*, vol. 86, no. 11, (2013) November, pp. 2877-2889.
- [11] F. E. Allen and J. Cocke, "A program data flow analysis procedure", *Communications of the ACM*, vol. 19, no. 3, (1976), pp. 137-146.
- [12] H. H. Bi and J. L. Zhao, "Applying propositional logic to workflow verification", *Information Technology and Management*, vol. 5, no. 324, (2004), pp. 293 -318.
- [13] Activebpel, http://www.activebpel.org/down_load.

Authors



Mei Xiaoyong, he was born in 1974, Ph.D. professor. His research interests include service-oriented computing, semantic Web and Petri-nets.



Huang Changqin, he was born in 1972, Ph.D. professor. His research interests include information technology and service-oriented computing.

