MaxInsTx: A Best-Effort Failure Recovery Approach for Artifact-Centric Business Processes

Haihuan Qin, Guosheng Kang, and Lipeng Guo

School of Computer Science, Fudan University, China
{09110240015,12110240015,10110240002}@fudan.edu.cn

Abstract. Process instances may overlap and interweave with each other. This significantly complicates the failure recovery issue. Most of existing mechanisms assume a one-to-one relationship between process instances, which will cause unnecessary recovery in such context. Artifact-centric business process models give equal consideration on both data and control flow of activities, thus facilitate addressing this issue. In this paper, we propose a best-effort failure recovery approach MaxInsTx: a transactional artifact-centric business process model with complex cardinality relationships and correlations considered; a recovery mechanism to resolve the impact of the failed process on concurrent processes meanwhile protect maximal instances involved in failures from failure impact.

1 Introduction

Business processes (BPs) have become a necessity for modern organizations to stay competitive. In real life, instances of one BP may be split or merged into instances of another BP due to business needs [1-3]. For example, an online shop, acting as a broker, may divide its orders into multiple purchase orders, one pur supplier, and merge items of (different) orders from the same supplier into one purchase order for bulk purchases, as shown in Fig.1. The cardinality between order process and purchase order process is many-to-many and correlations among their instances are complex. We call such scenarios relevant business scenarios. The fabric instances complicates problems and attracts increasing attention [1-3]. No works has been done on the recovery issue yet.

Most of existing workflow transactions simply assume a one-to-one relationship between BP instances (e.g., [4]), and resolve the impact of a failed BP on concurrent BPs through handling dirty reads and dirty writes [5]. Once a failed BP is recovered, all side effects of its committed tasks are semantically undone. However, in the context of relevant business scenario, such recovery is an "over-recovery" and no longer applicable.

Fig.2 illustrates a relevant business scenario for an online store represented by artifact-centric BP models. This modeling language is used since business data and runtime data gathered in artifacts facilitate modeling instance level correlation [3].

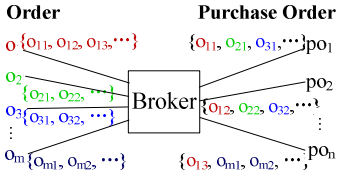
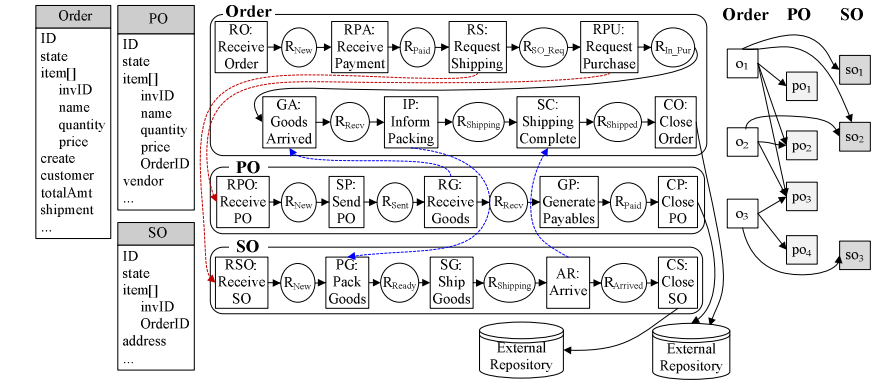


Fig. 1. A relevant business scenario

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 558–566, 2013.



(a) Information models (b) Example workflows of Order, PO and SO (c) Cardinality

Fig. 2. Workflows involved in the online store example and cardinality between them

Three (key) artifacts are involved: *Order*, *Purchase Order (PO)* and *Ship Order (SO)*. Fig. 2(a) shows their information models and Fig. 2(b) their workflows. Attributes of an artifact can be simple or relation-typed (denoted by "[]") with nested attributes. Once receiving a customer order, the store initiates an *Order*. After receiving payment, the *Order* create a *PO* by sending a message to *RPO*, and its task *GA* cannot be initiated until receives the message from *PO*, i.e., there exist creation and synchronization (sync for short) dependencies between instances of *Order* and *PO*. An *Order* may create multiple *POs*, one pur supplier, meanwhile multiple *Orders* may be merged into one *PO* for bulk purchases. Similar splitting and merge exist among *Orders* and *SOs*. The relationships between these artifacts are m-to-m, as shown in Fig. 2(c), and nested attribute *OrderID* contained in *PO* (or *SO*) indicates correlations among them.

Several issues are worthy of attention. First, creation dependencies cannot be covered by traditional transactions. For example, an *Order* o_1 creates a *PO* po_1, po_1 should be rolled back when o_1 is canceled, which is similar to dirty reads problem [5]. However, if po_1 is rolled back for items are out of stock, o_1 also needs to be recovered for it cannot be accomplished. Second, it is too costly to recover all dependent artifacts of a failed artifact when dependencies can be m-to-m. For example, if o_1 is canceled, po_1, po_2 and po_3 will be affected and recovered, however, the entire recovery of po_2 and po_3 will further cause other normal cases o_2 and o_3 to be affected and recovered. To avoid such over-recovery, the affected parts of po_2 and po_3 need to be identified, and fixed (i.e., partially rolled back) without affecting o_2 and o_3 as well as their purchases.

To solve above questions, we propose a best-effort failure recovery approach *MaxInsTx* and make two contributions: (1) we develop a transactional artifact-centric BP model with creation dependencies considered and cardinality types of dependencies distinguished; (2) we propose an approach to resolve the impact of the failed BP on concurrent BPs, avoiding unnecessary recoveries through overlap evaluation.

This paper is organized as follows. The transactional model and the failure recovery mechanism are presented in Section 2 and 3 (resp.). Section 4 discusses effectiveness and feasibility. Section 5 discusses related work and Section 6 concludes this paper.

2 A Transactional Model for Relevant Artifact-Centric BPs

2.1 Preliminaries

In EZFlow [6], a BP is represented by *artifact classes*, *tasks*, *repositories* and *workflow schemas*, with a core artifact carrying both data and the enactment. Each artifact class is a tuple A = (*name, Atts*), where *name* is its name, *Atts* is a set of attributes. Each class always contains the attribute *ID* to hold the unique identifier of its instances.

Definition 2.1: An *EZFlow* schema is a tuple (C, Γ, R, M, T, F, G), where

- C is the *core* artifact class, Γ is a set of *auxiliary* artifact classes; R is a set of repositories; M is a set of message types; T is a set of tasks;
- F maps each task t in T to a pair (m_i; M_o), where m_i is the message type that triggers t and M_o is the set of message types produced by t. Each message type is produced by at most one task and can be used to trigger at most one task;
- G is a set of triples (u, v, g) where (1) either $u \in R$ and $v \in T$ or $u \in T$ and $v \in R$; and (2) g is a guard on the edge (u, v).

EZFlow uses *snapshots* to represent the system state of a workflow at time instants, which contains tables for artifact classes, workflow enactments, etc. A workflow moves from one *snapshot* to another when an external message arrives or a triggered task completes its execution, evolving the artifact along its lifecycle. The enactment of a workflow is an alternating sequence of system states and tasks $s_0t_0s_1t_1..t_{n-1}s_n$ such that s_0 is the initial state and each s_{i+1} is derived from s_i through t_i ($0 \le i < n$).

2.2 Correlations and Dependencies of Artifacts

Correlations combine individual BPs into relevant business scenarios. We use a *correlation graph* to present correlations between artifacts. Correlation conditions are defined by projection operation π and intersection predicate $\sqcap$. The projection $\pi_{A_1,\ldots,A_n}(\alpha)$ restricts artifact α to its attributes set $\{A_1, \ldots, A_n\}$. The predicate $\sqcap$ checks whether two input sets have overlap. If overlap exists, the expression will be true. An atomic correlation condition is an intersection expression applied to projection expressions.

Definition 2.2: A *correlation graph* G is a tuple ($\mathcal{A}$, E, Φ, κ), where

- $\mathcal{A}$ is a nonempty set of artifacts closed under references (through attributes). Artifacts in $\mathcal{A}$ are called nodes of the graph;
- $E \subseteq \mathcal{A} \times \mathcal{A}$ is a set of edges which denotes correlations among artifacts;
- $\Phi: E \to Con$ is a mapping that assigns each edge a correlation condition, i.e., a set (conjunction) of atomic conditions; and
- $\kappa: E \to Card$ is a mapping from E to {1:1, 1:m, m:1, m:m} specifying cardinality relationship between artifacts of source node and end node of an edge.

Above definition adapts the correlation graph in [3] to support *m-to-m* cardinality relationship. Instances of two correlated artifacts are correlated if the correlation condition is true on these instances. For example, suppose correlation condition of *Order*

and PO is $con = \pi_{ID,item.invID}(Order) \sqcap \pi_{item.OrderID,item.invID}(PO)$, o_1 and po_1 are instances of *Order* and *PO*, resp., o_1 is correlated with po_1 if *con* is true on o_1 and po_1.

Only correlated artifacts may exchange messages. Thus, we define messages among artifacts with respect to correlation graph, using "ext" to denote external environment.

Definition 2.3: Given a correlation graph $G = (\mathcal{A}, E, \Phi, \kappa)$, a message type *msg* among artifacts wrt *G* is a tuple $(N, Atts, \alpha_s, \alpha_r, \vdash)$, where

- *N* is a distinct message name; *Atts* is a set of attributes of the message;
- α_s and α_r denotes the sender and receiver (resp.), satisfying at most one of them can be "ext", and if both are artifacts, they must be correlated;
- $\vdash$ indicates whether the sender creates an instance of the receiving artifact when a message instance arrives, "yes" means creation, "no" means no creation.

Correlated artifacts along with these *msgs* form a relevant business scenario.

Definition 2.4: A relevant artifact-centric BPs schema *RW* is a tuple *(G, Msg)*, where *G* is a correlation graph, and *Msg* is a set of message types wrt *G*.

Dependencies exist between two artifacts α_i and α_j if they are correlated. Basically, there are two categories of dependencies: creation dependency and sync dependency. α_j has creation dependency on α_i if α_i contributes to the creation of α_j (i.e., α_i sends a message whose $\vdash$ is "yes" to α_j). α_j has sync dependency on α_i if α_j cannot proceed until the message from α_i arrives (i.e., α_i sends a message whose $\vdash$ is "no" to α_j).

Instance level dependencies are created at runtime. Given instances $I_{\alpha i}$ of α_i and $I_{\alpha j}$ of α_j, we use notations $\alpha_j < I_{\alpha i} >$ and $\alpha_i \ll I_{\alpha j} \gg$ to denote the instances of α_j which are created by $I_{\alpha i}$ and the instances of α_i which contribute to the creation of $I_{\alpha j}$, resp; and $Rv(I_{\alpha i}, msg)$ and $Sd(I_{\alpha j}, msg)$ to denote the instances which receive the message *msg* sent from $I_{\alpha i}$ and the instances which send the message *msg* received by $I_{\alpha j}$, resp. Instance level dependencies are defined as follows.

Definition 2.5: Given artifacts α_i and α_j and their instances $I_{\alpha i}$ of α_i and $I_{\alpha j}$ of α_j, $I_{\alpha j}$ is creation dependent on $I_{\alpha i}$ if α_j is creation dependent on α_i and $I_{\alpha j} \in \alpha_j < I_{\alpha i} >$. Cardinality type of this dependency is: 1-to-1 if $|\alpha_j < I_{\alpha i} >|=|\alpha_i \ll I_{\alpha j} \gg|=1$; 1-to-m if $|\alpha_j < I_{\alpha i} >| >1$ and $|\alpha_i \ll I_{\alpha j} \gg|=1$; m-to-1 if $|\alpha_j < I_{\alpha i} >|=1$ and $|\alpha_i \ll I_{\alpha j} \gg|>1$; or m-to-m if $|\alpha_j < I_{\alpha i} >|>1$ and $|\alpha_i \ll I_{\alpha j} \gg|>1$.

Definition 2.6: Given artifacts α_i and α_j and their instances $I_{\alpha i}$ of α_i and $I_{\alpha j}$ of α_j, $I_{\alpha j}$ is sync dependent on $I_{\alpha i}$ if α_j is sync dependent on α_i and $I_{\alpha j} \in Rv(I_{\alpha i}, msg)$. Cardinality type of this dependency is: 1-to-1 if $|Rv(I_{\alpha i}, msg)| = |Sd(I_{\alpha j}, msg)|=1$; 1-to-m if $|Rv(I_{\alpha i}, msg)| >1$ and $|Sd(I_{\alpha j}, msg)|=1$; m-to-1 if $|Rv(I_{\alpha i}, msg)|=1$ and $|Sd(I_{\alpha j}, msg)|>1$; or m-to-m if $|Rv(I_{\alpha i}, msg)| >1$ and $|Sd(I_{\alpha j}, msg)|>1$.

2.3 EZFlow-Tx: A Transactional Artifact-Centric BP Model

To avoid unnecessary recovery, we relax the atomicity property: besides "all" or "nothing", we allow an artifact be in a new state "fixed committed", i.e., the artifact is fixed by eliminating its error parts and preserving other parts.

In detail, we define the following transactional states of an artifact: *initial*, *active*, *committed*, *compensated*, *fixed-running*, and *fixed committed*. An artifact can correctly evolve to state *committed*, or to state *compensated* via rolling back operation. Especially, an artifact reaches state *fixed-running* if some parts of it fail and are fixed. After its execution resumes, it will finally reach state *fixed committed*. At the same time, we define two transactional states for the relation-typed attributes of an artifact: *normal* and *compensated*, representing whether sub-tuples of the attribute is affected by certain errors. Consequently, we add attribute *txState* to an artifact, and *rstate* to each relation-type attribute to record their transactional state resp.

Definition 3.7: A transactional artifact-centric BP schema W is a tuple (EZ, Tx), where EZ is an *EZFlow*, and

- Tx defines transactional attributes, it maps each task t in T to a tuple (rec, comp, fix) where *rec*=*trivial* | *compensatable*, *trivial* means t has no need to be recovered; and *compensatable* means the side effects of t can be semantically undone by its compensating task *comp* or its partial side effects corresponded to the failed parts can be semantically undone through its error fixing task *fix* after its completion.

In normal execution, the enactment of the transactional workflow is similar to that of EZFLow. The sequence of tasks in the enactment $t_0t_1..t_{n-1}$ forms a workflow transaction, wherein each task is a sub-transaction that keeps CID properties. When a task t fails, backward recovery will be initiated unless task t is *trivial*.

Now we describe the recovery procedure. When an artifact I_f fails and is recovered, its dependent artifacts will be identified. Suppose artifact I_r depends on I_f with the cardinality type of m-to-1 or m-to-m, i.e., I_f only contributes to parts of I_r, we will (1) invoke the fixing task *fix* of I_r to fix the parts affected by I_f and preserve unaffected parts. After fixing, sub-transaction of fixed tasks are set to *fixed committed*, *rstate* of the fixed sub-tuples are set to *compensated*, and *txState* of I_r is set to *fixed-running*; then (2) resume the execution of I_r to handle its normal parts upon recovery completion. If the executions of rest tasks of I_r are successfully committed, *txState* of I_r will be set to *fixed committed*. Note that cardinality and correlations are used to determine the failure parts of I_r. If an artifact is entirely affected, it will be wholly compensated, the *txState* of the artifact together with the *rstate* of all its sub-tuples will be set to *compensated.*

The transaction of a relevant scenario is **correct** if every failed transaction has been semantically undone or fixed, and artifacts, which have creation/created or sync dependency on it, have also been semantically undone or fixed.

3 A Recovery Mechanism for Relevant BPs

Here, we outline a creation/sync dependency discovery method and a recovery mechnism for keeping data consistency of relevant business scenario when failure occurs.

3.1 Creation/Sync Dependency Discovery and Overlaps

To discover dependencies among artifacts, we design two relations: (1) instances creation record *InstCre(cre, cred, msg, pdT, rvT, ts)* where *cre* is an artifact which sends the

message *msg* through task *pdT*, *cred* is an artifact created by task *rvT* at the time the *msg* arrives, *ts* represent the execution time of *rvT*; (2) instances sync record *SynDepd (sd, syn, msg, pdT, rvT, ts)* where *sd* and *syn* are artifacts that sends and receives the message *msg* resp. When a task is triggered by a message *msg*, a tuple will be added into *InstCre* if *msg* has "yes" as value of $\vdash$, otherwise, a tuple will be added into *SynDepd*.

When an instance I_i of artifact α_i fails and is recovered, its dependent artifacts can be derived as follows: (1) artifacts with creation dependency: $\pi_{cred}(\sigma_{cre=I_i}(InstCre))$; (2) artifacts with created dependency: $\pi_{cre}(\sigma_{cred=I_i}(InstCre))$; and (3) artifacts with sync dependency: $\pi_{syn}(\sigma_{sd=I_i}(SynDepd))$. For every dependent instance I_x, we can similarly compute the number of artifacts on which it is creation/created or sync dependent, thus, the cardinality type of dependency between I_i and I_x can be determined.

When cardinality type of dependency between I_x and I_i is not 1-to-1, we need to fix the error parts of I_x which overlap I_i rather than wholly roll back. The error parts can be computed by $\sigma_{\varphi}(\alpha_x)$, where α_x is the artifact class of I_x, and φ is constructed as follows. First, get correlation condition between α_x and α_i from correlation graph G, construct an equation expressions for each pair of projection attributes of each intersection expression in turn. Then construct two equation expressions $\alpha_x.ID = I_x$ and $\alpha_i.ID = I_i$ to filter the result. φ is a conjunction of all these equation expressions.

For example, when the *Order* instance o_3 in Fig. 2(c) fails, po_3 will need to be fixed for $po_3 \in PO < o_3 >$ and cardinality type of the dependency is m-to-1. Suppose *Order* and *PO* are correlated and the correlation condition is $\pi_{ID,\ item.invID}(Order) \sqcap \pi_{item.OrderID,\ item.invID}(PO)$. The failed parts of po_3 can be computed by $\sigma_{PO.ID=po_3 \wedge PO.item.OrderID=Order.ID \wedge PO.item.invID=Order.item.invID \wedge Order.ID=o_3}(PO)$.

3.2 A Mechanism for Handling Cascaded Recovery

Cascaded recovery needs to be considered because isolation is relaxed. When an instance I_i fails, its dependent instances I_D will be identified. However, dependencies do not necessarily indicate being affected. To avoid over-recovery, we use overlaps to determine if an instance $I_d \in I_D$ is really affected. Different overlap degrees have different recovery strategies: (1) no overlap: resume its execution; (2) complete overlap: entirely recover it; (3) partial overlap: fix the parts that overlap the failed instance, then resu- me its execution to handle the rest normal parts upon recovery completion. When I_d needs to be recovered, it may cause its dependent instances to be cascaded recovered.

We construct a directed acyclic graph Artifact Dependency Graph (ADG) for I_i, containing all its dependent instances which overlap its failed parts. Each node in an ADG represents an artifact. Its structure contains: (1) *cID*: artifact identifier, denoted by I_i; (2) *deFrom<cID, type>*: a list records artifacts that directly cause I_i to be recovered and corresponding affecting type; (3) *failedPart*: the failed parts of I_i; and (4) *bkRecCmd*: recovery command for I_i, which can be "compensate" or "fix". Each edge represents dependency between nodes with solid line for creation dependency and dash line for sync dependency. If a node I_x depends on node I_y, there will be an edge from I_x to I_y. The structure of an edge is a tuple *e=(source, target, type)* where *type* describes dependency type between *source* and *target* nodes, i.e., "create", "created" or "sync".

When building an ADG for the failed instance I_f, instances with creation/created or sync dependency on I_f are identified, and those ones overlap its failed parts are added into the graph. For a creation dependent instance I_{cd}, only instances that have sync or creation dependencies on it meanwhile overlap its failed parts are affected and need to be added. For a created dependent instance I_c, only instances which have sync or created dependencies on it meanwhile overlap its failed part need to be added. The recovery of a sync dependent instance I_{sd} is similar to that of the failed instance.

If I_i is affected by one instance, its *failedPart* is the intersection between I_i and the *failedPart* of that instance. If I_i is affected by multiple instances, overlaps between I_i and these instances will be computed resp., and the *failedPart* is the union of these overlaps. With regard to instances that may depend on each other, we use the following rules to avoid creating a cycle. Given two nodes I_x and I_y, I_y is dependent on I_x and overlaps its *failedPart*. I_y will be added as a dependent instance of I_x if I_y has not existed in the ADG. Otherwise, missing nodes affected by I_y should be added, and *failedPart* should be adjusted for I_y and all instances derived from it. For each derived instance I_z with creation type, if overlap between *failedPart* of I_y and I_z contains I_z's *failedPart*, similar *failedpart* adjustment should be done for I_z. The overlap adjustment is iteratively handled until there is no change to the intersection or there is no overlap with other instances. In this way, cycling dependency is avoided, and the recovery of I_y does not miss any affected parts since the impact of I_x's failure on I_y has been handled.

The ADG construction algorithm builds an ADG for the failed instance I_f. The algorithm has two inputs: I_f and its fail parts *failedPart.* An ADG is represented by a set of nodes *N* and a set of edges *E*. The algorithm is outlined as follows.

1. Create two assisting lists: (1) *affLi<aff, deFrom, dp>* to store affected instances and iterate through different level of a ADG, where *aff* is derived from *deFrom* with dependency type *dp*; (2) *tempLi* to temporarily store the affected instances of instances in *affLi.*
2. Construct a node for I_f and add it into *affLi* to start ADG construction.
3. For each element *el* in *affLi*,
   3.1. If *el.aff* does not exist in *N*, add an edge from *el.deFrom* to *el.aff* into *E*, add every affected instances of *el.aff* into *tempLi.*
   3.2. Else retrieve *el.aff* into I_m (affected by multiple instances. No cycle). If NotContain (I_m*.deFrom.type, el.dp*) or NotEmptyDifferenceSet(*el.aff.failedPart*, I_m*.failedPart*), add missing instances affected by *el.aff* into *tempLi*, adjust *failedpart* of all affected instances.
4. After processing all elements of *affLi*, empty *affLi* and move instances from *tempLi* to *affLi* to start next level construction. If *affLi* is not empty, repeat step 3.

Fig.3 shows a sample ADG_1 for I_f containing ten concurrent BPs I_1-I_{10}.

The ADG is used to conduct cascaded recovery. When a BP I_f fails, its dependent BPs will be identified and suspended for overlap evaluation, while independent BPs keep running. Then the recovery procedure invokes the ADG construction algorithm to construct an ADG for I_f, with failed parts of each affected instance computed, and finally traverse the ADG to recover each affected instance I_a according to its recovery command *bkRecCmd*, which is set to "compensate" when its failed parts equal to its contents, or set to "fix" when its failed parts are a proper subset of its contents. Due to space limit, algorithm details are ignored here.

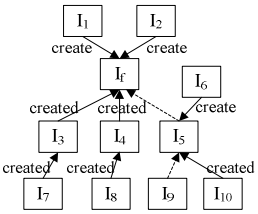


Fig. 3. Sample ADG_1 for I_f

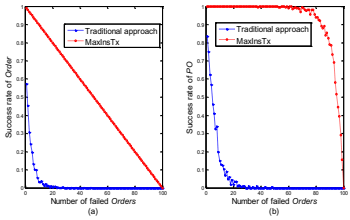


Fig. 4. Success rate comparison

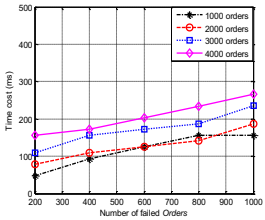


Fig. 5. Time cost

4 Evaluation of Effectiveness and Feasibility

To evaluate the effectiveness, we compare the BP success rate of *MaxInsTx* with that of traditional approach (all or nothing). Experiment is conducted on *m Orders* and *n POs*, where *m*=100, *n*=20. We assign [1,5] items to each *Order* with uniform distribution. The probability that an item in an *Order* belongs to a certain *PO* is 1/n.

Fig. 4 shows the success rate of *Orders* and *POs*. In Fig. 4(a), the success rate of *Orders* of *MaxInsTx* decrease linearly because only the failed *Orders* will fail. In Fig. 4(b), the success rate of *POs* of *MaxInsTx* shows no impact when the number of failed *Orders* varies from 1 to 54, because *POs* will fail only when they become empty due to the failed *Orders*. After that, the *SRPO* decreases gradually for a while and then decreases dramatically in the last period. However, both the success rate of *Orders* and *POs* of traditional approach decrease dramatically, since other *Orders* and all the *POs* correlated with the failed *Orders* will fail. Obviously, *MaxInsTx* has much better performance, which verifies its effectiveness.

Next, we analyze the time cost of *MaxInsTx*. In this experiment, the number of failed *Orders* varies from 200 to 1000, the number of total *Orders* varies from 1000 to 4000, and the number of *POs* is fixed to 20. The configuration of our microcomputer is as follows: 3.2 GHz Dual Core processor, 2 GB memory, Windows XP OS.

Fig. 5 shows that the time cost is affected by both the number of failed *Orders* and total *Orders*. The time cost curve increases slowly with the increasing of the number of failed *Orders*. The overall time cost is very small. Hence *MaxInsTx* is feasible.

5 Related Work

Many works have addressed the issue of long-live transaction coordination through ACID relaxation. One of the representative approaches is proposed in [4] which allows relaxing any attributes of ACID through introducing a "pre-commit" phase and a "negotiation" phase. Xiao *et al* [5] further address the issue of concurrency control. They propose a rule-based approach to resolve the concurrency issue through analyzing write dependency and potential read dependency among BPs.

None of the above works take the impact of complex cardinality relationships into consideration. In practice, the existing of m-to-m relationships among BPs complicates many problems and attracts increasing attention. Fahland *et al* [1] address the behavioral conformance checking problem complicated by the fabric of BP instances. Zhao *et al* [2] focus on managing instance correspondence through correlations attached to each

instance. Sun *et al* [3] develop a choreography language which supports instance level correlations and cardinality constraints. Different from these works, our research focuses on solving the recovery issue for the fabric BPs. A recovery approach is proposed to resolve the impact of a failed BP on concurrent BPs meanwhile avoid over-recovery.

6 Conclusion

This paper proposed a best-effort failure recovery approach *MaxInsTx* for relevant business scenarios, supporting complex dependencies between artifacts. We relax the atomicity property of transactions, allowing an artifact be partially fixed such that its unaffected parts are preserved as much as possible. A failure recovery mechanism is proposed with the advantage of avoiding unnecessary recoveries.

Acknowledgment. This work is partially supported by NSFC grant 60873115.

References

1. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Conformance Checking of Interacting Processes with Overlapping Instances. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 345–361. Springer, Heidelberg (2011)
2. Zhao, X., Liu, C., Yang, Y., Sadiq, W.: Handling instance correspondence in inter-organisational workflows. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007. LNCS, vol. 4495, pp. 51–65. Springer, Heidelberg (2007)
3. Sun, Y., Xu, W., Su, J.: Declarative Choreographies for Artifacts. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 420–434. Springer, Heidelberg (2012)
4. Khachana, R.T., James, A., Iqbal, R.: Relaxation of acid properties in AuTra, the adaptive user-defined transaction relaxing approach. Future Gener. Comput. Syst. 27(1), 58–66 (2011)
5. Xiao, Y., Urban, S.D.: Using rules and data dependencies for the recovery of concurrent processes in a service-oriented environment. IEEE Transactions on Services Computing 5(1), 59–71 (2012)
6. Xu, W., Su, J., Yan, Z., Yang, J., Zhang, L.: An Artifact-Centric Approach to Dynamic Modification of Workflow Execution. In: Meersman, R., et al. (eds.) OTM 2011, Part I. LNCS, vol. 7044, pp. 256–273. Springer, Heidelberg (2011)