12-15-2014

Practical Concurrency Support for Web Service Transactions

Emad Alsuwat
University of South Carolina - Columbia

PRACTICAL CONCURRENCY SUPPORT FOR WEB SERVICE TRANSACTIONS

by

Emad Alsuwat

Bachelor of Computer Science
Taif University, 2008

Submitted in Partial Fulfillment of the Requirements

For the Degree of Master of Science in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2014

Accepted by:

Csilla Farkas, Director of Thesis

Marco Valtorta, Reader

Manton M Matthews, Reader

Lacy Ford, Vice Provost and Dean of Graduate Studies

ACKNOWLEDGEMENTS

First and foremost I express my sincere gratitude for my advisor Prof. Csilla Farkas. It has been an honor to be her master student. I am very thankful for her motivation, immense knowledge, and continuous support. I appreciate all her contributions of time, and ideas to make my master experience productive and stimulating.

Besides my advisor, I would like to thank the rest of my thesis committee, Prof. Manton M Matthews, and Prof. Marco Valtorta, for their insightful comments and encouragement.

I would like to thank my family for all their unlimited encouragement and love. I would like to sincerely thank my parents who raised me with a love of science and for providing me the support I need. I also would like to thank my brothers and sisters for supporting me spiritually throughout my life.

Last but not the least, I would like to thank my friends for all their love and true friendship. My time at the University of South Carolina was made enjoyable in large part due to the many friends that became a part of my life. I am grateful for time spent with them and grateful for unforgettable memories.

ABSTRACT

Traditional database concurrency control methods use locking, timestamp-ordering, and optimistic-ordering to achieve DB consistency. However, these approaches are not suitable for long-running Web Service Compositions (WSCs) due to associated performance degradation. Our hypothesis asserts that, using transactional semantic and ordering information, the execution time of a WSC can be reduced, thus allowing the use of traditional database concurrency control methods while avoiding unacceptable performance degradation. Our solution is based on the following approaches:

- We model a WSC as WS-BPEL specification, i.e., a partial order of WS transactions.
- We allow some of the WS transactions in the WSC to execute in parallel.
- We use traditional locking mechanism for WSC to guarantee database consistency.

To identify WS transactions that can execute parallel, we analyzed the WS-BPEL specification of the WSCs. The research tasks follow:

- Task 1: Identify WS transaction precedence relations
- Task 2: Build Parallel Execution Scenarios (PES)
- Task 3: Investigate possible further improvement of WSC execution schedule. For Task 3, we propose the following sub-tasks:
    - Increase the number of WSs executing in parallel, and
    - Execute concurrently those WSs that have similar execution time

In our work we will present our theoretical model and complexity calculation.

TABLE OF CONTENTS

LIST OF FIGURES

List of Abbreviations

ACID ..... Atomicity, Consistency, Isolation, Durability

DB ..... Database

PES ..... Parallel Execution Scenario

TO ..... Topological Order

WS ..... Web Service

WSC ..... Web Service Compositions

WSP ..... WS-Pairs

WSS ..... Web Service Set

WS-BPEL ..... Web Services Business Process Execution Lang

CHAPTER 1

INTRODUCTION

Within the field of information technology and computer science, concurrency controls play a vital role in triggering concurrent operations at the quickest speeds possible based on hardware and software system compilations defined by specified consistency rules. During interactivity, problems may arise. The general basis for these controls is to safeguard system integrity. Database systems are essential for a wealth of applications. A database state defines the value of the database objects and this state is changed when a user executes a transaction; individual transactions are assumed to be correct in that they are running in isolation but this assumption changes when multiple users access multiple database objects on multiple sites within the context of a distributed database system (Bhargava, 1999). At this point, problems arise.

In traditional database management systems, concurrency controls are used to ensure that executed transactions on the database system are running concurrently without causing any violation to the consistency of the database; to ensure the correctness of concurrency control mechanisms, the database management system must meet two criteria for correctness: 1) database integrity and 2) serializability (Bhargava, 1999). Simply stated, concurrency controls protect data simultaneous data modification from impeding on changes made by another user. Simultaneous transaction executions over a shared database can erode data integrity and results in consistency problems, including

lost updates, uncommitted data, and inconsistent retrievals (Kung & Robinson, 1981). Techniques are needed to mitigate these risks through improved data management. Many concurrency control algorithms can be leveraged to validate serializability, including using mechanisms such as locking, timestamp ordering, or optimistic ordering. Serializability architects sequential database transitions from one state to the next, based on the blueprinted serial execution of all transactions (Bhargava, 1999).

The idea of concurrency control mechanisms is to guarantee database consistency while improve the performance of the database systems. In this process, serializability is the universally accepted correctness criterion for processing multiple transactions within centralized high-performance multiple transaction processing systems (Thomasian, 1998). Concurrency control mechanisms confirm serializability in centralized and distributed within traditional database management systems (DBMS) and this is achieved by verifying the reliability of database transactions by assuming that each transaction is ACID - that is, atomicity, consistency, isolation, and durability properties hold. Two-Phase Locking (2PL) is the most widely used concurrency control mechanism (Al-Jumaha et al., 2002). Yet these mechanisms are not suitable for long-running WSCs (Web Service Cluster) due to the associated performance degradation. That is, transactions in the service-oriented architecture are not ACID transactions due to the heterogeneous nature of the execution environment. The lack of concurrency control mechanisms in WSC causes a problem of WS-serializability (Web Service), which occurs when multiple transactions read and write the same data through independent services at the same time (i.e. allowing for dirty read and dirty write). The dirty reads and writes make the recovery in WSC more difficult, meaning that if a transaction fails and needs to

be aborted, it will affect the consistency of other concurrent transactions due to the lack of isolation levels (Gao et al., 2011).

In this thesis, the problem of concurrency control for WSCs is the target focal point. The study prioritizes the problem of WS-serializability given that current web service transactions do not ensure concurrency control due to the heterogeneous nature of the execution environment. In addition, the research proposes a concurrency control method that is suitable for WS-transactions. The research hypothesis predicts that, by reducing the execution time of WSCs, the applicability of concurrency control for WSCs can be improved, therefore enhancing both transaction correctness (i.e., serializability) and efficiency.

Our proposed solution is based on the following claims:

- We model a WSC as WS-BPEL (Web Services Business Process Execution Language) specification, i.e., a partial order of WS transactions.
- We allow some of the WS transactions in the WSC to execute in parallel.
- We use traditional locking mechanism for WSC to guarantee database consistency.

To identify WS transactions that can achieve parallel execution, we analyzed the WS-BPEL specification of the WSCs. The research tasks follow:

- Task1: identify WS transaction precedence relations
- Task 2: build Parallel Execution Scenarios (PES) that do not violate the precedence relations identified in Task 1.
- Task 3: Investigate possible further improvement of WSC execution schedule. For Task 3, we propose the following sub-tasks:

- Increase the number of WSs executing in parallel, and
- Execute concurrently those WSs that have similar execution time.

This thesis is organized as follows: Section 2 presents an overview of the related work and limitations of current research. A Motivated Example is presented in Section 3. Sections 4 and 5 provide the research problem and the proposed solution. Finally, our conclusion and future work is presented in section 6.

CHAPTER 2

RELATED WORK

2.1 CONCURRENCY CONTROL MECHANISMS

The problem of violating the consistency in database systems appears within the context of executing multiple concurrent transactions in the same database. Concurrency control algorithms are introduced to solve the problem of inconsistency and to guarantee the correctness and integrity of the execution of concurrent transactions within these database systems. Three generic approaches are used when designing concurrency control algorithms to achieve synchronization: 1) Wait - If and when two transactions conflict, one must wait until the actions of the other have been completed; 2) Timestamp - Transaction selection order is determined by unique timestamps and processing is based on these rankings; and 3) Rollback - If two transactions conflict, some actions are undone or rolled back or one of the transactions is restarted (deemed optimistic due to expectations that conflicts require only a few transactions would require rollback) (Bhargava, 1999). In other words, concurrency control mechanisms are used to ensure serializability in database systems (Sheikhan et al., 2013).

Standard locking or two-phase locking (2PL) is the primary concurrency control method utilized in centralized databases; this method mandate lock release only when the transaction is committed or aborted, as this prevents cascading aborts that occur when a transaction accesses modified object initiated by an uncommitted transaction (Thomasian, 1998). Locks are based on the wait strategy and characterized as pessimistic, meaning

that conflicts are assumed and common techniques involve detecting and resolving conflicts via blocking and as quickly as possible (Kung & Robinson, 1981). Waiting times can be reduced by employing *readlocks* or *writelocks*, each of which limits the entity's actions until the competing transaction has completed the operation and the unlock is triggered, thus rendering these entities readily available for access. Lock and unlock operations may be embedded within the transaction by the user or may be transparent to the transaction; in the later method, the system is accountable for appropriately granting and enforcing lock/unlock activity for each transaction (Bhargava, 1999). However, locking may create problems, including *livelock* and *deadlock*. In both instances, the lock mechanism fails in its function, resulting in either repeated failure to obtain a lock (livelock) or multiple, simultaneous locking on entities (deadlock) (Bhargava, 1999).

Timestamp ordering differs in that is based on the premise that the read set and write set of each transaction is known, thus allowing group transactions to be categorized. In this method, transaction execution is bound to timestamp ordering and clock synchronization safeguards system integrity. When transaction conflict arises, processing defaults to timestamp ordering (TSO), which produces serialization histories (Bhargava, 1999). Finally, optimistic-ordering methods or rollback mechanisms is based on validation mechanisms, employing conflicts and timestamp data to guide validation. According to Bhargava (1999), "The optimistic approach maximizes the utilization of syntactic information and attempts to make use of some semantic information about each transaction;" the four phases of transaction execution within this optimistic concurrency control framework are: 1) Read, 2) Compute, 3) Validate, and 4) Commit and Write (p.

6). Optimistic methods are grounded in the perspective that locking results in disadvantages and potential costs that are unnecessary given the supposed rarity of conflict, thus contending that better performance may be achieved by instead managing conflicts at the commit time (Kung & Robinson, 1981). As such, optimistic concurrency controls allow transactions to be executed without restrictions and check for conflict just prior to occurrence.

Our proposed approach is closely related to lock-based concurrency control. As previously stated, 2PL is the most widely lock-based mechanism that can ensure serializability in traditional database systems. A transaction in 2PL needs to make a lock request to obtain a lock (Read or write) on any data item that this transaction needs to access (Al-Jumaha et al., 2002). 2PL has two kinds of locks: shared locks and exclusive locks. Shared locks are not typically the source of problems since many transactions can obtain shared locks without affecting the system performance; however, two-phase locking may be subjected to deadlocks when two or more transactions are waiting for each other to release the exclusive lock on a data item since no two transactions can obtain exclusive locks on the same data item according to the rules of 2PL (Thomasian, 1998). Two-Phase Commit (2PC) is another protocol that is used to ensure serializability in distributed database systems. The goal of this protocol is to ensure the consistency of a transaction that executes in a distributed database. Therefore, all sites have the right to decide and reach the same final decision, whether committing the transition (if all sites say yes) or aborting the transaction (if there is a no vote or a failure in one site) (Lechtenbörger, 2009). In summary, these three basic methods define the current

concurrency control mechanisms used. Extensive research has been used to substantiate advantages and disadvantages of each method.

2.2 WEB SERVICE COMPOSITION AND TRANSACTIONS

The aim of services oriented architecture (SOA) is the integration of Web services (WS) over the Web. In the article, "Composing aggregate web services in BPEL," Ezenwoye and Sadjadi describe Web services (WS) as a software system that is designed with the purpose of simplifying machine-to-machine interaction over the Web. The service of each WS is described in the Web Service Description Language (WSDL), and the interaction with other WSs is through Simple Object Access Protocol (SOAP) messages (Ezenwoye and Sadjadi, 2006). Web Service Composition (WSC) is defined in the article, "Semantic Web Service Composition Approaches: Overview and Limitations," as the process of composing available Web services (WS) in order to get new functionality that has to meet the description a technical process in business areas (Zeshan and Mohamad, 2011).

There are fundamental requirements related to the process of defining web services (WS) in the services-oriented architecture (SOA) framework. For instance, a major requirement in the process of defining a Web service in a services-oriented architecture (SOA) framework is composition of services into business processes. Composition of services into business processes provides benefits, including: the ability to provide flexible support for business process; increased ease of modification of business processes; ability to efficiently change costumers' requirements (i.e., the changing the requirement after composition of services will be faster with less effort);

and use of Business Process Execution Language (BPEL), defined as an executable language that has the ability to work as an engine for action specification inside business processes (Juric, 2006). In total, the idea of web service composition is based on the understanding that web service composition aims to build a new composition that has specific goals set under specific constraints.

In "Transaction Compensation in Web Services," Strandenaes and Karlsen (2003) describe transactions in web service composition as transactions that enable each web service to effectively interact with other web services within a shared business domain. These transactions are also known as conversational transactions since each transaction typically consists of multiple sub transactions. Conversational transactions do not enforce ACID properties due to the loosely coupled and heterogeneous nature of web service environment. Therefore, if anyone of the sub transactions within a conversational transaction aborted, every transaction inside this conversational transaction must be aborted as a result, thus enforcing the all-or-nothing property on which the transaction is contingent (Strandenaes and Karlsen, 2003).

Similarly, compensation transaction support is an important requirement in the composition of business processes. Strandenaes and Karlsen (2003) define the compensation transaction as a transaction that "semantically undoes the partial effect of a [conversional] transaction … without performing cascading aborts of dependent transactions, restoring the system to a consistent state." Therefore, when aborting a conversational transaction, the compensation transaction can be used in the recovery of the committed sub transactions and thus assist in maintaining the total consistency of the database system.

Web Service Business Process Execution Language (WS-BPEL) is an additional, critical requirement in the composition of business processes. In "Composing aggregate web services in BPEL," researchers Ezenwoye and Sadjadi define WS-BPEL as a standard language that can be used to describing actions for web services within business processes. WS-BPEL must be able to meet stated standards such as supporting long running transactions, facilitating recovery after aborting conversational transactions, and managing various failures arising within the business process (Ezenwoye and Sadjadi, 2006). In conjunction, long-running processes support is another vital component. Long-running transactions are those transactions capable of running for a long period of time, varying from hours to days in duration. Wang et al. define long running processes as transactions that avoid obtaining locks on non-local resources and use compensation in order to handle failure and enforce all-or-nothing (ACID) properties. Therefore, long running processes must meet fundamental requirements to foster endurance, including possessing short running processes properties (i.e. all-or-nothing properties), the ability to rollback long running processes automatically without human participation, and actively aborting short running processes if there is a conflict with long running processes since long running processes have higher priority over short running processes (Wang et al., 2004).

2.3 WEB SERVICE SERIALIZABILITY

In a service-oriented architecture (SOA), there is no support for serializability because of the inefficiency of traditional database concurrency support. As a result, we can conclude the limitations of the current research in service-oriented architectures, arguing: there is limited concurrency support for WS-transactions since there is currently

no serializability and therefore no consistency in business processes due to the heterogeneous nature on the Web service environment. In addition, feasibly, the use of traditional database concurrency control algorithms such two-phase locking (2PL) to enforce ACID properties in long running processes is undesirable given that the performance level is deemed unacceptable. Therefore, the proposed solution presented in this study is different from pervious research as it is based on the claim that we can reduce the Web Service Compositions (WSCs) execution time by allowing some of the WS transactions in the WSC to execute on parallel paths; we also use a traditional locking mechanism in partnership with the parallel WSs to guarantee database consistency.

CHAPTER 3

MOTIVATED EXAMPLE

Consider a retailer X offering Web services with a service provider SPX, allowing for order placement (OP) and order cancellation (OC). Each one of these services is a compensation service of the other one. Suppose that two business processes (A and B) are invoked via these web services due to price differentiation: Process A – cancelled order in retailer X and placed order in retailer Y. Process A aims to avoid dual order placement completion; this action seeks to preserve scope atomicity, meaning that compensate actions are able to run smoothly in that a block of actions can be completed and therefore compensated. As such, the process designer creates a transaction process that maintains the atomicity property. The scenario is:

1. Process A releases initial order by invoking OC in SPX.

2. Process B reserves released by process A order by invoking OP in SPX.

3. Process A is trying to place order at a retailor Y. It invokes the web service OP in the service provider of retailor Y, SPY. It can be assumed that this effort fails due to fact that Y is out of requested item stock.

4. In the meantime, the process B, through payment, commits placing order at X.

5. The lack of supply stock in Y leads to the fact that the process A wants to make Web service compensation invoked in step 1. However, this compensation is not possible.

There are limitations of the applicability of the proposed solution:

- If we are dealing with fully automated WSC our pre-processing will reduce execution time; however,
- If we are dealing with WSC that requires manual interaction our pre-processing may not be able to semantically reduce the execution time.

CHAPTER 4

PERFORMANCE IMPROVEMENT: PAIR-WISE PARALLEL EXECUTION

The hypothesis of this research is that the execution time of WSC can be reduced and the efficiency of WS-concurrency control can be improved; thus enabling the use of locks based concurrency controls for WSCs.

The proposed solution is based on the observation that we can model a WSC as WS-BPEL specification (i.e., a partial order of WS transactions). This representation will allow us to identify WS transactions in the WSC that can be executed in parallel, and, therefore, reduce transaction execution time. Our expectation is that the reduced execution time will allow us to use traditional locking mechanism to guarantee database consistency. Furthermore, we are aiming to reduce the overhead created by data item locking by proposing semantically enhanced locks types.

The research problem is:

1. Identifying subgroups of WSs that can run concurrently without potential violation of consistency of the database.

To identify WS transactions that can execute parallel, we analyzed the WS-BPEL specification of the WSCs.

4.1 TASK 1: IDENTIFY WS TRANSACTIONS THAT CAN EXECUTE CONCURRENTLY

Let a WS composition be a directed acyclic graph (DAG), where the vertices are the WSs, and the links represent service request between the WSs, i.e., if WS_i requests service from WS_j, there is a directed edge from WS_i to WS_j. We evaluate each WS-composition from the perspective of:

- Precedence relation between WSs, and
- Conflicting operations between WSs.

First: for the identification of the WS subgroups, we use the precedence relation among WS in a composition.

Algorithm 1 finds all set of WS-pairs (WSP) that are not ordered by precedence and do not conflict.

Algorithm 1: Finding-WSP(WSC)

Input: WSC (WS-BPEL specification).
Output: *WSP*= {(WS_1, WS_2), … , (WS_i, WS_j)}, such that there is no path between the WSs of any of the pairs.

```
1. // Subroutine to insert all possible pairs of WSC into WSP
2. Inserting-all-possible-pairs(WSC)
3. WSP ← Empty set that will contain all possible set of pairs
4. WS ← Set of all nodes (Web services) in WSC
5. WSC'← WSC
6. while WS is non-empty do
7.     choose WSa∈ WS such that WSa has no incoming edge in WSC'
8.     remove WSa from WS and WSC'
9.      if WS is non-empty then
10.          for each WSb in WS do
11.              insert(WSa, WSb)into WSP
12.         endfor
13.     endif
14. endwhile
15. return WSP (all possible set of pairs)
16. // Subroutine to drop ordered pairs from WSP
```

```
17. Dropping ordered pairs from WSP (DAG, WSP)
18. for each pair (WS_a, WS_b) ∈ WSP do
19.    if there is a path from WS_a to WS_b then
20.              remove(WS_a, WS_b)from WSP
21.    endif
22. endfor
23. // we will return WSP
24. return WSP
25. // Subroutine Drop-conflicting-pairs
26. Drop-conflicting-pairs-from-WSP (WSP)
27. for all WS_i in WSC do
28. Write_set(WS_i) = {A_1, A_2, …, A_k} such that WS_i writes items
A_1, A_2, …, A_k
29. Read_set(WS_i) = {A_1, A_2, …, A_l} such that WS_i reads items A_1,
A_2, …, A_l
30. endfor
31. For each pair (WS_a, WS_b) ∈ WSP do
32.      if write set(WS_a) ∩ write set(WS_b) ≠ Φ
33.      or write set(WS_a) ∩ read set(WS_b) ≠ Φ
34.      or read set(WS_a) ∩ write set(WS_b) ≠ Φ then
35.            WS_a and WS_b are conflicting
36.             remove (WS_a, WS_b) from WSP
37.      endif
38. endfor
39. return WSP
```

THEOREM 1: ALGORITHM 1 GENERATES

1) (Completeness) all pairs of WSs that do not have precedence relation with each other and do not conflict; and

2) (Soundness) only those pairs of WSs that meet the above characteristics.

PROOF OF THEOREM 1:

(Completeness) Assume, by contradiction, that there is a pair (WS_i, WS_j) such that

i. there is no precedence relations between WS_i and WS_j,

ii. WS_i and WS_j have no conflicting operations, and

iii. (WS_i, WS_j) is not in WSP.

If WS_i and WS_j are in WSC, then by lines (6-14) there must be a pair (WS_i, WS_j) (or an equivalent pair (WS_j, WS_i)) in WSP. But then, by assumption iii (WS_i, WS_j) must have been removed from WSP if it is not in the final set. (WS_i, WS_j) could have been removed either during the process of dropping pairs that have 1) precedence relations in lines (18-22) or when 2) removing conflicting pairs in lines (27-38).

In the drop-order-subroutine, in lines (18-22), we only remove pairs from WSP if they have a precedence ordering. But then by assumption i, the pair (WS_i, WS_j) cannot be removed by this subroutine.

But then, the pairs (WS_i, WS_j) could have been removed only by the drop-conflicting-pairs subroutine, in lines (27-38). However, this subroutine removes a pair from WSP only if WS_i and WS_j have conflicting operations. By assumption ii, WS_i and WS_j have no conflicting operation; therefore, the pair (WS_i, WS_j) cannot be removed from WSP.

This illustrates that (WS_i, WS_j) must be in the final set WSP. This is a contradiction to our original assumption.

(Soundness) Assume, by contradiction, that the pair (WS_i, WS_j) is in WSP but either

i. there is a precedence relation between WS_i and WS_j.

or

ii. there is a conflicting operation between WS_i and WS_j.

In lines (6-14), **Algorithm 1** creates a pair (WS_i, WS_j) (or (WS_j, WS_i)) only if both WS_i and WS_j are in WSC. The pair (WS_i, WS_j) has not been removed from WSP since it is in the final set. (WS_i, WS_j) must be removed from WSP either during the process of dropping pairs that have 1) precedence relations in lines (18-22) or when 2) removing conflicting pairs in lines (27-38). In the drop-order-subroutine, in lines (16-24), we remove pairs if they have a precedence ordering. If WS_i and WS_j have precedence relation by assumption i, then the pair (WS_i, WS_j) must be removed by this subroutine, therefore, cannot be present in the final set WSP. This is a contradiction of our assumption. If (WS_i, WS_j) has conflicting operation (assumption ii), then the pair should have been removed by the drop-conflicting-pairs subroutine in lines (25-38), therefore, cannot be present in the final set WSP. This is a contradiction of our assumption.

This illustrates that (WS_i, WS_j) must not be in the final set WSP. This is a contradiction to our original assumption.

TASK 1 OUTCOME: Set of WS-pairs (WSP) that are not ordered by precedence and do not conflict.

4.2 TASK 2: BUILD ALL PARALLEL EXECUTION SCENARIOS (PES)

Given a web service composition (WSC) as a partial order over web services {WS_1, …, WS_k}, and WSP as a set of web service pairs (WS_i, WS_j) such that WS_i and WS_j can execute concurrently. In this task we perform two subtasks. First, we generate a topological order (TO) of the web service in WSC. Next, we replace individual services

of TO with pairs of concurrently executing services. For the second subtask, we aim to maximize the number of parallel executing services.

4.2.1 TOPOLOGICAL ORDER

In this section, we present algorithm 2 to find a topological order using Breadth First Search.

Algorithm 2: Find-Topological-Order(WSC)

Input: WSC (WS-BPEL specification).
Output: TO = {WS_1, WS_2, … , WS_k)}, which is the BFS topological ordering of web services in WSC such that for any edge from WS_i to WS_j in WSC, WS_i must precedes WS_j in the total ordering TO.

```
1. // Find-topological-order(WSC)
2. WS ← Set of all nodes (Web services) in WSC
3. E ← Set of all edges in the WSC represented as (WS_a, WS_b)
4.//indicating that there is a directed edge from WS_a to WS_b, i.e., WS_a must preceed WS_b
5. TO ← Empty list that will contain the topological order of WSC
6. while WS is not empty do
7.       for each WS_i in WS do
8.             if WS_i has no incoming edges do
9.               remove WS_i from WS
10.              append WS_i to TO
11.              for each edge e originating from WS_i do
12.                    remove e from E
13.              endfor
14.            endif
15.       endfor
16. endwhile
17. return TO
```

4.2.2 OPTIMAL WEB SERVICE REPLACEMENT

In this section we develop Algorithm 3 to find the optimal set of WS-Pairs ($WSP_{optimal}$) that can be substituted in the topological order TO.

Algorithm 3: Optimal-Pairs(WSP)

Input: WSP
Output: The optimal set of WS-Pairs ($WSP_{optimal}$) that that can be substituted in the topological order TO.

```
1. Optimal-Pairs(WSP)
2. L ← Empty set that will contain already used WSs while processing
3. PAIRS ← Empty set that will contain the current calculated
4. // set of pairs that might be WSP_optimal
5. WSP_optimal ← An empty set that will contain the optimal set of pairs
6. WEIGHT ← 0 // The corresponding weight for each set in Pairs
7. MAX_WEIGHT ← 0 // the weight of the current maximum set of pairs
8. for each pair (WS_i, WS_j) in WSP do
9.     WSP_Working ← WSP
10.    add WS_i and WS_j to L
11.    add (WS_i, WS_j) to PAIRS
12.    increase WEIGHT by one
13.    drop (WS_i, WS_j) from WSP_Working
14.    for each pair (WS_a, WS_b) in WSP_Working do
15.        if WS_a or WS_b in L then
16.            Drop (WS_a, WS_b) from WSP_Working
17.        else
18.            (WS_i, WS_j) ← (WS_a, WS_b)
19.            goto line 11
20.        endif
21.    endfor
22.    if WEIGHT is greater than MAX_WEIGHT then
23.        WSP_optimal ← PAIRS
24.        MAX_WEIGHT ← WEIGHT
25.    endif
26.    set PAIRS ← empty
27.    set L ← empty
28.    set WEIGHT ← 0
29. endfor
30. return WSP_optimal
```

DEFINITION 1: CORRECTNESS CRITERIA:

Given a WS-pair, (WS_i, WS_j) in TO, we can rewrite TO denoted as $TO^{\rightarrow}$ by replacing (WS_i, WS_j) with $WS_i \rightarrow WS_j$, where $\rightarrow$ indicate a precedence in the topological order.

We say that TO is correct if there exists a topological order, $TO^{\rightarrow}$, of WS-composition, such that

$$\text{to} = TO^{\rightarrow}$$

DEFINITION 2: OPTIMALITY:

Given a WS-composition, WSP, and a TO, we say that TO is optimal if there is no other correct topological orders to, such that

$$to_{\ \#\ of\ replaced\ WSs} > TO_{\ \#\ of\ replaced\ WSs}$$

- Generate a topological order
- Select optimal replacement
- Show properties

THEOREM 2: ALGORITHM 3 GENERATES

1) Correct WS-Pairs ($WSP_{optimal}$) as (Definition 1)

2) The optimal set of WS-Pairs ($WSP_{optimal}$) that can be substituted in the topological order TO as (Definition 2).

PROOF OF THEOREM 2:

To prove correctness and optimality of algorithm 3, we need to show the following properties of the loop invariant: 1) initialization, where we have to show that the algorithm is true prior to the first iteration of the loop, 2) maintenance, where we have to show that if our algorithm is true before an iteration of the loop, then it remains true before the next iteration, and 3) termination, which means that when the loop terminates, the invariant gives a useful information that allows to show that the algorithm is correct and optimal, i.e., we have to show that the resulted pairs of this algorithm are the correct and optimal pairs that can be substituted in the topological order TO.

Loop invariant: At the start of each iteration of the for loop of lines (8-29), the list $WSP_{optimal}$ consists of the current optimal pairs of web services that might be substituted in the topological order TO. By the end of the last iteration, the list $WSP_{optimal}$ will contain the optimal pairs of web services that will be substituted in the topological order TO.

Initialization: Prior to the first iteration of the for loop in lines (8-29), the list $WSP_{optimal}$ will be empty; **t**herefore, we can say that the loop invariant holds prior to the first iteration.

Maintenance: The body of the for loop works by picking up the first pair (WS_i, WS_j) from WSP, then adding the pair (WS_i, WS_j) to the list PAIRS, and the services WS_i and WS_j to the list L. The inner for loop in lines (14-21) will delete any pair from $WSP_{working}$ that has either WS_i or WS_j (lines 14-16). In the inner for loop, if the current picked up pair does not have WS_i or WS_j, then we will assign this new pair (WS_a, WS_b)

to (WS_i, WS_j), and restart the inner for loop until $WSP_{working}$ is empty (lines 11-19). By the end of the inner for loop, if the current weight is higher than the current maximum weight, then that means we find a new optimal set of pairs; therefore, we need to 1) assign the pairs in PAIRS to $WSP_{optimal}$ (line 23), 2) assign the value of the variable WEIGHT to the variable MAX_ WEIGHT (line 24), and 3) reset the variables PAIRS, L, WEIGHT getting ready for the next iteration (lines 26-28). Hence, we can say the loop invariant is true before the next iteration the for loop.

Termination: When the for loop terminates (lines 8-29), the list $WSP_{optimal}$ has the correct and optimal pairs of WSP that can be substituted in the topological order TO. Hence, we can say that algorithm 3 is correct and optimal.

COMPLEXITY:

Let n denote the number of times the for loop iterates (lines 8-29) (the number of times this loop iterates is equal to the number of pairs in WSP). Also, the inner for loop (lines 14-21) will iterate n-1 times in the worst case scenario since $WSP_{working}$ may have all the pairs of WSP except the one dropped by line 13; therefore, the running time of **algorithm 3** in the worst case scenario is

$$T(n) = n(n-1) = \Theta(n^2).$$

4.2.3 TOPOLOGICAL ORDER WITH WEB SERVICE PAIRS

In this section we present Algorithm 4 that is based on the following claim.

CLAIM 1: Given a topological order of the WSC (DAG), TO = {WS_1, …, WS_a, …, WS_b, …, WS_k}, and an unordered pair (WS_a,WS_b), then we can say that it is possible to move

WS_b forward to immediately follow WS_a or to move WS_a backward to immediately proceed WS_b in TO as follows:

FORWARD MOVE: We can move WS_b forward to immediately follow WS_a in TO if and only if there is no WS_i in TO such that

- WS_a proceeds WS_i in TO
- WS_i proceeds WS_b in TO, and
- there is no path in WSC from in WSC from WS_i to WS_b.

BACKWARD MOVE: We can move WS_a backward to immediately proceed WS_b in TO if and only if there is no WS_i in TO such that

- WS_a proceeds WS_i in TO
- WS_i proceeds WS_b in TO, and
- there is no path in WSC from in WSC from WS_a to WS_i.

Algorithm 4 generates Parallel Execution Scenarios (PES) that do not violate the precedence relations in WSC.

Algorithm 4: Optimal-Topological-Order(WSC,TO,$WSP_{optimal}$)

Input:
- TO, a topological ordering of web services in WSC
- $WSP_{optimal}$, all set of parallel optimized pairs

Output:
- Calculating the optimal topological order $TO_{optimal}$

```
1. Optimal-Topological-Order(WSC,TO,WSP_optimal)
2. TO_optimal ← TO
3. for each pair (WS_i, WS_j) in WSP_optimal do
4.    if WS_i can be moved forward in TO_optimal then
5.       // according to forward move definition in claim 1
6.       move WS_i forward to immediately follow WS_j
7.       substitute "WS_i, WS_j" in TO with the pair (WS_i, WS_j)
8.    else
```

```
9.          // according to backward move definition in claim 1
10.       move WSj backward to immediately proceed WSi
11.       substitute "WSi, WSj" in TO with the pair (WSi, WSj)
12.    endif
13.endfor
14.return TOoptimal
```

PROOF OF ALGORITHM 4: The proof is trivial according to **claim 1**.

EXAMPLE 1: Assume that we have the following WSC represented as a DAG in figure 4.1, the corresponding topological order TO = {WS_1 , WS_5 , WS_2 , WS_3 , WS_4 }, and the execution time of each web service, Execution time = {10 ,100 ,20 ,100 ,100}, then the execution time of the WSC can be calculated as follow:

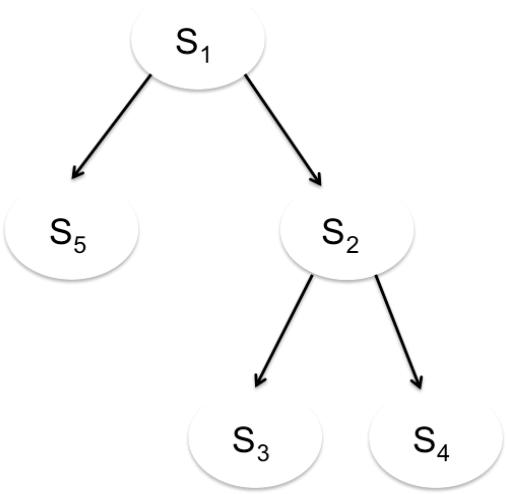

Figure 4.1 DAG Graph of Example 1

The execution time of the topological TO without any enhancements is 10 +100 + 20 + 100 + 100 = 330 units. After applying the concepts of Task 2, the optimal web service replacement ($WSP_{optimal}$) is { (S_5 , S_2), (S_3 , S_4) }; therefore, the topological order with web service pairs is { S_1 , (S_5 , S_2), (S_3 , S_4) }, and the improved running time will be 10 + 100 +100 = 210 units.

IMPROVEMENT OF WSC EXECUTION SCHEDULE AFTER USING THE PAIR-WISE PARALLEL EXECUTION:

In the above example (figure 4.1), the execution time of the web service schedule without any enhancements is 10 +100 + 20 + 100 + 100 = 330 units. The running time after using the WS-pairs approach is { S_1 , (S_5 , S_2), (S_3 , S_4) } = 10 + 100 +100 = 210 units.

Therefore, Speed $up_{overall}$ = $\frac{Execution\ Time\ old}{Execution\ Time\ new}$

Speed $up_{overall}$ = $\frac{330}{210}$

Speed $up_{overall}$ = 1.57143

Therefore, the execution time of web service schedule has improved after using the WS-pairs approach by 57%.

CHAPTER 5

PERFORMANCE IMPROVEMENT: MULTIPLE PARALLEL EXECUTION

In this chapter, we present Task 3 that investigates possible further improvement of WSC execution schedule. In this Task, we improve the execution performance of WSC of Task 2 by applying the following approaches:

5.1 INCREASING THE NUMBER OF WSS EXECUTING IN PARALLEL.

5.2 EXECUTE CONCURRENTLY THOSE WSS THAT HAVE SIMILAR EXECUTION TIME.

5.1 INCREASING THE NUMBER OF WSS EXECUTING PARALLEL

Given a WS-composition and the corresponding WSP, find all arrangements of parallel WS execution $WSS = \{WSS_1, \ldots, WSS_n\}$ such that *WSS* contains all possible execution schedules, where each WSS_i ($i = 1, \ldots, n$) contains sets of concurrent executing services that are not ordered by precedence and do not conflict.

DEFINITION 3: OPTIMAL WEB SERVICE SET (WSS)

Given a set $WSS = \{WSS_1, \ldots, WSS_n\}$, and k number of parallel processors, we say that $WSS_i = \{WS_1, \ldots, WS_l\}$ is optimal if the following conditions hold:

1.) $|WS_j| \leq k$, that is the number of web services in each set is less than or equal to k,

2.) there is no other WSS_m (where, $m = 1, \ldots, n$) that satisfies the first condition and has smaller execution time than WSS_i.

DEFINITION 4: CORRECTNESS CRITERIA

Given a WS-set, (WS_i, … , WS_j) in TO, we can rewrite TO denoted as TO' by replacing (WS_i, … ,WS_j) with WS_i → … → WS_j , where → indicate a precedence in the topological order.

We say that TO is correct if there exists a topological order, TO', of WS-composition, such that

to = TO'

DEFINITION 5:

Let WSP contains all pairs of web services that can execute parallel we say that if there are pairs (WS_i,WS_j), (WS_j,WS_k), and (WS_i,WS_k), then (WS_i,WS_j, WS_k) can execute concurrently.

Algorithm 5 generates the optimal WSS that has WS-sets that can be substitute in the topological order TO.

Algorithm 5: Building WSS

Input:

- **TO**, the BFS topological ordering of web services in **WSC**
- ***WSP***, all possible Web service pairs

Output:

- Generation the optimal (**WSS**)

```
1. Given WSC, WSP
2.  WSS_i = Φ ( i= 1, … , n)
3. for each web service (S_i) in WSC (i.e. each node in WSC) do
4. WSS_i = WSS_i ∪ {(S_i, S_j) | (S_j, S_i) ∈ WSP}
5. end for

6.  set extension function // final parallel sets
7.  for any two sets, S_i, S_j in WSSi, extend S_i as follows:
8.  Let IS := S_i ∩ S_j and
```

```
9.  DS := Sj – Si
10. then for any services sd ∈ DS,
11. Si := Si ∪ {d} if and only if for all services si in Si
there is a
12. pair (si,d) or (d, si) in WSP.
13. Use the new set and repeat until no more changes
14. end for

15. Drop any WSSi that has no sets

16. for each WSSi // limited k number of processors
17.     Drop all sets that have more than K elements where K is
the
18.     number of available processors
19.     For each set in each WSSi, set the weight equal to
execution time 20.     of the longest web service in that set
21. end for

22. Merge all WSSi sets into a new set denoted as WSSall

23. set optimal_set ← empty // optimality part
24. set optimal_weight ← 0
25. set current_optimal_set ← empty
26. set current_optimal_weight ← 0
27. set WSSall' ← WSSall
28. tested_sets ← empty
29. for each set { Si, …, Sk} in WSSall'
30.         Drop all other sets that have Si, …, or Sk from
WSSall'
31.         Find all possible combinations between { Si, …, Sk}
and the 32.         remaining sets in WSSall' such that this
combination is
33.         feasible to be substituted in the topological order

34.         Calculate the weight of each combination
35.         if the least weight is less than
current_optimal_weight then
36.             Set current_optimal_weight equal to weight of
this
37.             combination
38.             Set current_optimal_set equal to this
combination
39.         end if
40.         if current_optimal_weight is less than
optimal_weight then
41.             set optimal_set = current_optimal_set
42.             set optimal_weight = current_optimal_weight
43.         end if
44. add { Si, …, Sk} to tested_sets
45. set current_optimal_set ← empty
46. set current_optimal_weight ← 0
```

```
47. WSS_all' ← WSS_all – sets that have been tested already
48. end for
49. set weight = optimal_weight
50. set WSS = optimal_set
```

THEOREM 3: Algorithm 5 generates

1) (Completeness) all sets of WSs that do not have precedence relation with each other and do not conflict, i.e., generate all parallel execution schedules; and

2) (Soundness) only those sets of WSs are generated that meet the above characteristics, i.e., each set of parallel execution $i \leq k$.

PROOF OF THEOREM 3**:**

(Completeness) Assume, by contradiction, that there is a set S' of parallel execution services such that S' is sound but not in WSS_{all}. Assume that S' contains {(s_i, …, s_j, …)}. Service s_j can run parallel with any other services in S'. But then, for any other service, s_i, there must be a pair (S_i, S_j) (or the equivalent pair (s_j, s_i) in WSP. Therefore, in our initial set WSS_i, (line 6), there must be a pair (s_i,s_j), WSS_i = { … , (s_i,s_j), … }. Since s_j must be able to run concurrently with all other services in S', and by the set extension function, our algorithm must produce S' (lines 6-14). Therefore, if S' is produced, then it must be removed from the final set. However, our algorithm allows the removal of a set only if its size is lager than k (lines 17-18) which is a contradiction of our assumption that S' is a valid and sound selection.

(Soundness) Assume, by contradiction, that there is a set of sets S' contains { …, s_i, …, s_j,…} in WSS_i, such that either

i. there is a precedence relation between services s_i and s_j

or

ii. there is a conflicting operation between s_i and s_j.

PROOF OF SOUNDNESS: trivially follows from the algorithm of WSP and set extension function.

Algorithm 6 generates Parallel Execution Scenarios (PES) that do not violate the precedence relations in WSC by increasing the number of WSs executing concurrently.

Algorithm 6: Optimal-Topological-Order-With-Sets (WSC, TO, WSS)

Input:
- TO, a topological ordering of web services in WSC
- *WSS*, set of optimal WSS

Output:
- Calculating the optimal topological order $\mathbf{TO_c}$

```
1. TOc ← TO
2. for each set WSq in WSS do
3. if  WSq has only two elements {WSi, WSj} then
4.    if WSi can be moved forward in TOc then
5.         // according to forward move definition in claim 1
6.         move WSi forward to immediately follow WSj
7.         substitute "WSi, WSj" in TOc with the set {WSi, WSj}
8.    else
9.         // according to backward move definition in claim 1
10.        move WSj backward to immediately proceed WSi
11.        substitute "WSi, WSj" in TOc with the set {WSi, WSj}
12.   endif
13. elseif WSq has more than two elements {WSi, WSj, WSt, …} then
14.       if WSj can be moved forward to immediately follow WSi in TOc
then
15.                if WSt can be moved forward to immediately follow
WSj then
16.                   // according to forward move definition in
claim 1
17.                   move WSj forward to immediately follow WSi
18.                   move WSt forward to immediately follow WSj
19.                    substitute "WSi, WSj, WSt,…" in TOc with the set {WSi, WSj,
WSt, …}
20.             else
21.                    // according to backward move definition
in claim 1
22.                    move WSi backward to immediately proceed
WSj
23.                   if WSt can be moved forward to immediately follow WSj
```

```
then
24.              move WS_i backward to immediately proceed WS_j
25.              move WS_t forward to immediately follow WS_j
26.                substitute "WS_i, WS_j, WS_t,…" in TO_c with the set {WS_i, WS_j, WS_t, …}
27.                      else
28.              move WS_j backward to immediately proceed WS_t
29.              move WS_i backward to immediately proceed WS_j
30.              substitute "WS_i, WS_j, WS_t,…" in TO_c with the set {WS_i, WS_j, WS_t, …}
31.                      endif
32.                 endif
33.       endif
34. else // WSS is empty
35.      return TO_c
36. endif
37. endfor
38. return TO_c
```

PROOF OF ALGORITHM 6: The proof is trivial according to claim 1 (Chapter 4).

EXAMPLE 2: Given the WSC and the corresponding topological order (TO) in figure 4.1, the execution time of Task 2 can be improved by applying the concepts of Task 3 as follows:

The optimal Web Service Set (WSS) =$\{(S_5 , S_3 , S_4)\}$. Therefore, the topological order with web service sets (WSS) is =$\{ S_1, S_2, (S_5 , S_3 , S_4) \}$ and the improved running time will be = 10 + 20 + 100 = 130 units.

IMPROVEMENT OF WSC EXECUTION SCHEDULE AFTER INCREASING THE NUMBER OF WSS THAT EXECUTE CONCURRENTLY:

Using the approach of WS-sets, the running time of the web service schedule can be improved since the new execution time = 10 + 20 + 100 = 130 units. Therefore, we can compare the execution time after using the WSS approach to the execution time without any enhancement as follows:

$$\text{Speed up}_{\text{overall}} = \frac{Execution\ Time\ old}{Execution\ Time\ new}$$

$$\text{Speed up}_{\text{overall}} = \frac{330}{130}$$

$$\text{Speed up}_{\text{overall}} = 2.5385$$

Therefore, the execution time of web service schedule has improved 153% after using the WS-sets approach.

5.2 Execute Concurrently Those WSs that Have Similar Execution Time

Given a WSC, the corresponding WSS, and the execution time for each WS, i.e., (WS_i, t_i), where t_i is the execution time of web service WS_i, find all arrangements of parallel WS execution $WSS = \{WSS_1, \dots, WSS_n\}$ such that *WSS* contains all possible execution schedules, where each WSS_i (i = 1, … , n) contains sets of concurrent executing services that are not ordered by precedence, do not conflict, and have similar execution time.

Definition 6: Optimal Web Service Set (WSS) with similar execution time

Given a set $\boldsymbol{WSS} = \{WSS_1, \dots, WSS_n\}$, and k number of parallel processors, we say that $WSS_i = \{WS_1, \dots, WS_l\}$ is optimal if the following conditions hold:

1.) $|WS_j| \leq k$, that is the number of web services in each set is less than or equal to k,

2.) there is no other WSS_m (where, m = 1, … , n) that satisfies the first condition and has smaller threshold value than WSS_i.

Algorithm 5 can be extended to generate the Optimal Web Service Set (WSS) with similar execution time that can be substitute in the topological order TO as follows:

We define a threshold value, optimal_threshold, and require that the optimal WSS is the one with the least total threshold values.

Extension of Algorithm 5

```
23. set optimal_set ← empty
24. set optimal_ threshold ← 0
25. set current_optimal_set ← empty
26. set current_optimal_ threshold ← 0
27. set WSS_all' ← WSS_all
28. set tested_sets ← empty
29. for each set { Si, …, Sk} in WSS_all'
30.     set threshold of this set equal to the difference
31.     between the execution times of the largest and
32.     smallest WSs
33.     Drop all other sets that have Si, …, or Sk from WSS_all'
34.     Find all possible combinations between { Si, …, Sk} and the
35.     remaining sets in WSS_all' such that this combination is
36.     feasible to be substituted in the topological order
37.     Calculate the weight of each combination
38.        if the least threshold is less than
current_optimal_threshold then
39.            Set current_optimal_threshold equal to threshold of
this
40.            combination
41.            Set current_optimal_set equal to this combination
42.        end if
43.        if current_optimal_threshold is less than optimal_threshold
then
44.            set optimal_set = current_optimal_set
45.            set optimal_threshold = current_optimal_threshold
46.        end if
47. add { Si, …, Sk} to tested_sets
48. set current_optimal_set ← empty
49. set current_optimal_ threshold ← 0
50. WSS_all' ← WSS_all – sets that have been tested already
51. end for
52. set threshold = optimal_threshold
53. set WSS = optimal_set
```

EXAMPLE 3: In the above example, WS_5 , WS_3 , and WS_4 have similar execution times, 100 units; therefore, this will give us the least possible threshold when executing those WSs together, the total execution time = 10 + 20 + 100 = 130 units.

IMPROVEMENT OF WSC EXECUTION SCHEDULE AFTER EXECUTING CONCURRENTLY THOSE WSS THAT HAVE SIMILAR EXECUTION TIME:

Using the approach of executing concurrently WSs that have similar execution time, the running time of the web service schedule can be improved since the new execution time = 10 + 20 + 100 = 130 units. Therefore,

$$\text{Speed up}_{\text{overall}} = \frac{Execution\ Time\ old}{Execution\ Time\ new}$$

$$\text{Speed up}_{\text{overall}} = \frac{330}{130}$$

$$\text{Speed up}_{\text{overall}} = 2.5385$$

The execution time of web service schedule has improved 153% after using the approach of executing concurrently those WSs that have similar execution time.

CHAPTER 6

CONCLUSION

In this thesis we have presented improvement of web service composition (WSC) execution time using pair-wise and multiple parallel execution schedules. We modeled a WSC as WS-BPEL specification, i.e., a partial order of WS transactions, thus allowing some of the WS transactions in the WSC to execute concurrently. We analyzed the WS-BPEL specification of the WSCs to identify WS transactions that can execute parallel.

In chapter 4, we identified subgroups of WSs that can run concurrently without potential violation of consistency of the database. Our algorithm generated an optimal set of WS-Pairs ($WSP_{optimal}$) and substituted web services with concurrently executing web services pairs in the Parallel Execution Scenarios (PES), thus reducing the total execution time of the WSC.

In chapter 5, we investigated possible further improvement of WSC execution schedule. We increased the number of WSs that could execute concurrently, and we provided further improvement by scheduling services with similar execution time to run concurrently. In each case we generated optimal web service schedules, called Parallel Execution Scenarios (PES), thus further improving the total execution time of the WSC.

For future work, we will study possible further improvement of WSC execution schedule by balancing the workload of concurrent services over available processors. In addition, we will investigate the possibility of improving upon traditional locking mechanism for WSC to guarantee database consistency while avoiding unacceptable performance degradation.

BIBLIOGRAPHY

Al-Jumaha, N.B., Hassaneinb, H.S., & El-Sharkawia M. (2002). Implementation and modeling of two-phase locking concurrency control—a performance study. *Information and Software Technology, 42* (2000), 257–273.

Bhargava, B. (1999, Jan/Feb). Concurrency control in database systems. *IEEE Transactions on Knowledge and Data Engineering, 11*(1), 3-16.

Ezenwoye, O. and S.M. Sadjadi, Composing aggregate web services in BPEL, Proc. 44th ACM Southeast Conference (ACMSE), March 2006, Melbourne, FL.

Juric, M. B., Business Process Execution Language for Web Services. 2nd Edition. Packt Publishing, 2006.

Kung, H. T., & Robinson, J. T. (1981, June). On optimistic methods for concurrency control. *ACM Transactions on Database Systems, 6*(2), 213-226.

Le, Gao, Susan Darling Urban, Janani Ramachandran: A survey of transactional issues for Web Service composition and recovery. IJWGS 7(4): 331-356 (2011).

Lechtenbörger, Jens: Two-Phase Commit Protocol. Encyclopedia of Database Systems 2009: 3209-3213.

Sheikhan, Mansour, Mohsen Rohani, and Saeed Ahmadluei: A neural-based concurrency control algorithm for database systems. *Neural Computing and Applications 22*(1): 161-174 (2013).

Strandenaes, Thomas, and Randi Karlsen, Transaction Compensation in Web Services, IEEE 2003.

Thomasian, A. (1998). Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys, 30(*1), 70-119.

Wang, Jinling, Beihong Jin, and Jing Li: A Transaction Model for Long Running Business Processes. ICEIS (1) 2004: 267-274.

Zeshan Furkh and Radziah Mohamad, " Semantic Web Service Composition Approaches: Overview and Limitations", International Journal on New Computer Architectures and Their Applications (IJNCAA) 1(3): 64The Society of Digital Information and Wireless Communications, 2011 (ISSN: 2220-9085), pp 640-651.