D-040

# Automatic determination of compatibility of method invocations in object-oriented database systems

*Khamisi KALEGELE* † *Kouji HIRATA* † *Yoshinobu HIGAMI* † *Shin-ya KOBAYASHI* †

## 1. Introduction

Transaction is a mechanism by which users communicate with shared resources. From user viewpoint, it is defined as a request/reply unit expressed in the form of a source program and from system viewpoint it is defined as a sequence of operations (reads, writes etc) on computable objects. Operations on shared resources (database) must always be executed in the framework of a transaction. A correct transaction is characterized by ACID properties (Atomicity, Consistency, Isolation, and Durability). These are crucial for the general consistency of the database. Isolation property is specifically responsible for making concurrent executions possible whereby each transaction is isolated from the other during execution [2]. Isolation is enforced by Concurrency Control (CC) Mechanisms, commonly presented in three categories; Lock-based, Time stamping and Optimistic. Their description is not part of this paper.

At some stage these protocols seek to explore the semantics of transaction operations in order to increase concurrency either by increasing effective usage of locks or by avoiding concurrent execution of conflicting operations etc.

### 1.1. Motivation and purpose

Although relational databases are by far the most commonly used databases today, Object-Oriented Database Systems (OODB Systems) theoretically still hold a better promise of providing greater opportunities for supporting semantic-based concurrency control [1]. This is because of a number of reasons, first is the fact that the capability of including user-defined operations of arbitrary complexity in data object representation provides greater semantic information about the operations that can be exploited for CC. Second, because of the encapsulation mechanism of Object-Oriented (OO) models, operations defined on a data object provide the only means to access the object's data. Thus, data contention can occur only among operation invocations within the object. This characteristic of OO data models provides greater flexibility for CC in that it allows concurrency control specific to individual data objects.

According to [3], the required explicit specification of the Method Compatibility Table (MCT) needed for the exploitation of operation semantics can be achieved by commutativity. For instance, in a Queue object, enqueing the same item by two concurrent transactions is not a conflict because the order of these updates is insignificant in the sense that neither it can be observed by the two executions of the enqueue method nor by any method on queues that may be invoked later. The two enqueue operations are said to be commutative with respect to each other. This kind of commutativity is very general because only semantics of the operations are considered. Sometimes this general commutativity is relaxed by also considering the state of an object to enhance concurrency. This is referred to as state-based commutativity [9].

In practice object classes in an OODB are much bigger and complex than a simple Queue. Thus, it is very tedious for the object creator to look into the semantics of the class and determine whether two methods commute or not in order to increase concurrency.

In this paper we present our efforts in devising a way of establishing practical compatibility in terms of achieving better concurrency. We propose the use of affected set of attributes called AMS (Access Mode Set) to automate the process of determining commutativity.

### 1.2. Basic Assumptions

Although CC algorithms need better be designed with the general commutativity in mind, according to [6], state-based commutativity potentially offers more CC but incurs more overhead since the scheduler needs to know intermediate state preceding the two operations whose commutativity is under question. In this research, we assume state-based commutativity for greater method-invocation compatibility because we think that the overhead is reduced by the fact that in OODB we can have CC specific to individual objects.

Ideally, encapsulated objects are accessed only through explicitly defined interfaces. However, we believe bypassing these interfaces in some cases, like when making object-oriented code concurrent, is inevitable. We assume that enforcement of encapsulation is somehow relaxed in OODB as well. The same supposition is demanded by [5] where programming view of encapsulation have been differentiated from the database adaptation of that view. It was found in that work that there are some cases where encapsulation is either not needed or its enforcement need to be reduced.

## 2. Example OODB Object Class

We set forth explaining how to determine compatibility, using our approach, with flight Class shown in Fig. 1. The class provides 2 methods for booking economy class (EC) and business class (BC) seats, 2 methods for booking either class seat and if that class seats are sold-out then book the other class seat, and 2 classes for cancelling each class seat booking.

## 3. Compatibility of Method invocation

Transactions in OODB are modeled as sets of method invocations on objects. Concurrency level is subjected to the level at which object methods are concurrently invoked. This comes down to the effects each method has on the state of an object as presented by its attributes. In our approach, we employ dynamic monitoring of a set of these object state-attributes in terms of modes at which they are accessed. We call this set AMS (Access Mode Set) and is defined in section 3.1. Generally there are two types of attributes; primitive and Object (user defined) attributes. For primitive attributes, we use 'R ', 'W' and 'N' for 'READ', 'WRITE' and 'NULL' respectively as access modes. Since object-types attributes are user defined, their access modes are object-dependent. In our flight class, *passenger* is an object-type attribute and we assume it has 'A' for ADD, 'D' for DELETE, 'R' for READ and 'N' for NULL as access modes.

In case of multiple access modes, the most restrictive access mode takes precedence. From the basics of correctness of concurrent executions, write operation is more restrictive than a read operation for primitive attributes. Therefore precedence order is 'W' > 'R' > 'N'. Our method requires that an object

†Department of Electrical and Electronic Engineering and Computer Science, Graduate School of Science and Engineering, Ehime University

```
class flight{
private:  int e, b;  /*no. of EC seats and BC seats.*/
          list<person> passenger;  /*passenger list.*/
public:  boolean E(person psg); /*book EC only.*/
         boolean EB(person psg); /*if EC is sold-out book BC.*/
         boolean B(person psg);  /*book BC only.*/
         boolean BE(person psg);  /*if BC is sold-out book EC.*/
         boolean cancelE(person psg);  /*cancel EC booking.*/
         boolean cancelB(person psg);  /*cancel BC booking.*/
}
//implementation
boolean E(person psg){
  if (e< 40) then{
    e=e+1;  passenger.insert(psg,'E');
    return TRUE;
  }
   else return FALSE;
}
boolean EB(person psg){
  if (E(psg)) then return TRUE
  else{
    if(B(psg)) then return TRUE
    else  return FALSE;
  }
}
boolean B(person psg) {
  if (b< 10) then{
    b=b+1 ; passenger.insert(psg,'B');
    return TRUE;
  }
  else return FALSE;
}
boolean BE(person psg){
  if (B(psg)) then return TRUE
  else{
    if(E(psg)) then return TRUE
    else  return FALSE;
  }
}
void cancelE(person psg){e=e-1;passenger.delete(psg) }

void cancelB(person psg){b=b-1; passenger.delete(psg)}
```

*NB: The flags 'E' and 'B' serve to indicate booked class. And numbers 40 and 10 stands for maximum numbers of EC and BC seats.*

Fig. 1: Flight class

creator defines precedence for the access modes for object-type attributes. An additional access mode 'E', for EXCLUSIVE, is provided for cases where it is impossible or ambiguous to define restrictiveness. For instance, it is a little vague about restrictiveness of access modes 'A' and 'D' in a *passenger* object. So in case of multiple access modes ('A' and 'D') on *passenger* object we employ 'E' mode.

Compatibility is determined in two steps. We explain in section 3.1 how to establish AMS as a first step of our approach. Here we present a simple algorithm used to get AMS. In section 3.2, we explain method invocation conflicts using AMSs as a second step of our approach.

### 3.1. Determination of AMS

Fig. 2 shows an algorithm for determining AMS.

We use a simplified CFG (Control Flow Graph) to determine possible execution flows. For flight class in Fig. 1, CFG is shown in Fig. 3. A root node is the flight class with methods as its children. Down after this level, every control structure constitutes

*Algorithm for Determining AMS(M) of method M*

1. Declare AMS as a set **a** of ordered Access Modes $a^i$ for i=1~N written as $a^1a^2\ldots a^N$ where N=Number of State-Attributes
2. For each possible execution flow of Method M, establish
   2.1. $AMS(M_p)$ for primitive state-attributes only.
   2.2. $AMS(M_n)$ for nested invoked methods only.
   2.3. $AMS(M_o)$ for object-type attributes only.
3. Combine $AMS(M_p)$, $AMS(M_n)$ and $AMS(M_o)$ by considering the most restrictive access mode to get $AMS(M_f)$ for each possible execution flow.
4. Combine $AMS(M_f)$ by considering the most restrictive access mode to get AMS(M)

Fig. 2: Algorithm for determining AMS

a node. Children to any control structure will correspond to every possible evaluation of that control structure.

Now, using the algorithm in Fig. 2, AMS of any method can be established as follows;

For method M=EB (*psg*);

1. We declare AMS(M) as a set $a^1a^2a^3$ where $a^1$, $a^2$ and $a^3$ are access modes of *e*, *b*, and *passenger* respectively.
2. M=EB(*psg*) has 3 possible execution flows (c),(d) & (e) from Fig. 3. We proceed as follows;

| | (c) | (d) | (e) |
|---|---|---|---|
| $AMS(M_p)$ | NNN | NNN | NNN |
| $AMS(M_n)$ | WNA | RNN | RNN |
| | | NWA | NRN |
| $AMS(M_o)$ | NNN | NNN | NNN |
| $AMS(M_f)$ | WNA | RWA | RRN |

*Note that there are two $AMS(M_n)$s for each execution flows (d) and (e) because there are two nested invoked methods for each flow.*

3. This is shown in the last row of the table in step 2.
4. Lastly we combine $AMS(M_f)$ of the three possible execution flows.

| | |
|---|---|
| $AMS(M_f)$ of Execution flow (c) | WNA |
| $AMS(M_f)$ of Execution flow (d) | RWA |
| $AMS(M_f)$ of Execution flow (e) | RRN |
| Effective AMS(M=EB(*person*)) | WWA |

Shown below in Table 1 are AMSs for all methods of a flight class.

Table 1: AMSs for flight class

| **Method** | E() | EB() | B() | BE() | cancelE() | cancelB() |
|---|---|---|---|---|---|---|
| **AMS** | WNA | WWA | NWA | WWA | WND | NWD |

### 3.2. Invocation Compatibility

As mentioned in section 1.2, we use the criteria of state-based commutativity to establish compatibility. To test the effects of invocations, AMSs are compared item-wise for conflicting access modes. Below is how we define compatibility of access modes for flight class. We assume the definition of access modes conflicts is part of database metadata. For our flight class, we set up the following definition.

*The ordered pair $\alpha^1\alpha^2$ of access modes of an attribute by two different invocations is conflicting only for the following set of ordered pairs; {WR, WW, AR, DR, E$\alpha$'} where $\alpha$' is any access mode.*

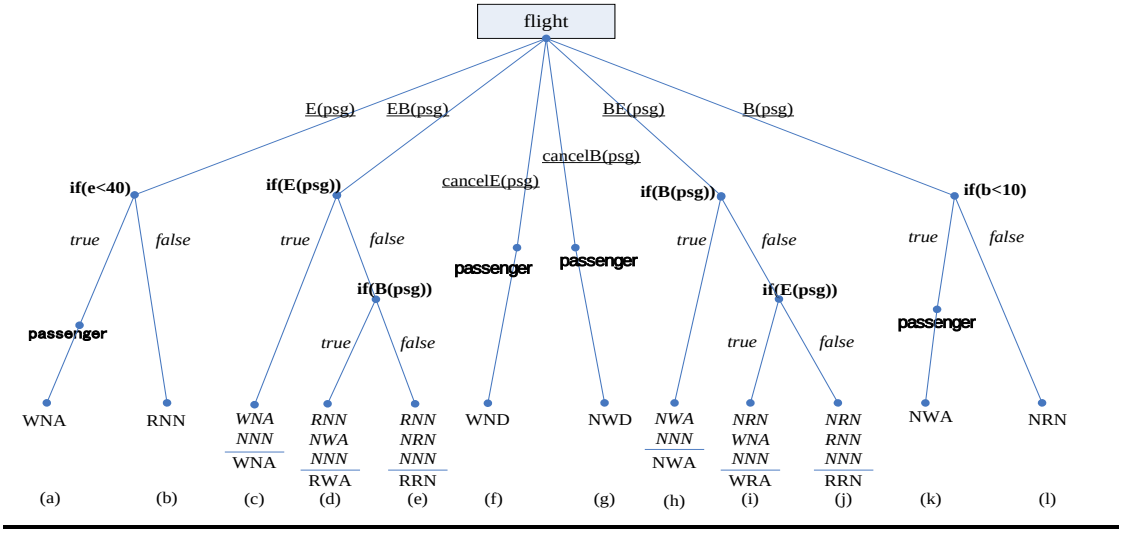


Fig. 3. Simplified CFG showing AMSs

Table 2: Method Compatibility Table (MCT) for the flight class

| | | Ongoing Execution | | | | | |
|---|---|---|---|---|---|---|---|
| | | E() (WNA) | EB() (WWA) | B() (NWA) | BE() (WWA) | cancelE() (WND) | cancelB() (NWD) |
| Requested Invocation | E() - (WNA) | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |
| | EB() - (WWA) | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ |
| | B() - (NWA) | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ |
| | BE()- (WWA) | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ |
| | cancelE() - (WND) | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |
| | cancelB() - (NWD) | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ |

Using $\mathcal{N}$ for Non-conflicting and $\mathcal{Y}$ for conflicting modes, the following can be depicted from the above definition; $WR \rightarrow \mathcal{Y}$, $WW \rightarrow \mathcal{Y}$, $WE \rightarrow \mathcal{N}$, $AD \rightarrow \mathcal{N}$ etc.

Now, AMSs are compatible only if the results of all item wise comparisons are not conflicting. Without considering dynamic AMSs, from Table 1 the complete MCT for the flight class will be as shown in Table 2. Note here that, among other observations, ongoing execution of EB() is not compatible with any other method. However it is possible to achieve some compatibility by using dynamic AMSs derived from Fig. 3.

Commonly, semantics have been exploited to increase concurrency. Likewise, in our approach of determining compatibility, we explore control structures to get more compatibility. Depending on evaluation of control structures, the access modes of some state-attributes might change. We take this into consideration and dynamically update AMS of an ongoing execution.

Dynamically, at any intermediate node in Fig. 3, AMS is established as a combination of all other AMSs down all the children of that node. For example before "if(B(*psg*))" in method EB(*psg*). Effective AMS is as shown below;

| | |
|---|---|
| AMS at leaf (d) | RWA |
| AMS at leaf (e) | RRN |
| Effective AMS(EB(*person*)) | RWA |

At any point in time, the execution of EB() can be in one of five stages corresponding to 5 edges in the CFG(Fig. 4).We show below, in Table 3, AMSs of these five states and respective compatibilities.

Table 3: MCT for ongoing execution of EB()

| | | Ongoing Execution of EB() | | | | |
|---|---|---|---|---|---|---|
| | | 1 (WWA) | 2 (WNA) | 3 (RWA) | 4 (RWA) | 5 (RRN) |
| Requested Invocation | E() - (WNA) | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NNN}=\emptyset$ |
| | EB() - (WWA) | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |
| | B() - (NWA) | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |
| | BE()- (WWA) | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |
| | cancelE() - (WND) | $\mathcal{YYN}=\mathcal{X}$ | $\mathcal{YNN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NNN}=\emptyset$ |
| | cancelB() - (NWD) | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NYN}=\mathcal{X}$ | $\mathcal{NNN}=\emptyset$ |

## 4. Evaluation

Table 3 shows possibilities for compatibility for a method {EB()} which without dynamic compatibility was not compatible with any other method. Needless to say, there are some potential in this approach as far as concurrency is concerned.

If we define $P(M/EB^0)$ as probability of compatibility of method M when method EB() is ongoing, then;

$$P(M/EB^0) = \frac{\text{No.of Compatible Methods when EB is ongoing}}{\text{Total No. of Methods}}.$$

From Table 2, we get $P(M/EB^0) = 0.$

After considering dynamic compatibility $P(M/EB^0)$ becomes;

$$= \sum_{i}^{\text{No. of Stages}} p_i \frac{\text{No. of Compatible Methods when EB at stage } i}{\text{Total No. of Methods}},$$

where $p_i$ is probability that EB is at stage *i*.

Therefore with dynamic compatibility, from Table 3, we get

$$P(M/EB^0) = \frac{0 + 2p_2 + 2p_3 + 2p_4 + 6p_5}{6}.$$

Practically, $p_i$ is time- dependent value and very difficult to establish. For simplicity if $p_i = p(t)$, progress probability of method EB() at execution time *t*, then

$$P(M/EB^0) = 2p(t).$$

Evidently, although $p(t)$ is a complex function, it is a non negative function. Thus, $P(M/EB^0)$ is always likely to be

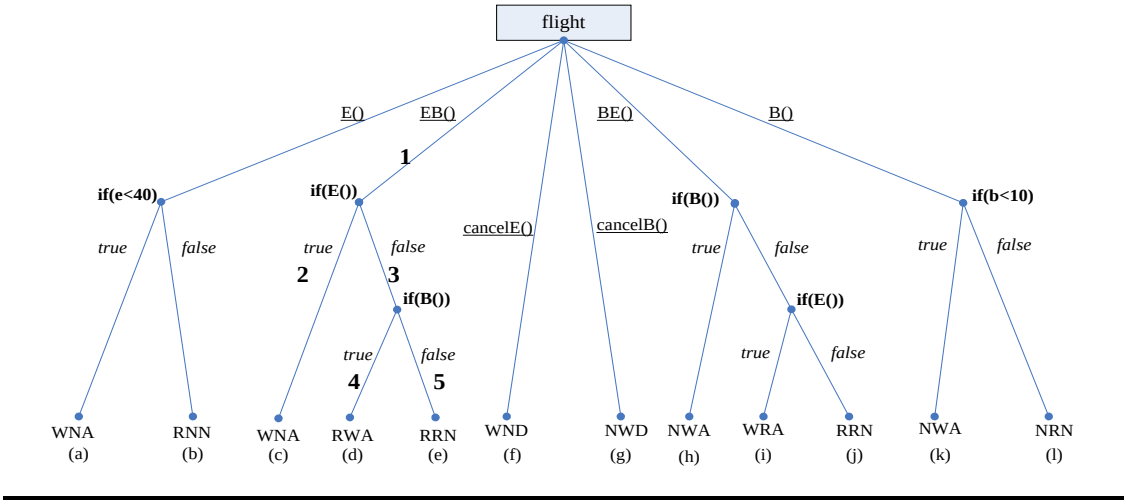


Fig. 4: Five execution stages of method EB()

higher when considering dynamic compatibility. Note that the purpose of this representation is just to emphasize a point made earlier that there are some potential in this approach. This is not a generic evaluation.

Before evaluating further performance benefits which can be achieved from this approach, there are some issues which need be looked upon. As pointed out in section 1.2, in some cases, bypassing encapsulation interfaces is inevitable in order to take advantage of OO. Since there is a clear tradeoff between concurrency and breach of OO concept, it is important that the extent by which encapsulation can be relaxed without compromising the meaning of OO be known.

Also due to the fact that in this approach, invocation compatibility relies on the definition of conflicting access modes which subsequently is based on the update type, evaluation of this approach must be specific to a CC mechanism. The commonest update types employed in CC mechanisms are In-place and deferred Updates.

Moreover, the increased possibilities for compatibility in Table 3 are all falling within a quite small window of ongoing execution of method EB(). With increasing processor speeds, it is fair to say that this window might not be long enough for the effective exploitation of these possibilities. Of course, unless this approach is used in long-transaction systems, it does not hold much promise in increasing concurrency.

## 5. Conclusion and Future work

The approach aims at using concepts of commutativity which are increasingly successful in compiler optimization [7, 8]. We presented an automatic way to establish whether two method invocations on an OODB object are compatible and thus a means to increase transaction processing concurrency can be sought. This means is CC mechanism-based. We have demonstrated this approach against a flight class by showing possibilities of compatibilities where there was none. Particularly for the demonstrated case, the possibility is up to 100% depending on the execution stage at which the ongoing execution is.

This approach of automatic determination of compatibility of method invocations is at a very preliminary stage. It seems to place much burden on object creator for complex objects. Among other complicated tasks is to define access mode precedence for object-type attributes.

Obviously future work in this line is finding better way of handling access modes of user-defined object-types attributes. Another unaddressed issue is the issue of nested transaction. Since OODB is modeled in terms of method invocations, nested transactions imply nested series of method invocations. In the mean time we feel that the best way of handling nested invocation is CC protocol dependent. This is also supported by [4] in which the transaction and sub transaction relationship seems to depend more on the way the two commit and abort. Also concerning state-base commutativity, there is an issue of tradeoff between increased concurrency and increased overhead on the scheduler.

## References

[1] J. Lee and S. H. Son, “Semantic-Based Concurrency Control for Object-Oriented Database Systems Supporting Real-Time Applications,” *6th IEEE Euromicro Workshop on Real-Time Systems*, Vaesteraas, Sweden, July 1994, pp. 156-161.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*, 5th ed. Harlow: Addison-Wesley, 2003.

[3] P. Muth et al., “Semantic Concurrency Control in Object-Oriented Database Systems,” *in Proc. Int'l Conf. Data Eng. (ICDE '93), April, 1993*, pp. 233-242.

[4] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann, 1997.

[5] M. P. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier and S. Zdonik, “The Object-Oriented Database System Manifesto,” in *Proc. First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December, 1989, pp. 40-57.

[6] C. Beeri, P. A. Bernstein, N. Goodman, Lai, M. Y, and D. E. Shasha, “A Concurrency Control Theory for Nested Transactions,” in *Proc. 1983 Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August, 1983, pp. 45–62.

[7] M. Diniz, and P. Diniz, “Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers,” *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, November, 1997, pp 1-47.

[8] F. Aleen, and N. Clark, “Commutativity Analysis for software parallelization: Letting Program Transformation see the Big Picture,” in *Proc. the 14th international conference on Architectural support for programming languages and operating systems*, Washington, DC, USA, March, 2009, pp 241-252.

[9] W. Gerhard and V. Gottfried. *Transactional information systems: Theory, Algorithms, and the Practice of Concurrency Control and recovery*. Elsevier Science and Technology Books: Morgan Kaufmann, May, 2001.