November 1994

A practical method for realizing semantics-based concurrency control

Roger Barga
Calton Pu
Wenwey Hseush

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
http://digitalcommons.ohsu.edu/csetech/342

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
A Practical Method for Realizing Semantics-Based Concurrency Control

Roger Barga, Calton Pu
Department of Computer Science and Engineering
Oregon Graduate Institute
P.O. Box 91000
Portland, OR 97291-1000
email: bargar,calton@cse.ogi.edu

Wenwey Hseush
Department of Computer Science
Columbia University
New York, NY 10027

November 8, 1994

Abstract

Semantics-based concurrency control (SBCC) protocols promise high concurrency but their dependency on application semantics makes them difficult to implement in practice. In this paper we describe a method to systematically implement and combine SBCC protocols by modularly extending conventional on-line transaction processing (TP) systems. The main idea is to use a semantic compatibility function to capture the added compatibility of otherwise conflicting operations from SBCC protocols. The semantic compatibility function and other similar extensions are added to a TP architecture that originally knows only read/write two-phase locking. The resulting system supports the combination of a variety of representative SBCC protocols including: commutativity [?], recoverability [?], cooperative serializability [?], and epsilon serializability [?]. The method is explained using Gray and Reuter’s TP architecture [?], which is specific enough to map into an implementation on commercial TP monitors such as Transarc Encina.
1 Introduction

Most commercial on-line transaction processing (OLTP) and database management systems that support OLTP use two-phase locking (2PL) for concurrency control. 2PL guarantees serializability [?] and in a low contention environment offers good performance with low overhead. Despite the advantages of 2PL, in many applications serializability unnecessarily restricts concurrency. This problem is exacerbated when system hardware performance grows quickly, pushing the envelope imposed by data contention. One way to alleviate this bottleneck is using application semantics in semantics-based concurrency control (SBCC) protocols (e.g., [?, ?, ?, ?]).

Despite the promise of SBCC protocols to improve concurrency, the lack of actual implementations has prevented the full realization of their potential. Few practical implementation details are presented in the literature — indeed, the bulk of the existing research in SBCC has, for the most part, been directed at algorithmic development or simulation studies. In the few cases where implementation details have been considered, the implementation is specialized to the exclusion of other SBCC protocols. There is an implicit assumption that each application can only use one SBCC protocol. Yet without an understanding of how to implement and combine SBCC protocols their potential for increased concurrency remains limited. In this paper, we address these issues, leading to the practical and integrated implementation of several representative SBCC protocols in a generic 2PL environment.

The main contribution of this paper is the Integrated Semantics Extension (ISE) Method for realizing SBCC protocols. The objective of the methodology is the implementation of multiple SBCC protocols through practical extensions to a modern transaction processing (TP) architecture. There are many applications where transaction or application semantics can be used to achieve more concurrency. We demonstrate the power of the ISE method by applying it to two practical data semantics based concurrency control protocols, i.e., commutativity and recoverability, and one transaction semantics based concurrency control protocol, i.e., cooperative serializability. These SBCC protocols are then combined with support for Epsilon Serializability (ESR) [?, ?, ?], a generalization and relaxation of serializability that explicitly allows a limited amount of inconsistency.

Although we do not introduce yet another new SBCC protocol in this paper, ISE is a significant method both for the new capabilities it offers and the feasibility of its practical implementation. On the functionality side, ISE makes it straightforward to design and add a variety of SBCC protocols to a TP system. Furthermore and perhaps more importantly, these protocols (and ESR) can be used in any combination as chosen by the application designers at run-time, at the fine-granularity of individual transactions. On the practical side, ISE is described using Gray and Reuter’s TP architecture [?]. Their modular decomposition of TP systems is accepted by the community as practical and comprehensive. For example, most commercial TP systems, whether
a TP monitor such as Tuxedo and Encina or database management systems such as Oracle and Informix, use two-phase locking concurrency control as they describe. Further, virtually all commercial systems contain all the functionality described in the architecture.

The remainder of the paper is organized as follows. In Section 2, we introduce the ISE method and outline the TP architecture we use to discuss implementation. Section 3 illustrates the application of the ISE method by applying it to a simple SBCC protocol based on operation commutativity [?]. In Section 4 we apply the ISE method to a more powerful SBCC protocol based on recoverability [?], and in Section 5 we apply the ISE method to an SBCC protocol based on transaction semantics: cooperative serializability [?, ?]. In Section 6, we combine these SBCC protocols with ESR. In Section 7, we explain the concrete implementation of SBCC protocols described in Sections 3, 4, 5, 6 by a detailed mapping of extension components into the algorithms and data structures of practical TP systems [?], and describe the implementation of extension components in the commercial TP system Encina. Section 8 concludes the paper.

2 The ISE Method — A Systematic Approach

2.1 A General Description of ISE

Semantic information is available at several levels in the execution of a transaction. The first is the data level, where object access semantics beyond read and write are considered. Many data objects support operations with richer semantics, for example, the incrementing and decrementing of a bank account. The commutativity property of Withdraw and Deposit operations helps the system achieve higher performance by allowing them to run concurrently, in situations where Read and Write could not. The second is the transaction level, where semantics of transaction cooperation, dependencies, and operation interleaving can be specified as well. For example, cooperative serializability [?, ?] uses explicit semantic information to permit conflicting operations to run concurrently, as long as the transactions that issued the conflicting operations are in the same cooperative transaction group. This supports collaborative work, where the exchange of intermediate information is desirable and necessary. A bank customer waiting for an account balance or activity summary at an automatic teller machine would not be delayed if the request was issued as a cooperative transaction to other transactions posting interest or auditing accounts.

Finally, at the application level, semantic information about the requirements of the application that issued the transaction is another rich source of information that can be used to achieve higher concurrency. For example, if the application can tolerate a limited amount of inconsistency in the result, then this information can be used to allow conflicting operations to execute concurrently as long as the total inconsistency is below the specified limit. A bank officer requiring branch
balance information accurate to within $\pm 10,000$ could issue such a transaction during times of peak customer activity. Other sources of semantic information exist, and various SBCC protocols, such as chopping transactions \[?] and escrow transactional method \[?], exploit them for higher concurrency, but their implementation is described separately \[?].

Fundamental to the ISE method is the notion of conflict — incompatibility between operations or transactions. Conflicts may be defined in terms of read and write \[?] (abbreviated as R/W) or as in ACTA, using application semantic information \[?, ?, ?, ?]. We observe that the basis of many SBCC protocols is the introduction of their own notion of conflict, which is weaker than R/W and so allow more concurrency. The challenge that ISE must meet is to include systematically these new notions of conflict in practical TP systems. This paper significantly extends our previous work on extending the classic notion of conflict in the implementation of divergence control algorithms \[?] to support ESR. Since ESR is complementary to SBCC, we will show that both can be implemented through modular extensions in the definitions of conflicts.

SBCC protocols are usually proposed as algorithms with semantics-based operations in their application program interface. For example, transactions may request object access through operation-specific or semantic locks. In contrast, a conventional OLTP system only supports R/W locks with the usual conflict definition. The ISE method bridges this gap by first translating a request for a higher-level primitive to an appropriate read or write lock. R/W (and W/W) conflicts are then analyzed further using both SBCC and ESR definitions. Conceptually, this is a generalization of the extension and relaxation methodology introduced in \[?].

The ISE method can be summarized in three steps:

1. **(Analysis)** – analyze the SBCC protocol to determine how it redefines operation and transaction conflicts.

2. **(Extension)** – analyze the TP system to identify where modular extensions are needed, for example, places where conflicts are detected and resolved. This step is facilitated by modern modular TP monitors such as Encina.

3. **(Relaxation)** – represent the semantic definition of conflicts in a uniform manner and integrate it into the TP system.

In divergence control algorithms, the extension to the TP system is localized to the place where R/W conflicts are detected and resolved. Understandably, different SBCC protocols may require additional information before or after conflicts are detected. Each time such a situation arises, we apply the above three steps to extend and relax the TP system appropriately. In a modular approach, ISE touches on a few well defined places in the TP system being extended. In order to discuss implementation details, we need to establish a common understanding of the
TP mechanisms involved. For this purpose we have selected a generic TP architecture on which to base our discussion.

2.2 Gray and Reuter’s Architecture

We have chosen Gray and Reuter’s transaction processing reference architecture [?] (abbreviated as “GR Architecture”) as the basis of our integrated implementation. The GR Architecture is abstract enough to allow observations on TP systems in general, and yet it is concrete enough to make implementation details obvious in a modern TP monitor such as Transarc’s Encina. We share the same assumptions made by the GR Architecture such as two-phase locking (2PL) concurrency control and write-ahead logging recovery. These assumptions are prevalent in existing TP monitors and database management systems (DBMS).

The major features of the GR Architecture with respect to concurrency control are related to the conflict detection mechanism and lock acquisition in support of two-phase locking. Hence, we focus our description on the interaction among four components: a Transaction Manager, a Lock Manager, a Log Manager and a Resource Manager (i.e., DBMS). The relationships among a transactional application and these four components are shown in Figure 1. In a commercial setting, we might find a TP system such as Transarc Encina or Novell Tuxedo providing access to various resource managers, such as an Oracle or Informix relational DBMS.

In the GR Architecture, transactions are initiated by a Begin Transaction call and terminated by either a Commit or an Abort call. When initiated each transaction is assigned a unique identifier
and entered into a transaction table managed by the Transaction Manager. Each entry in the transaction table contains the transaction identifier, the transaction status, and other information. The Lock Manager maintains a lock table which contains a lock entry for every data item on which a lock has been requested (each request corresponds to an operation). Two functions, Lock and Unlock, are supported as the interface to the Lock Manager.

Since the Lock Manager already detects R/W conflicts, our work is to add support for semantic testing in order to detect semantic conflicts. To extend the GR Architecture so that it supports SBCC protocols we first extend the 2PL conflict detection mechanism. The original Lock Manager detects R/W and W/W conflicts based on syntax. Our extension to the Lock Manager will use any available semantic information to determine if a syntactic conflict can be allowed using semantic information. The way to achieve this is by classifying semantically-rich operations as either read-typed or write-typed. A read-typed operation does not change the database state and a write-typed operation may change the database state. The GR Architecture can detect conflicts between read-typed and write-typed operations. In our extension, a semantic conflict test will determine whether two operations have a semantic conflict. Two operations that have a read-typed/write-typed or a write-typed/write-typed conflict may turn out to be semantically compatible and will be allowed to run concurrently. From now on, the term “conflict” will be used to denote a read-typed/write-typed conflict or a write-typed/write-typed conflict.

In Sections 3, 4, 5, and 6, the application of ISE will be described at this high level of abstraction. In Section 7, we will present further details of the implementation in the GR Architecture, by describing a set of practical extensions to existing functions and data structures. The full version of the paper [?] contains an appendix that summarizes the GR Architecture in sufficient detail to make the presentation self-contained.

3 Commutativity

For our first application of the ISE method, we select a simple semantics-based protocol based on operation commutativity, a well known property used to determine whether operations from different transactions can be allowed to interleave. Commutativity requires a semantic compatibility function that can use available semantic information to relax the notion of conflict but requires no other enhancements to the TP system. In this section we present a general representation for semantic information and identify enhancements to the GR Architecture required to support semantic compatibility testing.
3.1 Conflict Detection and Representation

Two operations *commute* if the results (return value) of the operations are independent of their execution order and changing their execution order does not affect the results of other operations executed after these operations. This semantic notion of conflict based on operation commutativity can be defined as follows:

**Definition 1 (Commutativity and Conflict):** Two operations $o_i$ and $o_j$ commute if:

1. the results of the two operation sequences $(o_i; o_j)$ and $(o_j; o_i)$ are the same, and
2. there does not exist any sequence of operations (s) such that $(((o_i; o_j); s)$ and $((o_j; o_i); s)$ have different results.

If two operations do not commute then they semantically conflict according to this definition of commutativity.

In the GR Architecture, this definition of conflict cannot be tested directly. It would require the results of operations $o_i$ and $o_j$ to be available for the test before operation execution. Instead, commutativity of operations is specified in advance so the semantic specification can be used at run-time by a lock compatibility test function. The semantic specification for commutativity can depend on the accessed data object, operation names, parameters, and the results of the operations. In general, the more complex the specification, the more overhead is incurred for the compatibility test. Fortunately, operation commutativity and conflict can be efficiently specified via a compatibility table. With a compatibility table it is possible to determine whether operations commute through a simple table lookup, thus allowing semantic conflicts to be efficiently detected at run time. In certain cases compatibility tables can be derived directly from the semantics of the operations for a data object [?], but typically the design of a compatibility table is a creative process like programming. To simplify the presentation, in this paper we consider compatibility tables based only on operation names.

Let us examine a banking example. Assume a set of *Account* data objects and three operations *Deposit*, *Withdraw*, and *Balance* that operate on an account. *Deposit* adds a specified amount to the account balance. *Withdraw* subtracts a specified amount from the account balance, if the account balance is greater than or equal to the subtracted amount. And *Balance* returns the current value of the account. The operation compatibility table is shown in Table 1, in which columns represent operations executed and rows represent operations requested. In the compatibility table, an entry of SOK-CM indicates that the requested operation commutes with the executed operation, and an entry marked Conflict indicates that the requested operation conflicts with the executed operation.
Table 1: Operation Compatibility based on Commutativity.

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance</th>
<th>Deposit(Amount)</th>
<th>Withdraw(Amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>SOK-CM</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
<tr>
<td>Deposit(Amount)</td>
<td>Conflict</td>
<td>SOK-CM</td>
<td>Conflict</td>
</tr>
<tr>
<td>Withdraw(Amount)</td>
<td>Conflict</td>
<td>SOK-CM</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

3.2 Capabilities Required to Support Commutativity

The Lock Manager (the code that maintains the lock table) of the original the GR Architecture only detects conflicts between read and write operations, using an operation compatibility table similar to the one illustrated in Table 2. A request to the Lock Manager to execute either a read or write operation on a data object will result in a conflict test being performed between the operation requested and each operation active on the data object, returning either “OK” or “Conflict”. This does not accommodate semantically rich operations, such as the operations Deposit and Withdraw for the Account data object. However, if the operations for a data object were first classified as either read-typed or write-typed to reflect the effect the operation will have on the data object, then this conflict test could be used to detect conflicts based on the update type of the operations. We will assume this classification of operations on a data object takes place when the data object is defined. For the initial operations defined for the Account data object the classification is as follows: Balance is read-typed, and both Deposit and Withdraw are write-typed.

We can extend this syntactic conflict test further to perform semantic conflict testing by replacing entries in the conflict table marked “Conflict” with a call to the function STest, which performs semantic conflict testing. For the case of operation commutativity, the function STest simply performs a table lookup using Table 1, allowing the Lock Manager to use the semantic conflict specifications for operation commutativity to relax the notion of conflict. Operationally, this extends the original syntactic conflict test to perform a two-step semantic conflict test, as illustrated in Table 3. The first step detects conflicts between operations using only the update type (read-typed or write-typed) of the operations. If a conflict is detected, the second step utilizes available semantic specifications to determine if the operations are semantically compatible or if they conflict. For example, if a transaction was to request permission from the Lock Manager to perform a Deposit operation while a Withdraw operation was active the semantic conflict function STest would return SOK-CM, indicating that the operations were semantically compatible and could be executed concurrently.

To summarize, the additional capability that has been added to the original GR Architecture to support SBCC based on operation commutativity is:
<table>
<thead>
<tr>
<th>Operation Requested</th>
<th>Uncommitted Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>Conflict</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>Conflict</td>
</tr>
<tr>
<td>Conflict</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation Requested</th>
<th>Uncommitted Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read - typed</td>
<td>OK</td>
</tr>
<tr>
<td>write - typed</td>
<td>STest</td>
</tr>
</tbody>
</table>

Table 2: Syntactic Conflict Detection.

Table 3: Semantic Conflict Detection — Step One.

The function $\text{STest}$ represents the semantic conflict test. For operation commutativity, $\text{STest}$ simply performs a table lookup in the operation compatibility table based on commutativity, Table 1, indexing with operation names and returns the value in the corresponding entry.

1. The Lock Manager performs a two-step conflict test, i.e., the semantic conflict function $\text{STest}$ will perform a compatibility table lookup whenever a syntactic conflict is detected during a lock request. The result from a call to $\text{STest}$ will be either OK, SOK-CM, or Conflict.

4 Recoverability

Recoverability [?] is a more powerful data semantics-based protocol used to relax the notion of conflict among operations. ISE builds upon the capabilities introduced in Section 3 for commutativity, and includes additional capabilities to track transaction dependencies and manage the ordered commit of transactions. In this section we will outline the recoverability protocol and introduce capabilities that will enable the TP system to track transaction commit dependencies imposed by recoverability and manage the ordered commit of transactions.

4.1 Conflict Definition

Intuitively, an operation $o_k$ is recoverable relative to operation $o_j$ if the value returned by $o_k$, and hence the observable semantics of $o_k$, is independent of whether $o_j$ executed before $o_k$. Thus, if transaction $t_k$ precedes transaction $t_j$, and $t_k$ aborts then $t_j$ is immune from cascading aborts since the operation effects on $t_j$ remains the same. A requirement for serializability, however, is that $t_j$ cannot commit until $t_k$ commits or aborts if $t_j$'s effects on the database depends on $t_k$.  

8
Therefore, from the database system designer’s point of view, the commit dependency imposed by recoverability must be enforced in order to maintain database consistency. Thus, if \( t_j \) is recoverable but not commutative relative to \( t_k \), then the final state written by \( t_j \) depends on \( t_k \). However, from a user’s perspective, once \( t_j \) issues Commit, \( t_j \) will commit regardless of the status of \( t_k \). The function Commit can immediately return the status to the users, indicating the commitment of \( t_j \), and then defer the actual commit of \( t_j \), which makes the changes of \( t_j \) durable, until the actual commit of \( t_k \). This type of commit is referred to as a pseudo-commit. This semantic notion of conflict based on operation recoverability can be defined formally as follows:

**Definition 2 (Recoverability and Conflict):** Operation \( o_i \) is recoverable relative to operation \( o_j \) if there does not exist any sequence of operations \( (s) \) such that \( (s;o_i;o_j) \) and \( (o_j;s) \) have different results. We say that operation \( o_i \) semantically conflicts with operation \( o_j \), if \( o_i \) is not recoverable relative to \( o_j \).

### 4.2 Capabilities Required to Support Recoverability

The extended Lock Manager, introduced in Section 3 for commutativity, will be generalized to support recoverability with two modifications. First, the semantic compatibility function \( \text{STest} \) is directed to use recoverability-based operation compatibility tables, such as the one shown in Table 4 for the Account data object, to determine operation conflicts. Entries marked with SOK-RR indicate that while there is no semantic conflict between the two operations, a commit dependency must be established between the transactions that issued the recoverable operations. The function \( \text{STest} \) will return the value found in the compatibility table but does not provide the information required to establish commit dependencies (viz. identifiers of transactions (TIDs) executing operations recoverable to the operation requesting the lock). The Lock Manager maintains the sequence of active operations on any given data object in the lock table. This sequence is basically an ordered list of lock requests for operations on the data object, in which each entry includes the identifier (TID) of the transaction issuing the operation. Our second modification to the Lock Manager is that the function \( \text{STest} \) record the TIDs of all syntactically conflicting but recoverable operations in the order in which they were encountered in the lock table. This provides sufficient information to record all transaction commit dependencies imposed by recoverability. We will now consider enhancements to the Transaction Manager so that it can utilize these dependencies for the ordered commitment of transactions.

The purpose of tracking commit dependencies that arise from recoverable operations is to manage the ordered commit of transactions. When a transaction \( t_i \) issues the Commit command a commit protocol is invoked to make the changes of \( t_i \) permanent and to release the locks acquired by \( t_i \). But, for recoverability the Transaction Manager must first check whether \( t_i \) is commit dependent upon any transactions and, if so, whether all these transactions have committed or
<table>
<thead>
<tr>
<th>Account</th>
<th>Balance</th>
<th>Deposit(Amount)</th>
<th>Withdraw(Amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>OK</td>
<td>SOK-RR</td>
<td>SOK-RR</td>
</tr>
<tr>
<td>Deposit(Amount)</td>
<td>Conflict</td>
<td>SOK-RR</td>
<td>Conflict</td>
</tr>
<tr>
<td>Withdraw(Amount)</td>
<td>Conflict</td>
<td>SOK-RR</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

Table 4: Account Operation Compatibility based on Recoverability

not. If not, then \( t_i \) is blocked until all transactions that it commit-depends on are complete. When \( t_i \) is finally allowed to commit then it enters into a writing state, where all changes made by \( t_i \) are written into the database and all locks acquired by \( t_i \) are released.

In order to realize the ordered commit of transactions we must have the means to determine when a transaction is allowed to commit and introduce an additional state which indicates that a transaction is finished executing operations but waiting to commit. The means to determine when a transaction is free to commit can be supported by having each transaction keep a list of transactions that must commit before it. This list of commit predecessors \( C_{pred} \) contains the transaction identifiers recorded by the semantic compatibility function \( \text{STest} \). When a transaction issues an operation request it records all commit dependencies that arise due to recoverable operations by appending the TIDs returned by \( \text{STest} \) to the \( C_{pred} \) list. Cycles in the commit dependency graph can be detected immediately by noting that a transaction identifier can not appear more than once in the \( C_{pred} \) list. Each transaction also maintains a list of successor transactions that are waiting for it to commit, called the \( C_{succ} \) list. When a transaction commits, the commit procedure will use the \( C_{succ} \) list to “notify” all waiting transactions that it has committed by removing its TID from their \( C_{pred} \) list. When a transaction has an empty \( C_{pred} \) list it is free to commit.

During the time when a transaction is waiting for its \( C_{pred} \) list to empty it will be in the pseudo-committed state. A transaction will enter the pseudo-committed state if it issues the Commit command, but must wait for other transactions to complete due to commit dependencies. From the user’s perspective a transaction in the pseudo-committed state has completed. Once all transactions on which it commit-depends on have completed, as indicated by an empty \( C_{pred} \) list, the transaction enters a writing phase and makes all changes to the database durable and releases all locks so that these changes will be visible to other transactions. After the writing phase is complete the transaction enters the committed state.

In summary, the additional capabilities that the GR Architecture must support in order to realize the recoverability protocol are the following:

1. A Lock Manager, as described in Section 3, with the function \( \text{STest} \) generalized to allow the selection of semantics (compatibility table) that should be used to relax the notion of conflict. If transaction \( t_i \) has selected recoverability as a SBCC method that is to be
used, then \texttt{STest} will utilize an operation compatibility table based on recoverability. If syntactically conflicting but recoverable operations are found, then \texttt{STest} will record the ordered list of transactions (TIDs) that must commit prior to $t_i$.

2. The Transaction Manager stores the commit dependencies between transactions induced by concurrent recoverable operations using the lists $C_{\text{pred}}$ and $C_{\text{succ}}$, and manages the ordered commit of transactions.

3. The \texttt{Commit} procedure supports the \textit{pseudo commit} state of a transaction by delaying the actual commit of the transaction and the release of acquired locks until all requisite transactions have completed (committed).

5 Cooperative Serializability

We now apply the ISE method to a protocol that guarantees cooperative serializability (CoSR) [?, ?]. Unlike commutativity and recoverability, CoSR uses semantic information about the cooperation between transactions to relax the notion of conflict. In CoSR, transactions can form \textit{cooperative transaction groups}, where the transactions in a group collaborate over a set of data objects while maintaining the consistency of the data objects. Data consistency can be maintained only if transactions which do not belong to the transaction group are serialized with respect to all transactions in the group. In other words, groups of cooperative transactions become the unit of concurrency with respect to serializability. This will require conflict detection based on a \textit{no-conflict with} relationship between transactions in the same cooperative group, as well as the tracking of dependencies between transaction groups to ensure serializability.

5.1 Conflict Definition

We say transaction $t_i$ is a \textit{cooperative transaction} of transaction $t_j$ if they are in the same group. Given a set of cooperative transaction groups, $T$, a transaction $t_i$ conflicts with a second transaction $t_j$, only if $t_i$ and $t_j$ are in different cooperative groups and there exists an operation $o_i$ issued by transaction $t_i$ and a second operation $o_j$ issued by transaction $t_j$ such that $o_i$ and $o_j$ conflict. In CoSR, when a transaction invokes a conflicting operation on a data object a serialization dependency is established between all the transactions in their corresponding transaction groups, denoted by $T_i \ C_{\text{CoSR}} T_j$, indicating that transactions in cooperative group $T_i$ must be serialized before transactions in cooperative group $T_j$. A history over $T$ is cooperative serializable, if and only if there are no group-conflict cycles in the transitive-closure of the serialization graph. This notion of CoSR-conflict is captured in the definition presented below.
**Definition 3 (Cooperative Serializability (CoSR) and Conflict):** For two transactions \( t_i \) and \( t_j \), we say that \( t_i \) CoSR-conflicts with \( t_j \), if \( t_i[\text{op}_i(\text{object})] \) conflicts with \( t_j[\text{op}_j(\text{object})] \) and \( \text{Group}(t_i) \neq \text{Group}(t_j) \). If \( t_i \) CoSR-conflicts with \( t_j \), then a serialization dependency arises between \( t_i \) and \( t_j \), and to preserve serializability every transaction in their respective transaction groups must satisfy this serialization dependency. If the serialization dependency is violated by any transaction in the two cooperative groups, then serializability is also violated.

The definition above states that two transactions do not conflict if they are members of the same cooperative transaction group, and expresses how a dependency between two transactions which do not belong to the same cooperative group is directly established when they invoke conflicting operations on a shared data object. This is similar to the clause in the classical definition of (conflict) serializability. It also reflects the fact that when a cooperative transaction establishes a dependency with another transaction, the same dependency is established between all the transactions in their corresponding cooperative transaction groups.

### 5.2 Capabilities Required for Cooperative Serializability (CoSR)

The GR Architecture does not support transaction groups. We enhance the TP interface to indicate the initiation and completion of a cooperative transaction group, so in addition to Begin_Transaction, Commit and Abort commands we will use the functions Begin_Group, Commit_Group and Abort_Group to manage transaction groups. Also, we will define a Join(Transaction, Group) command which indicates the transaction is to become a member of the specified transaction group. We now demonstrate how the data structures and functions used to implement the Transaction Manager can be easily extended to support cooperative transaction groups.

When the command Begin_Group is issued the Transaction Manager will simply create an entry in the transaction table corresponding to the transaction group. An identifier (TID) is assigned to the transaction group entry and will be used as the cooperative transaction group identifier by all member transactions. For the Join command the Transaction Manager will record the group TID in a new Group field of the transaction entry and add the TID of the member transaction to the Members field of the group transaction entry. When an individual transaction in the transaction group issues Commit it will enter the pseudo-commit state, delaying the actual commit and release of locks. The function for the command Commit_Group will issue the Commit command for all TIDs in the Members field of the transaction group entry, actually committing each member transaction and releasing all locks; The Abort_Group command is handled similarly.

In CoSR two operations semantically conflict only if they are issued by transactions in different cooperative groups. When a conflict occurs between two operations CoSR requires that a conflict dependency be established between the transactions that issued the operations. But when a
transaction establishes a conflict dependency with another transaction, this same dependency is established between all the transactions in their corresponding transaction groups. Instead of maintaining conflict dependency information for each transaction in a cooperative group we record the conflict dependency information only in the transaction table entry corresponding to the group. To manage these conflict dependencies and ensure that cycles do not occur we generalize the transaction dependency tracking capabilities of the extended Transaction Manager, introduced in Section 4 for maintaining commit dependencies, to also track serialization (conflict) dependencies.

Before we discuss extensions to the Lock Manager, we must first consider how the compatibility table for CoSR will be constructed. A CoSR compatibility table need only contain the identifier (TID) for each cooperative transaction group along with a list of member transaction identifiers (TIDs) belonging to the group. If a conflict is detected a table lookup in the CoSR compatibility table will reveal whether the transactions that issued the conflicting operations are in the same cooperative group and, hence, semantically compatible according to CoSR. Since cooperative transaction groups are intrinsically dynamic we must be able to construct and adjust the CoSR compatibility table at runtime. Fortunately all the information necessary to construct and manage the CoSR compatibility table is available to the Transaction Manager as it processes the commands to manage transaction groups. When a transaction group is created through the command Begin Group the Transaction Manager will create a new entry (row) in the CoSR compatibility table corresponding to the transaction group, using the TID of the group as the owner of that entry. When a transaction joins a cooperative transaction group its TID is added to the CoSR compatibility table as a column entry, in the row corresponding to the transaction group it joined. Operations Commit Group and Abort Group remove all entries in the CoSR compatibility table corresponding to the transaction group. Table 5 illustrates a CoSR compatibility table, in which a no-conflict-with relationship exists between member transactions of the same cooperative group.

<table>
<thead>
<tr>
<th>Group TIDs</th>
<th>Member TIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{12}</td>
<td>t_8, t_{62}, t_{123}</td>
</tr>
<tr>
<td>T_{19}</td>
<td>t_9, t_{71}</td>
</tr>
<tr>
<td>T_{22}</td>
<td>t_{16}, t_{82}, t_{83}</td>
</tr>
</tbody>
</table>

Table 5: Transaction Compatibility based on CoSR.

The semantic compatibility function STest will use the CoSR compatibility table if the transaction requesting the lock was initiated with COSR as a selected semantic method. STest can index the transaction table using the TID to determine what Group the transaction belongs to. STest will first determine if the operation conflicts with any active operation(s) and, if so perform a lookup in the CoSR compatibility table for each conflicting operation to determine if the oper-

13
ations are semantically compatible according to CoSR. If not, then STest will record the TID of the group in which conflicting transactions belong, providing the necessary information to record transaction serialization dependencies imposed by CoSR.

In summary, the following capabilities will enable the GR Architecture to support SBCC based on Cooperative Serializability (CoSR).

1. The function STest allows CoSR semantics to be used to relax syntactic conflict. If transaction $t_i$ has been initiated with COSR semantics selected then STest will utilize the CoSR compatibility table and, if a conflict is discovered, will record the Group transactions identifier (TIDs) of the transactions that must be serialized before $t_i$.

2. The Transaction Manager supports the notion of a transaction group and records the transaction identifiers (TIDs) of each member transaction. When an individual transaction in a transaction group issues Commit it enters the pseudo-commit state and no locks are released. Finally, when the last transaction in the group issues CommitGroup the Transaction Manager can traverse through the list of member TIDs to commit each transaction and release all locks that were acquired by transactions in the group.

3. The Transaction Manager records conflict dependencies between transaction groups induced by conflicting operations on shared data objects, and prevents cycles from occurring. If a cycle is detected the transaction that operation requested is blocked.

6 Epsilon Serializability

This section describes the addition of Epsilon Serializability (ESR) support using the ISE method. Due to the lack of space for a conference submission, we omit most of explanations and show only the definition and crucial tables for the combination of ESR with other SBCC protocols.

**Definition 4 (Epsilon Serializability (ESR) and Conflict):** For two transactions $t_i$ and $t_j$, we say that $t_i$ epsilon-conflicts with $t_j$ if $t_i[\text{operation}(\text{object})]$ conflicts with $t_j[\text{operation}(\text{object})]$ and $\neg \text{Safe}(t_i)$. The safety pre-condition for transaction $t_i$ with respect to performing operation $op$ on data object $Account$ is defined as follows:

$$\text{Safe}(t_i) = \begin{cases} 
\text{import}^{t_i} + \text{import\_inconsistency}^{t_i}_{(\text{op}, \text{Account})} \leq \text{espec}^{t_i}_{\text{import}} \\
\text{export}^{t_i} + \text{export\_inconsistency}^{t_i}_{(\text{op}, \text{Account})} \leq \text{espec}^{t_i}_{\text{export}} 
\end{cases}$$
Where \( \text{import}_t^j \) and \( \text{export}_t^j \) stand for the amount of inconsistency that has already been imported and exported by transaction \( t_j \). And, \( \text{import \_ inconsistency}^{(O_p, \text{Account})}_t \) is defined as the maximum amount of inconsistency that transaction \( t_j \) can import with respect to performing operation \( op \) on data object \( \text{Account} \), while \( \text{export \_ inconsistency}^{(O_p, \text{Account})}_t \) is the maximum amount of inconsistency exported by transaction \( t_j \) performing operation \( op \) on data object \( \text{Account} \).

In addition to specifying compatibility between two operations, the compatibility table for ESR must express information about the potential inconsistency that could be introduced by interleaving operation executions. Without this information the semantic compatibility function \( \text{STest} \) would not be possible to guarantee ESR-safeness. For example, assume that a \( \text{Balance} \) operation is currently active and request for a \( \text{Deposit} \) operation is received. These operations conflict because \( \text{Balance} \)'s view of the \( \text{Account} \) data object could be corrupted. \( \text{STest} \) must be able to perform a table lookup in the ESR compatibility table and determine if the operation interleaving is allowed under ESR, and, if so, use the value of the potential inconsistency to determine if the ESR-safeness condition is satisfied. As Table 6 shows, the interleaving of \( \text{Balance} \) and \( \text{Deposit} \) operations is allowed with a potential inconsistency equal to the amount being deposited. In general, a conflict would have the potential increase in inconsistency equal to the difference between the value of \( \text{Account} \) before the operation takes place and the value after the operation takes place.

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance</th>
<th>Deposit(Amount)</th>
<th>Withdraw(Amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Balance} )</td>
<td>OK</td>
<td>Amount</td>
<td>Amount</td>
</tr>
<tr>
<td>( \text{Deposit(Amount)} )</td>
<td>Amount</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
<tr>
<td>( \text{Withdraw(Amount)} )</td>
<td>Amount</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

Table 6: Operation Compatibility and Inconsistency based on ESR

In summary, the following capabilities will enable the GR Architecture to support ESR:

1. The Transaction Manager will record import and export c-specs for a transaction. Entries in the transaction table will be augmented with fields to store the import and export specifications and corresponding inconsistency counters.
2. The semantic compatibility function \( \text{STest} \) will utilize ESR semantics to relax syntactic conflicts, and if a conflict is detected it will perform ESR-safeness testing.

7 A 2PL-Based Implementation of ISE

We now discuss how capabilities required for SBCC can be realized through a set of practical extensions to a transaction processing system. Table 7 below, summarizes the additional capa-
abilities described in Sections 3–6, and identifies the SBCC protocols each capability supports. Entries marked with a “*” indicate that the capability is required to implement the SBCC technique. Table 8 then lists the data structures and algorithms of the GR Architecture that must be extended to support each capability. The mapping from capabilities to the data structures and algorithms, presented in Tables 7 and 8, identifies that all changes to the GR Architecture are centered in the Transaction Manager and the Lock Manager.

<table>
<thead>
<tr>
<th>Additional Capabilities Required of GR Architecture</th>
<th>Semantics-Based Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Commutativity</td>
</tr>
<tr>
<td>Semantic conflict testing</td>
<td>*</td>
</tr>
<tr>
<td>Transaction dependency tracking</td>
<td></td>
</tr>
<tr>
<td>Pseudo commit of transactions</td>
<td></td>
</tr>
<tr>
<td>Transaction groups</td>
<td></td>
</tr>
<tr>
<td>ESR-safeness testing</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Summary of Capabilities Required for SBCC.

Using Tables 7 and 8 as specifications, we describe extensions first to the Transaction Manager and then to the Lock Manager, to realize the capabilities required for SBCC. We then outline the implementation of these extensions in the commercial TP system Encina. In describing the required extensions and their implementation in Encina, we wish to demonstrate the practicality of the ISE method and illustrate how relatively simple extensions can be used to implement SBCC in a commercial TP system. For reasons of space we can not present an detailed overview of Encina nor elaborate on the data structures and algorithms of the GR Architecture here. The full version of this paper [?] contains both an appendix describing Encina and an appendix which summarizes the most important aspects of transaction management and lock acquisition in the GR Architecture.

7.1 Transaction Manager Extensions

The Transaction Manager assigns a unique transaction identifier (TID) to a transaction when it is initiated, creating an entry in the transaction table to record the TID and other pertinent information, and tracks the transaction through its execution. The Transaction Manager also provides the transaction application programming interface, consisting of commands such as BeginTransaction, Commit, and Abort, and orchestrates the transaction actions associated with these commands. We first describe the additional information we will require the Transac-
Table 8: Data Structures and Algorithms in the GR Architecture extended to support capabilities.

<table>
<thead>
<tr>
<th>Capabilities</th>
<th>Data Structure</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic conflict testing</td>
<td>• Compatibility Table(s)</td>
<td>• Lock Function</td>
</tr>
<tr>
<td></td>
<td>• Lock Table Entry</td>
<td>• STest Function</td>
</tr>
<tr>
<td>Transaction dependency tracking</td>
<td>• Transaction Table entry</td>
<td>• Depends-on Function</td>
</tr>
<tr>
<td>Pseudo commit of transactions</td>
<td>• Transaction Table entry</td>
<td>• Commit Function</td>
</tr>
<tr>
<td>Transaction groups</td>
<td>• Transaction Table entry</td>
<td>• BeginGroup Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CommitGroup Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AbortGroup Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Join Function</td>
</tr>
<tr>
<td>ESR-safeness testing</td>
<td>• Transaction Table entry</td>
<td>• ESR_safe Function</td>
</tr>
</tbody>
</table>

The data structure of interest that is managed by the Transaction Manager is called the transaction table. Among other things, each entry in the transaction table contains a list of the operations performed by the transaction, its transaction identifier (TID), status, and pointers to the lock entries related to this particular transaction. For a transaction $t_j$ we add the following fields to its entry in the transaction table:

- **SBCC** List of SBCC methods that are to be applied in semantic compatibility testing.
- **C_pred** List of identifiers (TIDs) of the transactions that must commit before $t_j$.
- **C_succ** List of TIDs of the transactions that are waiting for $t_j$ to commit.
- **Group** The TID of the group to which $t_j$ belongs.
- **Members** List of TIDs of the transactions that are members of the group represented by $t_j$.
- **S_pred** List of TIDs of the transactions that must be serialized before $t_j$.
- **S_succ** List of TIDs of the transactions that must be serialized after $t_j$.
- **Import-limit** The $\epsilon$-specification limit on the inconsistency that $t_j$ can import (read).
- **Export-limit** The $\epsilon$-specification limit on the inconsistency that $t_j$ can export (write).
- **Import** The amount of inconsistency that $t_j$ has already imported.
- **Export** The amount of inconsistency that $t_j$ has already exported.
These extensions to the data structures of the Transaction Manager are summarized in Figure 2.

The interface of the `Begin Transaction` command is extended to accept optional parameters for the selection of SBCC methods that are to be used during transaction processing: `Begin Transaction(&[CM,RC,CoSR,<ESR, Import-limit, Export-limit>])`, where the prefix “&” indicates an optional parameter. If the parameter ESR is selected then the values for the transactions import and export ε-specification are stored in the appropriate fields of the transactions entry in the transaction table. The list of selected SBCC methods that are to be used will be utilized by the Lock Manager in requesting locks on behalf of the transaction.

![Figure 2: Extended data structures of the Transaction Manager and Lock Manager.](image)

### 7.1.1 Transaction Groups

Extensions for transaction groups are added to support Cooperative Serializability (CoSR), where individual transactions can form cooperative groups. A transaction group is created using the new command `Begin Group(GroupName)`, and individual transactions join a cooperative group using the new command `Join(GroupName)`. The `Begin Group` procedure creates a standard transaction entry in the transaction table for the group and associates the `GroupName` with the TID assigned to the new entry. When transaction $t_j$ issues `Join(GroupName)` the Join procedure creates an entry in the transaction table for $t_j$ and records the TID associated with `GroupName` in the Group field, and then records the TID of $t_j$ in the Members field of the transaction entry corresponding to the group. The procedures for the commands `Commit Group` and `Abort Group` can traverse through the Members list of the transaction group entry to issue `Commit` and `Abort` commands to the individual member transactions.
In addition to the list of lock requests maintained for each transaction, the Transaction Manager will also maintain a list of requests for each group. When a member transaction in a transaction group issues `Commit` no locks are released. However, if it aborts the locks acquired by the transaction are released. Finally, when the last transaction in the group issues `Commit_Group` the Transaction Manager can traverse through the lock list associated with the group and release all locks that were acquired by transactions in the group.

### 7.1.2 Transaction Dependency Tracking

We extend the Transaction Manager to record dependencies between transactions through the introduction of a new `Depends-on` function. Transaction dependencies are induced by both concurrent recoverable operations and conflicts detected between transactions in different cooperative transaction groups. Each transaction entry has separate fields to record both commit and serialization dependencies, and the `Depends-on` function provides an argument to select the dependency type that is being recorded: `Depends-on([Commit,Serial],Ti,Tj)`. Conceptually, the dependency information stored in the transaction entry forms a directed graph, in which each transaction entry is a node in the graph and the fields `pred` and `succ` are the incoming and outgoing edges, respectively. Before the dependency (edge) is recorded between transactions $t_i$ and $t_j$, the function `Depends-on` performs a check to prevent dependency cycles from occurring. If a cycle is detected then the transaction issuing the operation request is blocked.

### 7.1.3 Managing the Pseudo Commit of Transactions

The status of a transaction $t_j$ in the transaction table, is one of the following states: `running`, `blocked`, `writing`, `pseudo commit`, `committed`, or `aborted`. A transaction enters into the running state when it starts and enters into the blocked state when it is waiting for a lock. We modify the `Commit` procedure to manage the `pseudo commit` for the transaction. When transaction $t_j$ issues `Commit` the commit protocol is invoked. It will first check whether the field $C_{pred}$ (the set of transactions that $t$ commit-depends on) is empty or not. If yes, it enters the transaction into the writing state, where all updates to the database are made durable and made visible to other transactions by releasing all locks held by $t_j$ at this point. It also removes all commit dependencies recorded through $t_j$’s $C_{succ}$ list, notifying all transactions that are commit-dependent on $t_j$ that it has completed, and releases all locks acquired by $t_j$. If $C_{pred}$ is not empty, it is blocked until all transactions that $t_j$ commit-depends on complete (i.e., wait until $C_{pred}$ becomes empty).
7.2 Lock Manager Extensions

The Lock Manager provides transactions with the means to acquire and release locks on data objects. The Lock Manager maintains a lock table which contains a lock entry for every data object on which a lock has been requested (each request corresponds to an operation). Each lock request is in one of two states, Granted or Waiting. In order to determine if a lock request is semantically compatible with Granted lock requests we extend the information stored with each lock request entry in the lock table to include the following additional information:

**Operation Name** The name of the operation the transaction is attempting to execute.

**TID** Transaction identifier \(T_{id}\) of the transaction that issued the lock request.

**Max** Maximum change to the value of the data object that could result from the active operation. This provides an upper bound on the inconsistency that could be imported or exported to a transaction executing under ESR.

These extensions to the data structures of the Lock Manager are summarized in Figure 2.

Accordingly, the interface to the `Lock` procedure is extended to optionally pass the additional information that will be stored in each lock table entry: \(\text{Lock}(\text{Object}, \text{operation}, \text{OpName}, \text{TID}, \text{Max})\). The Max parameter will be used for ESR-safeness testing, while the OpName parameter is used by `STest` to index selected compatibility tables using the operation name, and the TID parameter provides access to transaction information, such as dependencies, group identification, and inconsistency specification and counters for ESR-safeness testing.

### 7.2.1 Semantic Conflict Testing

Semantic conflict testing is a generalization of standard conflict testing mechanisms, in which semantic information in the form of a compatibility table is used to relax the notion of conflict. The function `Lock` implements semantic conflict testing as a two-step test. Step one is the standard syntactic conflict test based on the update of the operation (e.g., read-typed or write-typed) and, if a conflict is detected, step two performs semantic compatibility testing using the appropriate semantic compatibility table(s) to determine if the two operations semantically conflict or are compatible. If the two operations semantically conflict, then ESR-safeness can be tested, if ESR has been selected, to determine if the inconsistency introduced from the conflict would exceed the trans-c-spec. Figure 3 shows the procedure that functions `Lock` and `STest` will follow when a lock request is received.

The logic of the extended `Lock` function is described in detail in Figure 4, while Figure 5 presents the logic of the semantic compatibility function `STest`. Together these functions imple-
ment semantic conflict detection, using selected semantic methods and ESR to relax syntactic conflicts.

7.3 An Encina Implementation of SBCC

We now discuss how the extensions required for SBCC can be implemented in Encina, a commercial TP system distributed and supported by Transarc. The components of Encina that need to be modified to support SBCC are in the Encina Toolkit. Specifically, the Tran code of the Encina Toolkit is modified to support the necessary changes to the Transaction Manager and lock requests, while the Lock Service code of the Toolkit is modified to support semantic conflict testing. Most of these modifications follow immediately from the description of the extensions previously presented.

We first modify the function tranBegin, which denotes the start of a transaction. It takes as arguments the transaction identifier of the parent transaction (NULL if none) and a pointer (TID) to the transaction entry in the transaction table. To support SBCC, we augment the interface to accept optional arguments, one for the SBCC methods that are to be used for this transaction, the import inconsistency specification, and the export inconsistency specification. To store these values, and other values that may be supplied later, such as commit and serialization dependencies, group and member transaction identifiers, and corresponding inconsistency counters, we use property lists. Property lists are a mechanism provided by the Encina Toolkit to associate

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{semantic_compatibility_function.png}
\caption{Semantic Compatibility Function for Acquiring a Lock}
\end{figure}
status Lock(object, operation, TID, OpName, Max) {
  lock = getLock(object);
  request = new request(operation, TID, OpName, Max);
  /* Step 1 - syntactic conflict test */
  if(lock compatible(operation, lock→grantedMode)) {
    lock→grantedMode = computeLockMode(operation, lock);
    request→status = Granted;
    return OK;
  }
  /* Step 2 - perform semantic conflict testing if SBCC methods selected */
  else if (transactionTable[TID]SBCC ≠ nil)
    STest(request, object, operation, TID, OpName, Max);
  if (request→status == Granted)
    return OK;
  else {
    /* Wait for lock until timeout or deadlock detected */
    request→status = Waiting;
    Wait(timeout);
    if (request→status == Granted) return OK
    else {
      delete request(request);
      return request→status; /* Timeout or Deadlock */
    }
  }
}

Figure 4: Extended Lock Function

additional information with each transaction. This additional transaction information can be accessed using the TID and property name. For the transaction interface we introduce the functions Begin Group, Commit Group, Abort Group, and Join, to support transaction groups. The definition of these functions follows directly from their specifications presented in Section 7.1.1.

In Encina, lock requests are done through the lock Acquire function, which accepts arguments for the identifier of the requesting transaction (TID), the desired lock mode, and the logical name of the lock. To support SBCC, we augment the lock entry data structure to record the name of the operation requesting the lock (OpName), and the maximum change to be performed on the data object by the operation (Max).

A requested lock is granted only if it is compatible with the lock modes of the transactions currently holding a lock on the same data object. Conflict testing is performed by the function lock Conflict With Holders which takes three arguments, the requested lock mode, the lock entry of the desired item, and the TID of the requesting transaction. Syntactic conflicts are detected by a table lookup keyed on the current lock status and the requested lock mode. The table lookup returns either true or false for compatibility.

After Encina detects a syntactic conflict by comparing the overall lock status and the request lock mode, it performs further analysis to determine whether the detected conflict is real or whether it should be allowed. The conflict may be not be a real one, for instance, if the requesting
status STest(request, object, operation, TID, OpName, Max) {
    Commit list = nil;
    SOK = False;
    ALL_SOK = True;
    /* Test compatibility with all granted lock requests */
    for each granted request in lock→REQ granted {
        /* If CM selected test for commutativity */
        if (CM in transaction table[TID]→SBCC)
            /* If not(SOK) then RC selected, test for recoverability */
            if (not(SOK) AND (RC in transaction table[TID]→SBCC))
                if (objectCC[0] = = SOK-CC)
                    SOK = True;
        /* If not(SOK) and COSR selected, test for cooperative serializability */
        if (not(SOK) AND (COSR in transaction table[TID]→SBCC))
            if granted request→TID in COSR[request→Group]
                SOK = True;
        /* record serialization dependencies in transaction table for CoSR */
        if (not(SOK) AND (COSR in transaction table[TID]→SBCC))
            ALL_SOK = (ALL_SOK AND SOK);
    }
    if ALL_SOK {
        request →status = Granted;
        /* record commit dependencies in transaction table */
        for each dependency in Commit list
            Dependson[Commit, dependency];
        return OK;
    } else
        return CONFLICT;
}

Figure 5: Semantic Compatibility Function STest.

transaction already has the requested lock on the data item or is part of a nested transaction that has the lock. A conflict may be allowed if every transaction currently holding the lock agrees that the conflict is allowable; this is exactly the functionality required for semantic compatibility testing. Encina supports this latter type of testing with conflict callback functions. Conflict callback functions are associated with individual transactions and are used to determine whether the detected syntactic conflict between an associated transaction and the conflict transaction should be treated as a conflict or should be allowed. If there is no conflict callback function associated with the transaction or if the callback function returns false, the conflict is not allowed. Otherwise the conflict is relaxed and the lock request is granted.

We use the conflict callback functions to implement semantic conflict testing for SBCC. Traditional transactions will not have conflict callback functions associated with them and will therefore
reject any attempt to allow syntactic conflicts. If the transaction designer or user has selected SBCC methods or ESR to be used with the transaction, then the function \texttt{STest}, described in Figure 5, is registered as the conflict callback function, so the selected methods will be used to relax the notion of conflict. A transaction that does not pass all the conflict callback function testing in \texttt{STest} is said to cause a semantic conflict and the transaction must block. Otherwise, there is no semantic conflict so \texttt{STest} will record any inconsistencies or dependencies, and the lock request is granted.

8 Conclusion

In this paper, we described the Integrated Semantics Extension (ISE) Method for implementing and combining semantics-based concurrency control (SBCC) protocols in practical TP systems. ISE is not yet another new SBCC protocol. Rather, it extends practical TP systems in a modular way to implement a variety of SBCC protocols. Unlike previous proposals of SBCC protocols which are largely incompatible with each other, ISE shows that we can systematically implement many SBCC protocols in the same TP system and allow the users a choice of any combination of them at run-time at the fine granularity of each transaction.

The implementation was described in detail for several specific SBCC protocols, representing important classes. We chose commutativity [2] and recoverability [2] to show that ISE handles well the protocols based on data operation semantics. For SBCC protocols based on transaction semantics, we chose cooperative serializability [2]. To relax serializability in bounded amounts, we showed that ISE also supports epsilon serializability [2]. With such a wide coverage of examples, we make a case for the general applicability of ISE.

We achieved this wide coverage without resorting to a highly abstract model for algorithm description. Gray and Reuter [2] have described an effective and concrete TP architecture that has simple and direct mapping into production TP monitors such as Encina. ISE uses Gray and Reuter’s architecture to specify the implementation of concrete SBCC protocols. In particular, Section 7 specifies the ISE modular extensions in terms of Gray and Reuter architecture, and then follows the specifications to implement the extension in Encina, through a set of practical modifications to modules the Encina Toolkit.

Once applied to a practical TP system, ISE gives application designers the ability to mix and match the semantics of importance to them. Any transaction, at any time, can run under the conventional two-phase locking rules, without having to perform any special operation. The SBCC protocols can be selectively employed for data objects, transactions or classes of transactions for which they are likely to provide the most benefit.
Given a research topic as wide as semantics-based concurrency control, it is perhaps not surprising that we have not solved all the problems in the area. For example, different SBCC protocols may interact with each other in unexpected ways, both in terms of conflict semantics and implementation. We have resolved these interactions for the four SBCC protocols described in this paper. However, a more rigorous classification of SBCC protocols and strengthening of ISE to follow the classification is a subject of current research. Furthermore, although we have chosen four representative protocols, there are some SBCC protocols that defy classification. We have had some success with these "exotic" methods, e.g., chopping transactions and Escrow method [?], but their integration with a systematic approach such as ISE remains a topic of future research.

References


