

# Automating semantics-based reconciliation for mobile databases

Nuno Preguiça\*, Marc Shapiro<sup>+</sup>, J. Legatheaux Martins\*

\*Dep. Informática, Universidade Nova de Lisboa, Portugal

<sup>+</sup>Microsoft Research Ltd., Cambridge, UK

{nmp, jalm}@di.fct.unl.pt

Marc.Shapiro@acm.org

## Abstract

Optimistic replication lets multiple users update local replicas of shared data independently. These replicas may diverge and must be reconciled. In this paper, we present a general-purpose reconciliation system for mobile transactions. The basic reconciliation engine treats reconciliation as an optimization problem. To direct the search, it relies on semantic information and user intents expressed as relations among mobile transactions. Unlike previous semantics-based reconciliation systems, our system includes a module that automatically infers semantic relations from the code of mobile transactions. Thus, it is possible to use semantics-based reconciliation without incurring the overhead of specifying the semantics of the data types and/or operations.

## 1. Introduction

Distributed systems replicate shared data to improve read availability and performance. Optimistic replication allows a user to *update* a local replica independently. This improves write availability in the presence of high network latencies, failures, voluntary disconnection, or parallel development, but allows replicas to diverge. This is especially useful for mobile computing environments.

Repairing divergence after the fact, called *reconciliation*, combines the isolated updates [15]. In operation-based (or log-based) approaches, update actions are recorded in a log; reconciliation *replays* the combined actions, from the initial state, according to some *schedule*.

Sometimes, it is impossible to execute all updates because the execution of some updates would violate a *precondition* or an *invariant*. In this case, relying on conflict resolution rules (e.g. exploring alternative updates) and running updates in a different order may allow better reconciliation results (i.e., more updates can be executed). The updates that cannot be executed must be *dropped*. Reconciliation must minimize the dropped updates, as dropping an action may have a high impact.

### 1.1. Limitations of existing systems

Several systems use optimistic replication and implement some form of reconciliation — see [15] for a recent survey. Many older systems reconcile by comparing final tentative states (e.g. Coda [8]). Other systems use history-based reconciliation (e.g. [17, 12, 10]). This is also our approach.

Many existing reconciliation systems are dedicated to a single application semantics — e.g. CVS [2]. Others (e.g. Bayou [17] and Deno [7]) are general-purpose and use a simple syntactic

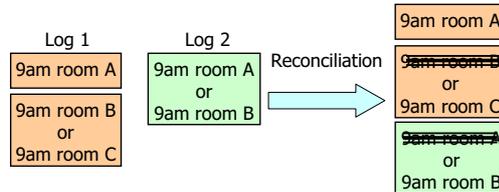


Figure 1: Syntactic scheduling spuriously fails on this example

criterion (e.g. timestamps) to decide which updates to retain and/or the order to run them. This is inflexible and may cause spurious conflicts even in very small problems. Consider the example of figure 1, where two users make meeting requests to a calendar program. One user requests room A at 9:00, and either room B or C, also at 9:00. Meanwhile, other user requests either room A or B at 9:00. Combining the logs in some simple way does not work. For instance running Log 1 then Log 2 will reserve rooms A and B for the first user, and the second user's requests is dropped. Running Log 2 first has a similar problem. Satisfying all three requests requires reordering them, which syntactic systems cannot do.

Semantics-based reconciliation systems use semantic information to merge divergent streams of activity. Several approaches have been proposed. In [18], commutativity information is used to improve concurrency. In Sync [10], the programmers may define reconciliation rules for each pair of actions that can be executed upon each object. Operational transformation algorithms [16] re-write update parameters to enable order-independent execution of non-conflicting actions, even when they do not commute.

Experiments show that semantics-based reconciliation tends to produce better reconciliation results [14]. However, the overhead of expressing the semantic information needed by the reconciliation engine is usually non-trivial. Therefore, some programmers will not be willing to execute this extra work or end up creating bad semantic rules that leads to poor reconciliation results.

## 1.2. Our approach

This paper presents the SQLIceCube general-purpose reconciliation system for mobile database systems.

In SQLIceCube, update actions are expressed as small programs written in a subset of PL/SQL [11]. These programs, dubbed mobile transactions (or simply transactions), are submitted by users' applications to modify the database. Figure 2 presents a simple mobile transaction to submit a new order in a mobile sales application. The code of the transaction verifies if the stock is sufficient and the price is acceptable before adding the new order to the database.

Mobile transactions may be tentatively executed against local replicas or not. These mobile transactions are then submitted to the SQLIceCube reconciliation system that combines concurrently executed transactions. During reconciliation, the mobile transactions' programs are executed again (i.e., mobile transactions are not integrated in the database using the read sets/write sets of the tentative execution). Reconciliation is executed in two steps: the semantic inference and the basic reconciliation step.

The semantic inference module automatically infers the semantic information needed by the

**RENPAR'15 / CFSE'3 / SympAAA'2003**  
**La Colle sur Loup, France, 15 au 17 octobre 2003**

```
----- ORDER PRODUCT: id = 80; quantity = 10; highest price = 100.00 -----  
DECLARE  
  l_stock INTEGER;  
  l_price FLOAT;  
BEGIN  
  SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;  
  IF l_price <= 100.0 AND l_stock >= 10 THEN -- check price acceptability and stock availability  
    UPDATE products SET stock = stock - 10 WHERE id = 80;  
    INSERT INTO orders VALUES (newid,8785,80,10,l_price,'to process'); -- newid returns the same unique  
                                                                    -- identifier in the client and in the server  
    COMMIT; -- commit transaction (and return)  
  ELSE  
    ROLLBACK; -- abort transaction (and return)  
  ENDIF;  
END;
```

Figure 2: Mobile transaction adding a new order in a mobile sales application.

basic reconciler. This information is obtained in two steps. First, each transaction is statically analyzed to extract relevant information (read/written data items and preconditions). Second, for each pair of transactions, a set of semantic static relations (e.g. overlap) is inferred using the extracted information.

The basic reconciler treats reconciliation as an optimization problem, trying to find the schedule that allows more transactions to succeed. The static semantic information directs the search using an heuristic approach. This reconciler is an extension of the IceCube reconciler detailed in [14].

The main contribution of this work is the semantic inference approach to simplify the use of semantics-based reconciliation. Unlike previous systems, semantic information used during reconciliation is automatically extracted from operations, thus alleviating programmers from most of the work usually involved in semantics-based reconciliation. We also present a semantics-based reconciler that uses the automatically-extracted semantic information.

The remainder of this paper is organized as follows. Section 2 outlines the general-purpose semantics-based reconciler. Section 3 describes the semantic inference module. Section 4 discusses some extensions an on-going work and section 5 concludes the paper with some final remarks.

## **2. Generic semantics-based reconciler**

In this section we outline the generic SQLIceCube reconciler. We start by describing the semantic information used.

### **2.1. Dynamic constraints/pre-conditions**

Dynamic constraints are restrictions to the execution of a mobile transaction that depend on the database state. Therefore, they can only be verified in runtime. These restrictions are specified in the code of transactions as preconditions that lead to commit or rollback statements. For example, the transaction of figure 2 can only commit if the stock and price for the specified product meet the given conditions. Whenever possible, we use this information to infer suitable static relations, as explained later.

## 2.2. Static constraints/relations

Static constraints are relations among mobile transactions that do not depend on the database state. Therefore, they can be used during the reconciliation process without accessing to the database. Two types of static relations are used in our reconciler: log relations and data relations.

### 2.2.1. Static log constraints/relations

Log constraints are relations that express users' intents. These relations are independent of the semantics of each specific basic transaction. Instead, they encode the semantics of a macro-operation composed by several basic transactions. Usually they are explicitly set by applications but some of them can also be automatically inferred as discussed later. The following log relations are defined:

**Alternatives** Specify that a single transaction must be committed from a set of alternatives. This allows the definition of basic conflict resolution rules.

**Weak predecessor-successor** If both transactions commit, the predecessor must be executed before the successor.

**Strong predecessor-successor** The successor transaction can only be executed after (and if) the predecessor transaction commits.

**Parcel** Defines an all-or-nothing group of transactions.

### 2.2.2. Static data constraints/relations

Data constraints are relations between transactions that encode the semantics of the operations, independently of the database state. The following static data relations are used:

**Overlap** Two transactions overlap if they act upon the same data items.

**Commute** Two transactions  $t_1$  commute if the result of executing both is independent of the execution order.

**Helps** The transaction  $t_1$  *helps* the transaction  $t_2$  if the commitment of  $t_1$  changes the database state in a favorable way for the success of  $t_2$  (or more precisely, if  $\forall s \in S, valid(t_2, s) \Rightarrow valid(t_2, t_1(s))$  and  $\exists s \in S, \neg valid(t_2, s) \wedge commit(t_2, t_1(s))$ ), with  $S$  the set of all possible database states and  $valid(t, s)$  true if  $t$  can commit in state  $s$ .

**Prejudices** The transaction  $t_1$  *prejudices* the transaction  $t_2$  if the commitment of  $t_1$  changes the database state in a prejudicial way for the success of  $t_2$ .

**Makes impossible** The transaction  $t_1$  *makes impossible* the transaction  $t_2$  if the commitment of  $t_1$  changes the database state in a way that makes impossible to commit  $t_2$  (note that executing a transaction  $t_3$  after  $t_1$  may make  $t_2$  possible again).

The static relations *helps*, *prejudices* and *makes impossible* encode the influence of one transaction over the dynamic constraints/preconditions of the other transaction.

### **2.3. Heuristic reconciliation engine**

The goal of the reconciler is to create a schedule that maximizes the number (or value) of transactions that can be executed with success. As, for non-trivial problems, it is impossible to check all possible schedules, we have implemented a reconciliation algorithm that probes the space of possible solutions heuristically. To this end, the reconciler creates and executes a sequence of schedules that combines the transactions submitted concurrently by multiple mobile clients. The reconciliation process ends when a schedule that meets a specified optimality condition is generated (or the specified search time is exhausted).

The reconciler creates each schedule incrementally. At each step, the reconciler selects for execution one transaction based on its merit (with randomization). An oracle estimates the merit of each transaction, heuristically computing the benefit of adding a given action to a given partial schedule. To this end, the oracle uses the information about the semantic static relations established among transactions.

Due to space limitations it is impossible to further detail our reconciliation algorithms and optimizations (e.g. clustering). However, when compared to the IceCube algorithms detailed in [14], the new algorithms only presents small modifications to integrate the new static relations and the cleaner execution of transactions (in contrast to actions).

### **3. Automatic extraction of (data) relations**

As explained before, the semantic static relations between transactions are the inputs needed by the basic reconciler. In this section we describe how data relations are automatically inferred (log relations are naturally added by applications when they submit complex operations composed by several transactions).

The common approach in semantics-based reconciliation is to require the definition of methods (e.g. IceCube [14]) or tables (e.g. Sync [10]) that specify semantics-based information/rules between two operations. This approach has two drawbacks. First, even when it is simple to specify each rule, it tends to be a repetitive, verbose and error-prone work. Second, it makes difficult to introduce new operations, as it is necessary to extend the rules for data relations. In a database system where generic mobile transactions may be submitted, this becomes an important issue.

In SQLIceCube we have designed a mechanism to automatically infer the data relations between mobile transactions. To this end, the code of mobile transactions is statically analyzed extracting a set of relevant information. This information is used to infer the relations between transactions. In this section we detail this mechanism.

Static analysis [5] of programs has been used before in several systems to verify the correctness [4, 1] and equivalence of programs [9]. However, to our knowledge, this work is the first to use a similar approach in reconciliation. Our static analysis presents the following uncommon characteristics: it is executed at reconciliation runtime and the information extracted from each program is used to infer relations between two programs (other than equivalence).

#### **3.1. Extract information**

To extract information from a mobile transaction, the transaction program is statically analyzed checking all execution paths. For each path that ends in a commit statement, the following group of information is extracted:

**RENPAR'15 / CFSE'3 / SympAAA'2003**  
**La Colle sur Loup, France, 15 au 17 octobre 2003**

```
BEGIN
  SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
  -- l_stock = read(products,id=80,stock)
  -- l_price = read(products,id=80,price)
  IF l_price <= 100 AND l_stock >= 10 THEN
    -- as the contrary leads to a rollback
    -- precondition(read(products,id=80,stock)>=10 AND read(products,id=80,price)<=100)
    UPDATE products SET stock = stock - 10 WHERE id = 80;
    -- update(products,id=80,stock,stock-10)
    INSERT INTO orders
      VALUES (newid,8785,80,10,l_price,'to process');
    -- insert (orders, (id,client,product,qty,price,status), (647,8785,80,10,l_price,'to process'))
  COMMIT;
ELSE
  ROLLBACK;
ENDIF;
END;
```

Figure 3: Information extracted from a mobile transaction that adds an order in a mobile sales application.

**Semantic read set** The *semantic read set* contains the semantic description of each relevant *read data item* (if the read value is not used, it is not relevant). This information is obtained from select statements.

**Semantic write set** The *semantic write set* contains the semantic description of each *written data item*. This information is obtained from insert, update and delete statements.

**Precondition set** The *precondition set* contains all conditions in the given execution path. This information is obtained from if instructions.

The semantic description of a data item includes the name of the table, the name of the column and the condition used to refer the data item. For example, figure 3 shows, as comments (lines starting with --), the information that can be automatically extracted from the simple mobile transaction that adds a new order in a mobile sales applications.

### 3.2. Infer relations

The relations between each pair of transactions are inferred *comparing* the information extracted from each transaction. As this information only contains the semantic description of each data item, it is necessary to verify if the conditions expressed in the different semantic descriptions refer the same data items or not. To this end, in our prototype, we are using a general-purpose constraint solver to verify the compatibility of semantic descriptions. In database systems, similar analysis are also executed in the context of query optimization [6] and semantic caching [3]. Given two transactions  $t_1$  and  $t_2$ , each one with a single group of information, the following rules are used to infer each data relation.

**Overlap (default:true)**  $t_1$  overlaps with  $t_2$  if  $t_1$  reads a data item written by  $t_2$  (or vice-versa), or  $t_1$  writes a data item written by  $t_2$  (or vice-versa).

**Commute (default:false)**  $t_1$  commutes with  $t_2$  if neither  $t_1$  nor  $t_2$  reads a data item written by  $t_2$  or  $t_1$ , and all  $t_1$  writes commute with all  $t_2$  writes (and vice-versa) (i.e. only blind and commutative writes overlap).

**RENPAR'15 / CFSE'3 / SympAAA'2003**  
**La Colle sur Loup, France, 15 au 17 octobre 2003**

```
BEGIN
SELECT status INTO l_status FROM orders WHERE id = 3;
-- l_status = read(orders,id=3,status)
IF l_status = 'to process' THEN
-- precondition(read(orders,id=3,status)='to process')
UPDATE orders SET status = 'cancelled' WHERE id = 3;
-- update(orders,id=3,status,'cancelled')
UPDATE products SET stock = stock + 30 WHERE id = 80;
-- update((products,id=80),stock,stock+30)
COMMIT;
ELSE
ROLLBACK;
ENDIF;
END;
```

Figure 4: Information extracted from a mobile transaction that cancels an order in a mobile sales application.

**Helps (default:true)**  $t_1$  helps  $t_2$  if  $t_1$  writes changes the database in a favorable way for  $t_2$  preconditions to be true.

**Prejudices (default:true)**  $t_1$  prejudices  $t_2$  if  $t_1$  writes changes the database in a prejudicial way for  $t_2$  preconditions to be true.

**Makes impossible (default:false)**  $t_1$  makes impossible  $t_2$  if  $t_1$  writes makes  $t_2$  preconditions false.

When there is more than one possible execution path in each program (and several groups of information are obtained from each transaction), it is necessary to analyze the different possible combinations — a default value is assumed if different results are obtained from different combinations. A default value is also assumed when it is not possible to compare the semantic descriptions (e.g. two database accesses use conditions that do not include the same column — we address this problem later in this paper).

The default values (shown before) were set to guarantee the correctness of the reconciliation algorithms. However, the *overlap* relation is a special case: assuming that two transactions do not overlap do not sacrifice correctness, as overlap is only used to partition the set of transactions to reconcile into smaller subsets to improve reconciliation performance. If these subsets are not disjoint, correctness is not sacrificed, although it may be impossible to achieve the best reconciliation result as it is impossible to create some relevant schedules.

### 3.3. Examples

Now, we present two examples from typical mobile application that show how relations are inferred from the code of mobile transactions written in PL/SQL.

#### Mobile sales application:

In the first example, we consider two types of mobile transactions submitted in the context of a mobile sales application. The first, presented in figure 3, adds a new order. The second, cancels an order if it has not been processed yet, as shown in figure 4.

## RENPAR'15 / CFSE'3 / SympAAA'2003

### La Colle sur Loup, France, 15 au 17 octobre 2003

```
--- RESERVES ROOM 'Ballroom A' FOR DAY '16-FEB-2002'
BEGIN
  SELECT count(*) INTO cnt FROM datebook WHERE day='16-FEB-2002' AND room='Ballroom A';
  -- cnt = read(datebook,day='16-FEB-2002' AND room='Ballroom A',count(*))
  IF (cnt = 0) THEN
    -- precondition(read(datebook,day='16-FEB-2002' AND room='Ballroom A',count(*))=0)
    INSERT INTO datebook VALUES( '16-FEB-2002', 'Ballroom A', 'Demo BLUE THING');
    -- insert(datebook,(day,room,info),'16-FEB-2002','Ballroom A','Demo BLUE THING')
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 5: Information extracted from a mobile transaction that inserts an appointment in a shared calendar.

```
--REMOVE RESERVATION
BEGIN
  SELECT count(*) INTO cnt FROM datebook WHERE day='16-FEB-2002' AND room='Ballroom A' AND info='Demo BLUE THING';
  -- cnt = read(datebook,day='16-FEB-2002' AND room='Ballroom A' AND info='Demo BLUE THING',count(*))
  IF (cnt > 0) THEN
    -- precondition(read(datebook,info='Demo BLUE THING' AND
    -- day='16-FEB-2002' AND room='Ballroom A',count(*)) > 0)
    DELETE FROM datebook WHERE day='16-FEB-2002' AND room='Ballroom A';
    -- delete(datebook,day='16-FEB-2002' AND room='Ballroom A')
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 6: Information extracted from a mobile transaction that cancels an appointment in a shared calendar.

The information that can be extracted from each mobile transaction is shown, as comments, in the figures<sup>1</sup>. Let  $a$  be an add transaction and  $c$  be a cancel transaction acting on the same product. The system infers the following relations.  $a$  and  $c$  do not commute, as  $a$  reads the stock of product with  $id = 80$  and  $c$  writes (updates) it.  $c$  helps  $a$ , as adding a positive value to the stock helps the stock to be bigger than some constant. No other relations exist. The inferred relations are the expected ones, leading the reconciler to execute the cancel transaction before executing the add transaction, thus improving the chances of committing both transactions.

As expected, two mobile transactions adding a new order for the same product do not commute and each one prejudices the other, as subtracting a positive value from the stock reduces the chances of the stock being larger than some constant.

Two mobile transactions cancelling the same order do not commute and each one makes impossible the other, as setting the order state to *cancelled* makes the precondition expressed in the transaction false. This is the expected behavior as an order can only be cancelled once.

Two mobile transactions acting on different products and orders commute because they do not access the same data items.

---

<sup>1</sup> As explained before, this information is automatically extracted from the code of the mobile transaction: presenting it as comments in the code of transactions is just a convenience for showing the information that can be extracted from each statement.

### Shared calendar:

In the second example, we consider typical transactions submitted in the context of a shared calendar used to maintain meeting room reservations. In the transaction presented in figure 5, a new reservation for a room is added if possible. In the transaction presented in figure 6, an existing reservation is removed.

The systems infers the following relations. Any two transactions that refer non-overlapping reservations commute as they access different data items. Transactions with overlapping transactions do not commute. For these transactions, the following additional relations are inferred. For two transactions that insert a new reservation, each one makes impossible (and prejudices) the other, as the inserted record leads the precondition to be false.

For two transactions that cancel the same reservation, each one makes impossible (and prejudices) the other, as the delete statement guarantees that no record exists that satisfy the specified precondition.

For a transaction,  $a$ , that inserts a new reservation and a transaction,  $c$ , that cancels a reservation, two cases should be considered. If both refer to the same reservation (i.e. the value of the info field is equal),  $a$  helps  $c$  because the inserted record guarantees that the reservation exists (and the precondition of  $c$  is true). Furthermore,  $c$  also helps  $a$  because removing the reservation guarantees that it is possible to schedule a new reservation for that time and place. The reason for this lies in the fact that it might exist or not an identical reservation in the initial database. In this case, the static relations do not help the reconciler to decide which transaction should be executed first. Assume, without loss of generality, that  $c$  should be executed before  $a$  and the reconciler selects  $a$  first. In this case,  $a$  fails and it is moved to the set of bad transactions. However, after executing  $c$ ,  $a$  is removed from the set of bad transactions and it will be selected in one of the following steps. Thus, both transactions are successfully executed.

If  $a$  and  $c$  refer to different reservations (i.e. the value of the info field is different),  $c$  helps  $a$  as removing the reservation guarantees that it is possible to schedule a new reservation for that time and place. However, unlike before,  $a$  does not help  $c$  because the inserted record does not satisfy the condition of the select statement — the info field is different. Thus, as expected, the reconciler will cancel appointments before trying to schedule new appointments for the same time and place.

## 4. Extensions and on-going work

In this section we discuss some extensions to the basic solution described so far.

### 4.1. Comparison of semantic descriptions

Sometimes it is impossible to compare precisely two semantic descriptions because they include conditions over different columns of the same table or indirections (i.e., they use values read in previous *select* statements). This problem could be solved reading from the database the extra information needed to solve indirections or to *compare* conditions precisely. As, in many of these cases, the needed information cannot be modified by any transaction under reconciliation, it is possible to use this information to infer static relations (reading it from the database and using it as a constant value).

## 4.2. Loops

The modification of multiple records using a single SQL statement is addressed by the basic approach described so far. However, to process transactions that include loops (controlled by a cursor or not), we need to consider the following two cases. First, when the sequence of values for the variable that controls the loop is (directly or indirectly) known (e.g. in "for  $i$  in 1..3 loop",  $i$  will be assigned the values 1,2 and 3), the analysis is trivial — it is just necessary to evaluate the code inside the loop once for each value of the control variable. Second, when the sequence of values for the control variable is not known, it may be very difficult or impossible to extract precise information. We are currently investigating this problem.

## 4.3. Complex transactions and log constraints

Our semantic inference module may have problems to handle very long transactions that access many data items, as these transactions may end up having all types of relations with (many of) the other transactions. To avoid this problem, the SQLIceCube API allows programmers to submit complex operations as a composition of smaller transactions linked by suitable log relations. Moreover, the SQLIceCube system may also pre-process each transaction received, splitting it into smaller transactions linked by suitable log relations. For example, a transaction composed by a set of alternative actions coded as a sequence of if instructions can be split into an ordered set of alternative transactions (implemented with the *alternatives* and *weak predecessor-successor* log relations), each one with a single alternative action. Other log relations can also be automatically inferred between transactions submitted in the same mobile client — e.g. if a transaction reads a data item inserted by a previous transaction, the *strong predecessor-successor* log relation can be added between the older and the new transaction.

## 5. Final Remarks

SQLIceCube is a general-purpose semantics-based reconciliation system for mobile databases. The system includes a semantic inference module that automatically extracts semantic information from the code of mobile transactions. This information, exposed as a set of static relations among transactions, is used by the basic reconciler to create near-optimal reconciliation results. To our knowledge, SQLIceCube is the first to automatically infer the semantic information needed in reconciliation. This approach simplifies the use of semantics-based reconciliation by eliminating the overhead of writing long/complex rules to expose the semantics of operations. SQLIceCube basic reconciler is based on the IceCube reconciler [14], the first to treat reconciliation as an optimization problem. Some changes have been made to accommodate mobile transactions.

SQLIceCube has been initially developed to be used in Mobisnap, as a complement of the reservation (conflict-avoidance) mechanism [13]. However, SQLIceCube is an architecture-independent reconciliation system that can be easily used in other SQL-based systems — e.g. in Bayou [17], it could be used in the primary server to order updates received in epidemic communication sessions (if updates were expressed in PL/SQL).

## Bibliographie

1. Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proc. 29<sup>th</sup> ACM Symp. on Principles of programming languages*, 2002.

## RENPAR'15 / CFSE'3 / SympAAA'2003

La Colle sur Loup, France, 15 au 17 octobre 2003

2. Per Cederqvist, Roland Pesch, et al. Version management with CVS, date unknown. <http://www.cvshome.org/docs/manual>.
3. Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. VLDB'96*, 1996.
4. Dawson Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18<sup>th</sup> Symp. on Op. Sys. Principles*, 2001.
5. H.R. Nielson F. Nielson and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
6. Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM Transactions on Database Systems (TODS)*, 20(3):288–324, 1995.
7. Peter Keleher. Decentralized replicated-object protocols. In *Proc. 18<sup>th</sup> Symp. on Princ. of Distr. Comp. (PODC)*, 1999.
8. P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. USENIX Winter Tech. Conf.*, 1995.
9. David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29<sup>th</sup> ACM Symp. on Principles of Programming Languages*, 2002.
10. Jon Munson and Prasun Dewan. A flexible object merging framework. In *Proc. Conf. on Comp.-Supported Cooperative Work (CSCW)*, 1994.
11. Oracle. PL/sql user's guide and reference - release 8.0, June 1997.
12. Shirish Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. In *Proc. 15<sup>th</sup> Int. Conference on Data Engineering*, 1999.
13. Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. A middleware approach to independently guarantee the results of operations in a mobile database system. In *Proc. Mobisys 2003 (to appear)*, 2003.
14. Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.
15. Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, March 2002.
16. Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. Conf. on Comp.-Supported Cooperative Work (CSCW)*, 1998.
17. Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15<sup>th</sup> Symp. on Op. Sys. Principles*, 1995.
18. William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.