Database Evolution under Non-deterministic and Non-chronological Updates

Yannis Dimopoulos* Antonis Kakas† Suryanarayana M. Sripada‡

Abstract

Data base applications that model aspects of the real-world, should keep track of the changes that are happening in it, and reflect these changes as accurately as possible in the database. If the information about changes, in the form of *updates*, is not supplied to the database in an ideal way, the database needs to have mechanisms that assimilate correctly, whenever possible, this new information.

We propose a model called FlexUp that addresses two cases of non-ideal information supply. The first is the well-known, in the context of deductive databases, problem of *non-deterministic updates* of intensional predicates. The second is the problem of *non-chronological updates*, i.e. updates that come to the system in an order that does necessarily reflect the order in which the corresponding changes happen in the real-world.

The main features of FlexUp are (a) it is conservative in the sense that in case of ambiguity it avoids realizing updates in ad hoc ways (b) it satisfies ACID-like properties and therefore it is in accordance with traditional *transaction systems*.

1 Introduction

Databases in general and deductive databases in particular try to model the real-world and keep track of the changes that are happening in it. The information that the database holds forms the basis for taking decisions by the application system, and therefore it is important that the state of the database reflects the state of the real-world as correctly as possible. When the changes that happen in the world are provided to the database in an ideal way, it is easy to satisfy this criterion of correctly reflecting the state of the world. However, there are cases where this information about the changes in the world is not always ideally provided. The information may be incomplete or it may arrive at the database in an order different from that of the changes in the world. This problem is particularly acute when we have intensional (view) relations in our database which can be updated in several different ways. Let us examine the following illustrative example.

Example 1.1 *Consider the database in an organisation, comprising of the extensional relations Employee(EmployeeName, Department) and Manager(ManagerName, Department) and the intensional relation Manages(ManagerName, EmployeeName) defined as follows:*
$Manages(MName, EName) \leftarrow Manager(MName, Dept), Employee(EName, Dept)$
Assume the following data in the extensional relations to start with:
Manager(Bob, Audio), Manager(Mary, Video),
Employee(Tom, Audio), Employee(Peter, Accounts)
Now consider the update request U_1*: Add Manages(Mary, Tom)*

*Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany.
†Department of Computer Science, University of Cyprus, Nicosia, Cyprus.
Corresponding author. Email: antonis@turing.cs.ucy.ac.cy
‡Department of Computer Science, University of Aachen, Aachen, Germany.

The intensional relation *Manages* can be updated by any combination of (basic) updates (add and delete) on the extensional relations (*Manager*, *Employee*) that would bring the database in a state that makes the body of the rule hold. Therefore, the specific update request U_1 can be realised in at least 3 ways:

• Tom moved to the Video department: { delete Employee(Tom, Audio), add Employee(Tom, Video) }.

• Mary moved to the Audio department: { delete Manager(Bob, Audio), add Manager(Mary, Audio) }.

• Both Tom and Mary moved to a new department, say Marketing: { delete Manager(Mary, Video), delete Employee(Tom, Audio), add Manager(Mary, Marketing), add Employee(Tom, Marketing)}.

Such an update request U_1 is *non-deterministic* since it is not clear which of the alternatives above reflects the actual change that has occurred in the world and therefore should be used to realize the update request.

The problem of non-deterministic view updates is well-studied in the context of deductive databases (see e.g. [11, 2, 7, 20, 12, 1]). In these works the primary concern is that of realizing a single update working under the implicit assumption that one of the many different realizations can be choosen at the time when the update is presented to the database. Several important criteria for tackling the problem of non-deterministic updates such as preferring minimal updates have been considered e.g. [9, 13, 14, 6]. Although these can be useful, they do not solve the problem completely and can sometimes lead to ad hoc solution. Any arbitrary mechanism for resolution of ambiguity would mis-represent the state of affairs in the real-world, thus compromising the criterion of faithful representation of the world by the database. In this paper we will assume that the information about the changes in the world may be distributed over several updates, and the update that can make them deterministic has not been presented to the database yet. We thus need to consider a sequence of updates and the evolution of the database under this sequence. In [19] the notion of cumulative updates is proposed, where update ambiguities are resolved by combining information from several updates. However, [19] does not propose a transaction model that enables such update mechanisms to be implemented.

Another important case of non-ideal supply of information to the database is when this arrives in an order that does not necessarily correspond to the order in which changes have occurred in the real-world. In such a case we say that the database is receiving *non-chronological updates*. An underlying assumption of traditional update and transaction models is that updates are chronological (full fledged temporal databases are an exception to this rule [18]). This assumption may be too strong for databases in advanced applications. To illustrate the problem consider again the above example and assume that after U_1 we get another update $U_2 = \{delete\ Manager(Mary, Video),\ add\ Manager(Mary, Marketing)\}$. If this change in the world recorded by U_2 occurred before that of U_1 then U_1 must be realized in the third way. But if U_2 is treated by the database as a later change, because it is presented to the database after U_1, there is no guarantee that U_1 will be resolved in the same way. Thus, a database after a series of updates may or may not reflect the state of the world, depending upon the order in which such updates are committed.

It is clear from the foregoing that both of the above scenarios require updates to be handled in a more flexible way, by exchanging information between different update requests. In this paper, we develop a model, called FlexUp, suitable for the non-ideal situations of non-deterministic and non-chronological updates by allowing more flexibility in realising them. The basic idea is to allow non-deterministic update requests the possibility to wait (*pending* updates) for more information that will make them deterministic. When a new update is submitted to the system, carrying some deterministic information, this may help some of the pending updates to become deterministic and therefore commit. The model also uses information concerning the time at which the update becomes valid in the real-world, referred to as *valid time* (in temporal database parlance [18]), to allow updates to commit irrespective of the order in which they are submitted while respecting their chronological ordering.

An indirect effect of the non-ideal supply of information to the database is the fact that an update request may be inconsistent at the time when it is presented to the database due to some information that has not arrived at the database yet. The FlexUp model allows such updates to pend in the same way as the non-deterministic updates, waiting for information that would render them consistent. If such information arrives then the update can be realised on the database otherwise it is rejected as in the standard database system.

Looking at the problem of deductive database updates from the point of view of *transactions*, we can identify some important criteria that an update model should satisfy. An update can be seen as a special case of a write-only transaction (i.e. a transaction that does not depend on the state of the extensional database) that changes the state of the database. Under this view, each update/transaction should satisfy ACID-like properties developed by the traditional database community [10]. It turns out that our model satisfies the atomicity and consistency properties. Therefore, the FlexUp model is in accordance with traditional transaction models. Each transaction/update either commits or aborts as a single unit without any dependencies or post conditions. In this sense our approach also contributes ideas for developing transaction models capable to cope with situations of non-ideal information supply. This is an important issue as many kinds of transaction models have been proposed, motivated by advanced applications from various domains [8, 17, 15, 16]. These models attempt to relax the ACID properties of transactions in order to achieve greater functionality and higher efficiency [3]. However, most of the commercial database systems rely only on the standard transaction model satisfying ACID properties [10].

It should be pointed out that, in general, a database is updated through more complex forms of transactions that could involve reading from the database and depending on what is read, different sets of updates are selected. In these cases the problem of handling non-ideal information supply in the database, becomes more complex, and an extension of FlexUp would be needed.

Summarizing, the main features of the FlexUp model are the following.

- It handles non-deterministic updates by keeping these pending until complementary information from other update requests helps resolve the ambiguities.
- It exploits the valid time to handle non-chronological updates.
- Guarantees the consistency of the database according to the specified integrity constraints.
- It enjoys ACID-like properties.

2 The Update model

In this section we describe the FlexUp model and illustrate its behaviour with examples. We start by introducing some basic concepts that are used by FlexUp.

A database contains a set of *extensional* relations. An *extensional* database is a set of ground facts on these extensional relations. A database may also have *view or intensional* relations defined from the extensional relations. The intensional relations are not explicitly stored in the database.

Definition 2.1 *A basic update operation on the database has the form add(p) or delete(p) where "p" is a ground fact of an extensional relation. When add(p) (resp. delete(p)) is effected, the resulting extensional database contains "p" (resp. does not contain "p").*

An update on extensional relations is a set of basic update operations together with an associated time called the valid time of the update. Updates on view (intensional) relations must be translated into a set of basic update operations. Such a translation is not always unique, but instead there can be many different sets of basic operations for the same update (see e.g. [12]). This is the phenomenon of *non-determinism* in the update requests. For instance, in the context of updating intensional relations through abduction (e.g. [12]), every abductive explanation of the update gives us a different way of realizing this update. Abstracting from the details of how view update requests are related and translated to basic operations (these details are not important in the study of this paper) we adopt the following definition of an update on the database.

Definition 2.2 *A* realization *is a set of basic update operations. An* update *is a pair $U = < R, vt >$, where R is a set of realizations and vt is a time point called the valid time of U.*

The *deterministic part* of an update is the set of basic operations that are common in all of its realizations. Hence, no matter how this update will be realized, once it commits, the basic update operations in the deterministic part will be effected on the database.

Definition 2.3 *The* deterministic part *$DP(U)$ of an update $U = < R, vt >$ is defined to be $DP(U) = \bigcap_{i=1}^{n} R_i$, where $R_1, \ldots R_n$ are the realizations of U contained in R.*

We will assume that for any set of updates with the same valid time the union of their deterministic parts does not contain both $add(p)$ and $delete(p)$ for some p.

Within the FlexUp model an update U, can be in one of three states: *Pending, Commit,* or *Abort.* A FlexUp update is kept pending if it is non-deterministic or inconsistent, until further updates provide enough information to resolve the non-determinism or inconsistency in the pending update. Upon the arrival of another update the FlexUp procedure tries to identify a set of updates that are mutually deterministic and consistent. Then these updates move into the commit status and are committed. The new committed updates start this process again with the possibility of other updates to commit. First we give some preliminary definitions.

Definition 2.4 *A* state *of a database is a triple $S_i = (DB_0, CU, PU)$, where DB_0 is an extensional database, CU is a set of pairs of the form $< U_j, R_j >$, where U_j is an update committed via the realization R_j, and PU is a set of pending updates.*

The extensional DB_0 is to be thought as the state at some initial time of reference and CU (respectively PU) the sets of committed (respectively pending) updates at some time after this initial time.

Definition 2.5 *The extensional database wrt some database state* $S = (DB_0, CU, PU)$ *and some valid time* t*, denoted by* DB_t^S*, is the extensional database obtained after committing, in their valid time order, all the realizations in* CU *that are associated with updates with valid time less or equal to* t*. The current extensional database, denoted by* DB^S*, is the database* $DB_{t_f}^S$*, where* t_f *is the greatest valid time amongst the committed updates* CU*.*

We next give the definition of the key notion of *committable* updates which defines which (pending) updates have become ready to commit.

Definition 2.6 *Let* $S = (DB_0, CU, PU)$ *be a state of the database. Then a set of updates* $U \in PU$ *is* committable *wrt the state* S *iff the following conditions hold:*
a) All updates in U *refer to the same valid time* t*,*
b) Every update $U_i \in U$ *has a realization* r_{ij} *such that for every* $add(p) \in r_{ij}$ *(resp.* $delete(p)$*) either* $p \in DB_t^S$ *(resp.* $p \notin DB_t^S$*) or* p *belongs to the deterministic part of another update* $U_k \in U$*.*
c) Let U' *be the set of pairs* $< U_i, r_{ij} >$*, where* $U_i \in U$ *and* r_{ij} *is the realization of the update* U_i *selected in (b). Then for every valid time* $t' \geq t$ *and the state* $S' = (DB_0, CU \cup U', PU - U)$*,* $DB_{t'}^S$ *is consistent.*

This notion of committable combines two elements. The first element, captured by conditions (a, b), simulates the conditions required to hold for the whole set of updates in U to be accepted (and committed), by a conventional update model, when these are not distributed over the different updates but are given all together as one update to the database DB_t referring to the valid time "t" of interest. It therefore requires that any non-determinism that may exist in the updates of this set can be resolved using the collected information distributed over the different updates in U and information from DB_t. The second element (condition c) checks that the realizations of the updates in the set U do not violate any of the integrity constraints at any valid time starting from the valid time of U up to the latest valid time.

Example 2.7 *Consider a database that contains the integrity constraint* $\neg(s, r)$*, the fact* r *in its initial extensional state* DB_0*, with no committed updates yet, and a set of pending updates* $U_1 =< \{(add(p), add(q)), (add(p), add(l))\}, 4 >$*,* $U_2 =< \{(add(m), add(s)), (add(m), add(t))\}, 4 >$*,* $U_3 =< \{(add(s))\}, 4 >$*. Assume that the latest update that has arrived at the database is* $U_4 =< \{(add(q))\}, 4 >$*.*

We first note that U_1 and U_2 are pending because they are non-deterministic and that U_3 is pending because it is inconsistent. When U_4 arrives then the set of updates $\{U_1, U_4\}$ is committable since both refer to valid time 4, the first realization of U_1 belongs to the deterministic part of U_4, and if they are realized on DB_0, the extensional database remains consistent. Note that U_3 can not be included in the set of committable updates, as it violates, together with the fact r, the integrity constraint. Consequently, U_2 can not also be committable since it remains non-deterministic. Notice that if a new update $U_5 =< \{(delete(r), 3 >$ comes after and commits, U_3 and so also U_2 become committable.

We note here that in the definition of committable we require that the valid time of different updates must be exactly the same for these to be able to exchange information among their deterministic parts. This is a formal idealization of a notion of "closeness" of valid times that is dependent on the specific domain of the application for which the database is built. In general, we can employ a suitable temporal reasoning mechanism to project (forwards and backwards) information from updates along the valid time axis.

The basic FlexUp model consists of two main procedures. The first one, called Process-New, is activated when a new update arrives at the system. The procedure checks if the information that the update carries, helps to make some of the pending updates (with valid time equal to the valid time of the new one) committable. When this happens the committable updates are committed and the sets of pending and committed updates are changed accordingly. In detail, the procedure Process-New is as follows.

Assume that the database starts at some initial time with an extensional state DB_0 and is currently in the state $S = (DB_0, CU, PU)$ when the new update, U_{new}, arrives.

Procedure Process-New (U_{new}, PU, CU)
t_e := valid time of T_{new}; $PU := PU \cup U_{new}$;
CS_{t_e} := Subset of updates of PU with valid time t_e that are committable wrt $S = (DB_0, CU, PU)$;
If $CS_{t_e} \neq \emptyset$ **then**
begin
For each $U_i \in CS_{t_e}$ do
$CU = CU \cup < U_i, R_i >$, where R_i is the deterministic part of the update U_i;
$PU := PU - CS_{t_e}$;
Process-Rest (PU, CU);
end

Note that although we do not add the whole realization of a committed update (but only its deterministic part), the rest of the realization belongs to the deterministic part of the other updates which will be committed together with this. As we will see in Section 3, this means that updates that commit have the atomicity property.

When a new set of updates is committed by the procedure Process-New, the state of the database changes. This means that other updates pending either due to non-determinism or inconsistency before this activation of procedure Process-New, may now they also become committable. This job is carried out by the procedure Process-Rest which checks the whole set of the pending updates (in their valid time order) to determine if some of them can commit. This is repeated until no additional updates become committable.

Procedure Process-Rest (PU, CU)
finished:=false;
While not finished **do**
begin   finished:=true;
t_e := earliest valid time in PU; t_{last} := latest valid time in PU;
While $t_e \leq t_{last}$ **do**
begin
CS_{t_e} := Subset of updates of PU with valid time t_e that are committable wrt $S = (DB_0, CU, PU)$;
If $CS_{t_e} \neq \emptyset$ **then**
begin
finished:=false;
For each $U_i \in CS_{t_e}$ do

$CU = CU \cup < U_i, R_i >$, where R_i is the deterministic part of the update U_i;
$PU := PU - CS_{t_e}$;
end;
$t_e := t_e + 1$;
end; **end**;

The model as described so far keeps track of all the updates submitted to the system (committed and pending). This might be unrealistic for practical applications. For this reason the system may periodically "time out" all the pending updates with valid time earlier than a time t by removing them from the set of pending updates and aborting them. Also all committed updates with valid time earlier than t are removed form the set of committed updates. This time t then becomes the next initial time and the extensional database DB_t the next initial extensional database. Any new update that arrives and has valid time earlier than t will be aborted without further processing.

Let us illustrate the behaviour of FlexUp with the following examples. The first example is a simple example that shows how non-determinism is resolved using information from later updates.

Example 2.8 *Consider a database of a company that consists of the view relation* new-stock(id, ammount) *where* id *is the identification number of a new stock and* "ammount" *the quantity of the stock. Furthermore assume that there are three extensional relations,* stock(id, ammount) *that records the quantity of each stock, the relation* retail(id) *that records which stock is of retail type, and the relation* wholesale(id) *that records the stock of wholesale type.*

Suppose that at valid time 1 a new update U_1 arrives where we want to update the database with the fact, $new - stock(id123, 50)$, on this view relation. Also assume that the type of stock id123 has not been decided yet, and therefore there are two possible realizations of this update, $U_1 =< \{(add(stock(id123, 50), add(retail(id123)), (add(stock(id123, 50), add(wholesale(id123))\}, 1 >$. So the update is non-deterministic and is added to the pending updates. After some time we decide that id123 will be of wholesale type and give the update $U_2 =< \{(add(wholesale(id123))\}, 1 >$ which then renders U_1 and U_2 committable.

The second example given below is representative of the way non-chronological updates are handled within FlexUp model and how updates can be pending due to inconsistency.

Example 2.9 *Consider a database that consists of the extensional relation* rank(Name, Rank) *that records the rank of the faculty of a university and the relation* pay(Name, Document, Ammount) *that records reimbursements of faculty traveling expenses. The database also contains the integrity constraint that only tenured faculty get reimbursed for traveling and the fact* rank(Tom, Assistant).

Assume that at valid time 5 Tom gets promoted to Associate Professor but the relevant update is not given to the system (until e.g. the Senate approves his promotion). At time 7 Tom goes to a conference and the update request $U_1 =< \{add(pay(Tom, Receipt - 101, 1200))\}, 7 >$ comes to the system. This update can not be realized since Tom is still an Assistant Professor in the database and the integrity constraint is violated. So U_1 is added to the pending updates. At some later point the update $U_2 =< \{(add(rank(Tom, Associate)),\ delete(rank(Tom, Assistant)))\}, 5 >$ comes and it commits. Now update U_1 can also commit since it is now consistent wrt the constraint.

We now give a more complex example that demonstrates the general behaviour of FlexUp model in complicated situations.

Example 2.10 *Consider the following set of updates:*
$U_1 =< \{(add(p), add(s)), (add(p), add(t))\}, 1 >$
$U_2 =< \{(add(q))\}, 2 >$ $U_3 =< \{(delete(p), add(r))\}, 2 >$
$U_4 =< \{(add(s), add(c)), (add(s), add(d))\}, 1 >$
$U_5 =< \{(add(m))\}, 4 >$ $U_6 =< \{(add(c))\}, 1 >$
$U_7 =< \{(delete(a), add(b))\}, 3 >$ $U_8 =< \{(add(k))\}, 2 >$
Suppose that the initial extensional database is $DB_0 = \{a\}$ *and that the database has the integrity constraints* $\neg(a, m)$ *and* $\neg(b, k)$*. Also assume that the updates are submitted in the order* $(U_1, U_2, \ldots, U_8)$.

When U_1 arrives to the system it becomes pending since it is non-deterministic and there is not enough information to resolve this non-determinism. Hence the set of pending updates becomes $PU = \{U_1\}$. Then U_2 comes, with valid time 2, and since it is deterministic and consistent it commits. The set of committed updates CU becomes $CU = \{U_2\}$. Then U_3 arrives which is deterministic and consistent and commits. Hence, $CU = \{U_2, U_3\}$ and the current (wrt valid time 2) extensional database is $DB = \{a, q, r\}$. When U_4 is submitted FlexUp sees that its non-determinism can not be resolved with the information we have, hence $PU = \{U_1, U_4\}$. Note that U_4 carries the deterministic information $add(s)$ that can make U_1, which is pending, to commit, but this is not possible since U_4 itself is not committable.
When U_5 arrives, although this is a deterministic update it is inconsistent since a was in the database before valid time 4 and adding m will violate the constraint $\neg(a, m)$. Hence U_5 becomes pending and $PU = \{U_1, U_4, U_5\}$. When U_6 comes we see that U_4 becomes deterministic (since U_4 can now commit on its first realization using deterministic information provided by U_6) and in turn U_1 becomes also deterministic. Hence all of U_1, U_4, U_6 are committable since also they do not violate any of the integrity constraints. The set of committed updates becomes $CU = \{U_1, U_4, U_6, U_2, U_3\}$ and the current (wrt valid time 2) extensional database is $DB = \{a, q, r, s, c\}$. Now U_7 arrives which is deterministic and consistent and commits. This then makes the pending update U_5 consistent, since a is deleted, and therefore commits as well. The current extensional database is $DB = \{q, r, s, c, b, m\}$ and $PU = \emptyset$. Next U_8 is submitted which is deterministic but it can not be committed since it is inconsistent with the database of valid time 3 that contains (due to U_7) b. Hence it will be added to the pending updates.

3 Properties of the FlexUp Update Model

In this section we present some of the formal properties of FlexUp, and discuss briefly their significance.

We first note that FlexUp keeps the current extensional database consistent. Furthermore, each extensional database referring to any valid time, is also consistent.

Proposition 3.1 *For any resulting state of the database* $S = (DB_0, CT, PT)$ *and for any valid time* t*, the extensional database* DB_t *is consistent.*

The resolution of non-determinism by FlexUp *does not involve any arbitrary choice* within the set of committed updates. The realization selected for a particular update is supported by the deterministic information in other committed updates.

Proposition 3.2 *Let DB denote the extensional database resulting after a sequence of updates SU has been submitted to FlexUp on an database with initial state DB_0. Then $p \in DB$ (resp. $p \notin DB$) if $p \in DB_0$ (resp. $p \notin DB_0$) or add(p) (resp. delete(p)) belongs to the deterministic part of of a committed update $U \in SU$.*

Another property of FlexUp is that this assimilates information about changes in the world while *respecting the time order in which these changes happened in the world* (their valid times). The following two results demonstrate this property and show an order independence of FlexUp amongst the committed updates.

Proposition 3.3 *Let SU be a sequence of updates ordered by their valid times such that if submitted to FlexUp all the updates in SU commit resulting to a final extensional database DB_f. Let SU' be a permutation of SU submitted to FlexUp. Then FlexUp will commit all the updates in SU' resulting to the same final extensional database DB_f.*

Let us denote by $Do(DB, R)$ the extensional database obtained when effecting a realization R on the database DB.

Theorem 3.4 *Let SU be a sequence of updates submitted to FlexUp on an initial state DB_0. Let also $U' \subseteq SU$ be the updates that have been committed within a time period t and $(U_1, \ldots, U_n)$ a permutation of the updates in U' ordered by their valid time. Then there exists a sequence $(R_1, \ldots, R_n)$ where each R_i is a realization of the update U_i, such that the extensional database, DB, at the end of period t, is $Do(\ldots Do(Do(DB_0, R_1), R_2) \ldots R_n)$.*

This last theorem shows that the set of updates U' can be given in any order but when they commit their effect is as if they were given in their valid time order. The theorem also shows that at any time *the set of committed updates satisfies the atomicity property*, in the sense that each committed update has in effect been completely realized (by one of its realizations) on the initial database.

The above results can be put together to give the following corollary that characterises uniquely the extensional database at any time.

Corollary 3.5 *Let SU be a sequence of updates submitted to FlexUp on an initial database DB_0. Let also $U' \subseteq SU$ be the updates that have been committed within a time period t and $(U_1, \ldots, U_n)$ a permutation of the updates in U' ordered by their valid time. The extensional database DB, at the end of the period t is equal to $Do(\ldots Do(Do(DB_0, DP(U_1)), DP(U_2)) \ldots DP(U_n))$, where $DP(U_i)$ is the deterministic part of the update U_i.*

Hence, FlexUp simulates, for the committed updates, an ideal supply of information.

Finally, we note that when information arrives at the database in an ideal way so that all the updates can commit under the standard update model, FlexUp will exhibit the same behaviour. For example, if all the updates arrive in their valid time order and are all consistent (but not deterministic) then all updates committed under the standard model will also commit under FlexUp.

4 Concluding remarks

In this paper we introduced the FlexUp model that supports non-deterministic and non-chronological updates. FlexUp resolves update ambiguities and assimilates information in non-chronological order. We showed that FlexUp has the following properties: (a) It does not make any arbitrary choices in its resolution of non-determinism. (b) It assimilates information about changes in the world while respecting the chronological order of these changes. (c) When the update requests are chronological it performs as well as the standard model. (d) It satisfies ACID-like properties.

We emphasise that FlexUp is a conservative extension of the standard update model: we have used a very simple temporal reasoning model to transfer information from one update to another. In part, this model was motivated by the desire to stay close to traditional transaction systems. But the main reason for its simplicity is the pragmatic need for the high-level of reliability and accuracy of the information that is held by a database: in practice, we can not have a recovery mechanism from incorrect data that has been extracted from the database and acted upon. Our work has shown that even with such simple temporal reasoning mechanisms it is a non-trivial task to define a suitable update model with the desired database properties. Further study is needed to understand how more advanced forms of temporal reasoning, that would allow a more general form of exchange of information between updates at different valid times can be incorporated in the update model allowing the same degree of reliability. The non-monotonic nature of such frameworks often leads to the need of belief revison that may not be possible in the pragmatic context of databases and hence these frameworks will need to be specially adapted and refined for their use in database evolution.

References

[1] Halfeld Ferrari Alves, M., Laurent, D., Spyratos, N., Update Rules in Datalog Programs. *Proc. 2nd Int. Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'95*, LNAI 928, Springer Verlag, 1995.

[2] Bry, F., Intensional updates: abduction via deduction. *7th ICLP*, Jerusalem,561–575, 1990.

[3] Chrysanthis, P.K., Ramamritham, K. ACTA: A framework for specifying and reasoning about transaction structure and behaviour. In *Proc. ACM SIGMOD Conference*, 1990.

[4] Console, L., Sapino, M.L., Theseider Dupré, D., The role of abduction in database view updating. *Journal of Intelligent Information Systems*,Vol 4, pp. 261-280, 1995.

[5] Date and White *An introduction to DB2*, 1989, Prentice Hall.

[6] Dekhtyar, M., Dikovsky, A., Spyratos, N., On Conservative Enforced Updates. *Proc. 4th Int. Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'97*, Springer Verlag, 1997.

[7] Decker, H., Drawing Updates from Derivations *In proceedings of ICTP'90.*

[8] Elmagarmid, A.K. (ed.). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann.

[9] Fagin, R., Ullman, J. and Vardi, M., On the Semantics of Updates in Databases *Proc. 2nd ACM PODS*, 1983.

[10] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques* Morgan Kaufmann, 1993.

[11] Guessoum, A. and Lloyd, J.W., Updating Knowledge Bases *New Generation Computing*, 1990.

[12] Kakas, A. C., Mancarella, P., Database updates through abduction. *Proc. 16th International Conference on Very Large Databases, VLDB'90*, Brisbane, Australia (1990)

[13] Katsumo, H., Mendelzon, A., Propositional Knowledge Base Revision and Minimal Change *Artificial Intelligence, Vol 52, 1991.*

[14] Marek, V.W., Truszcynski, M. Revision Programming, Database Updates and Integrity Constraints *International Conference on Database Theory, ICDT'95* LNCS 893, 1995.

[15] Moss, J.E.B. Nested Transactions: An approach to reliable distributed computing. Ph.D.Thesis, MIT, April 1981.

[16] Nodine, M., Zdonik, S. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings 10th VLDB Conference*, 1984.

[17] Pu, C., Kaiser, G., and Hutchinson, N. Split Transactions for Open-Ended activities. In *Proc. 14th VLDB Conference*, August 1988.

[18] Snodgrass, R. et al. *The TSQL2 Language*, Kluwer Publishers, 1995.

[19] Sripada, S.M. and Wuethrich, B. Cumulative Updates. In *Proc. of the 20th VLDB Conference*, September 1994.

[20] Tomasic, A., View Update Annotation in Definite Deductive Databases *Proc. ICDT'88*, Springer Verlag, 1988.