How to Execute ULTRA Transactions

Carl-Alexander Wichert
Alfred Fent
Burkhard Freitag

Universität Passau
Fakultät für Mathematik und Informatik
D-94030 Passau, Germany

e-mail: {wichert,fent,freitag}@fmi.uni-passau.de
Tel: +49 (0)851 509 3019   Fax: +49 (0)851 509 3002

Universität Passau
Technical Report MIP-9812

Abstract

In [WF97, WFF98a, WFF98b] we describe the language ULTRA that has been designed for the specification of database updates in a modular, rule-based fashion. We provide a model-theoretic semantics based on deferred updates, which means that update requests are determined and collected in an evaluation phase without changing the database state and the corresponding updates are performed in a subsequent materialization phase. As shown in [WFF98a, WFF98b], the semantics can be implemented using an optimistic transaction processing method. From a practical point of view, however, a straightforward implementation of the logical semantics is not satisfactory. In this paper we describe some of the arising problems and outline a flexible system architecture, which can serve as a framework for the implementation. Basic updates are performed (and even committed) on an underlying database system simultaneously with the logical evaluation of the ULTRA transactions. Recovery is done by issuing compensating operations in case of failure. Although the architecture appears to be rather different from the logical concepts at a first glance, we actually stay close to the logical semantics and can exploit a significant amount of the evaluation flexibility provided by the *LOLA* system [ZF97]. In particular, we are not tied to a pure top-down evaluation with backtracking.

1 Introduction

In [WF97, WFF98a, WFF98b] we present the rule-based update language ULTRA. Updates are implemented by goals, and rules can be used to define complex operations to be reused in other goals. The concept is similar to procedures/functions in classical programming languages. Thus, it is possible to build arbitrarily complex database operations in a modular fashion. Constructs for concurrent composition, sequential composition, and bulk updates are provided, which subsume the classical database operations. The semantics of an update goal w.r.t. a fixed database state is defined in terms of update request sets which contain insertion and deletion requests for ground EDB tuples. This form of deferred updates, which describe transitions between database states, was already used in the languages Heraclitus [GHJ96] and U-Datalog [MBM97] and has also been investigated by Chen [Che95, Che97]. The deferred update approach has some advantages, as it provides a modular semantics for the concurrent composition of transactions as well as for bulk updates built from single updates.

In [WFF98a, WFF98b] we show that an optimistic transaction processing method can be used to guarantee isolation of any two independent transactions, which are invoked by two top-level goals. The execution of a transaction is divided into two separate phases. In an *evaluation phase*, all possible transitions for the top-level goal are calculated without changing the physical database state. In a subsequent *materialization phase*, one of the transitions is actually performed. This kind of two-phase transaction processing corresponds exactly to the formal semantics of the ULTRA language. However, as we will see in Section 3, an operational semantics completely based on deferred updates has several drawbacks.

In this paper we develop an operational model that implements the logical semantics in terms of nested transactions performed on top of a loosely coupled database system (DBS). The transactions can execute arbitrary basic operations, although we restrict ourselves to the insertion/deletion model of ULTRA in this paper. We sketch how the logical semantics based on deferred updates can be implemented using *(recoverable) immediate updates*. In other words, update operations, which are semantically deferred, are executed as soon as possible. This enables us to interleave logical reasoning and materialization of the physical changes – it is no longer necessary to separate the evaluation from the materialization phase. A complex update goal is not completely resolved according to the logical semantics. Instead, only parts of the goal are evaluated, and then some selected updates are applied to the database. This yields a new physical database state and a continuation goal, i.e. a new update goal, that is again evaluated w.r.t. the intermediate state, i.e. the new current state. As the computed updates leading to the intermediate state have already been applied to the database, there is no need for hypothetical reasoning, when the evaluation of the continuation goal takes place. The interleaving of evaluation and materialization is depicted in Figure 1, where DB_0 denotes the initial state and $DB_1, DB_2, \ldots$ denote intermediate states.

To enable transactional behaviour in the new architecture, we propose a nested transaction model and sketch the requirements of the scheduler. The evaluation of a subquery

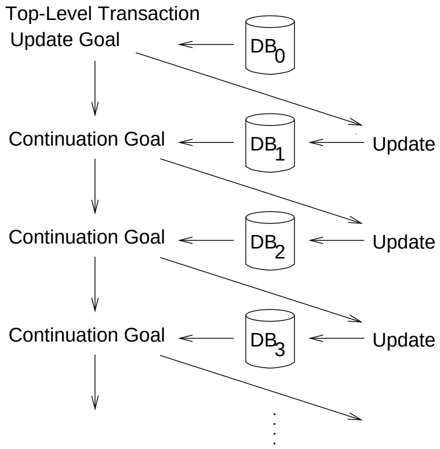

Figure 1: Interleaving of evaluation and materialization in the new ULTRA system

below a choice point is run as a *subtransaction*. So, if one of the branches selected at a choice point fails, the changes of that branch can be undone by aborting the corresponding subtransaction. Choice points result from the evaluation of non-deterministic update rules just in analogy to conventional logic programming languages. Updates are committed immediately on an underlying database system, and recovery of aborted subtransactions is done by compensation [KLS90]. This way, the system can be easily extended to arbitrary basic operations and becomes rather independent from transaction support provided by the external database system.

Based on the ideas and requirements mentioned above we have designed an architecture consisting of loosely coupled components. This makes up one more step towards a seamless integration of updates specified in terms of logic and transactions on database systems.

The rest of the paper is organized as follows: In Section 2 we recall syntax and semantics of a fragment of the ULTRA language and present a short running example. We discuss some problems arising and develop a new ULTRA architecture in Section 3. Sections 4 and 5 show how transactions are executed in the new architecture and what properties are required from the scheduler. Section 6 summarizes the paper and collects research issues to be addressed in the future.

The work described in this paper has been funded by the German Research Agency (DFG) under contract number Fr 1021/3-1.

2 Syntax and Semantics of ULTRA

In this section we recall the essentials of the ULTRA approach [WF97, WFF98a, WFF98b]. In this paper we restrict ourselves to sequential goals, for which we sketch the operational semantics; an extension to the full language is one of our future research topics (cf. Section 6). A reader who is familiar with ULTRA may skip most parts of this section, but may want to have a look at the running example.

We consider an extended logic database which consists of three components: an *extensional database (EDB)* comprising some base relations, e.g. a relational database, an *intensional database (IDB)* defined by a set of normal deductive rules, and an *update program* consisting of update rules to be defined below. Correspondingly, we distinguish *EDB predicates/atoms*, *IDB predicates/atoms*, and *definable update predicates/atoms*.

The deductive rules (*IDB rules*) and their semantics are not described in detail here. We refer the reader to any textbook on DATALOG with function symbols and negation, e.g. [Llo87].

Update rules are formed like IDB rules, but they have an update goal (see Definition 2.1) in the body. In contrast to IDB rules, which define views on the EDB, the update rules define complex update operations on the EDB. Note that we do not allow view updates, i.e. update operations on IDB relations. Instead, the effect of view updates can be obtained by suitably defined update operations that operate on the EDB (see [MW88b] for instance). In a full-blown system, these update operations may be defined automatically by an abduction-based system component.

Definition 2.1 [Update Goals] The set of *update goals* (in this fragment of ULTRA) is defined by the following cases:

1. Every EDB/IDB atom is an *update goal*.
2. The *basic update atoms* $INS\ r(\vec{t})$ and $DEL\ r(\vec{t})$, where $r(\vec{t})$ is an EDB atom, are *update goals* ("insert", "delete").
3. Every definable update atom $p(\vec{t})$ is an *update goal*.
4. Let φ and ψ be update goals. The *sequential conjunction* $\varphi : \psi$ is an *update goal*.

□

Update goals specify retrieval (EDB/IDB atoms), simple modifications on the EDB (INS, DEL), and calls to user defined complex operations (definable update atoms). Update goals can be composed by the sequential conjunction. Note that the two subgoals of a sequential conjunction refer to different database states (see below for details).

An ULTRA *database* is a triple (DB, P_{IDB}, P_{UP}) with an EDB DB, an IDB P_{IDB} and a set of update rules P_{UP}.

Example 2.2 [Personal Calendar] We use a very simple personal calendar as a running example. In our model, appointments have a descriptive text and occupy one time slot with the length of one hour. Thus, we only need one single EDB relation $entry(Day, Slot, Text)$ which associates a time slot $Slot$ on day Day with an appointment $Text$. The reserved string "-" represents free slots. To keep our example short we consider only six time slots on Monday.

In the rest of the paper we always refer to the initial database state given by the EDB instance DB_0 shown in Table 1. This EDB instance can be interpreted as follows: The

$entry$	Day	$Slot$	$Text$
	mon	10	Briefing
	mon	11	Mrs. Grant
	mon	12	Discussion
	mon	13	-
	mon	14	-
	mon	15	-

Table 1: Initial EDB instance DB_0

owner of the calendar has to attend a morning briefing on Monday from 10am to 11am, she has an appointment with Mrs. Grant at 11am and a discussion from noon to 1pm. The three time slots on Monday ranging from 1pm to 4pm are not reserved, yet.

Next, two IDB predicates are defined: a predicate $free(Day, Slot)$ representing the free slots on a day and a predicate ic representing an integrity constraint. For the sake of simplicity, the constraint here states only that there has to be at least one free slot for lunch on Monday. However, more complex constraints that are definable in DATALOG with function symbols and negation could be used as well. The IDB is defined by the following rules:

$$
\begin{array}{lcl}
free(D, S) & \leftarrow & entry(D, S, \text{“-”}) \\
ic & \leftarrow & free(mon, 12) \\
ic & \leftarrow & free(mon, 13)
\end{array}
$$

The update program contains an update rule

$$
\begin{array}{lcl}
do\_insert(D, T) & \leftarrow & free(D, S) \ : \\
& & DEL\ entry(D, S, \text{“-”}) \ : \ INS\ entry(D, S, T) \ : \ ic
\end{array}
$$

which specifies the insertion of a new appointment with descriptive text T on day D. The system is to find a free slot and record the appointment. The integrity constraint must hold finally. Another rule

$$
\begin{array}{lcl}
do\_delete(D, S) & \leftarrow & entry(D, S, T) \ : \\
& & DEL\ entry(D, S, T) \ : \ INS\ entry(D, S, \text{“-”})
\end{array}
$$

specifies the simple deletion of an appointment from a certain slot S on day D. (Here we take the reasonable assumption that the integrity constraint does not need to be checked for this operation.) □

Let us recall the semantics for update goals and update programs. For every given EDB instance, the interpretation of the IDB atoms is determined by a standard semantics of logic programs, e.g. the well-founded semantics [vGRS91]. In the following, we write $DB \models A$, iff an atom A is true w.r.t. a given EDB instance DB.

Let the set of all EDB instances be given. Transitions from one EDB instance to another can be represented by consistent update request sets.

Definition 2.3 [Update Request Set] An *update request set* Δ consists of *ground basic update requests* of the form $+r(\vec{t})$ or $-r(\vec{t})$, where $r(\vec{t})$ is a ground EDB atom, and *read tags* of the form $?q(\vec{t})$, where $q(\vec{t})$ is an EDB/IDB atom.

An update request set Δ is *consistent*, iff there exists no ground EDB atom $r(\vec{t})$ such that $+r(\vec{t}) \in \Delta$ and $-r(\vec{t}) \in \Delta$.

In the following, $\mathcal{D}$ denotes the set of all consistent update request sets. □

Intuitively, a ground basic update request specifies the insertion (+) or deletion (−) of an EDB atom, i.e. a tuple in an EDB relation. The read tags are needed for the logging of read operations. Note, however, that read access is an immediate operation as opposed to update operations which are (semantically) deferred. As known from optimistic transaction protocols [BHG87], the read tags must be certified to detect read/write conflicts with concurrent transactions. This certification is also (semantically) deferred.

For consistent update request sets $\Delta \in \mathcal{D}$ we can define an execution operator $\oplus_E$ such that $DB \oplus_E \Delta$ denotes the new EDB instance DB' which results from applying to DB the updates collected in Δ.

Furthermore, we can define the sequential composition of two update request sets.

Definition 2.4 [Sequential Composition] The *sequential composition* $\oplus$ of two update request sets $\Delta_1, \Delta_2 \in \mathcal{D}$ is defined by:

$$\begin{array}{rcll} \Delta_1 \oplus \Delta_2 & := & \{+r(\vec{t}) & \mid \; (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2) \vee +r(\vec{t}) \in \Delta_2\} \\ & \cup & \{-r(\vec{t}) & \mid \; (-r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \notin \Delta_2) \vee -r(\vec{t}) \in \Delta_2\} \\ & \cup & \{?q(\vec{t}) & \mid \; ?q(\vec{t}) \in \Delta_1 \;\vee\; ?q(\vec{t}) \in \Delta_2\} \end{array}$$

□

The sequential composition is associative; it coincides with the set union, if the set union is consistent (see [WFF98b] for details). The following property holds for arbitrary consistent update request sets $\Delta_1, \Delta_2 \in \mathcal{D}$ and an arbitrary EDB instance DB:

$$(DB \oplus_E \Delta_1) \oplus_E \Delta_2 = DB \oplus_E (\Delta_1 \oplus \Delta_2)$$

As depicted in Figure 2, its natural meaning is that the transition from the state DB to the state DB' via Δ_1 and on to the state DB'' via Δ_2 is equivalent to a direct transition from DB to DB'' represented by $\Delta_1 \oplus \Delta_2$.

We will now define the interpretation of update goals. We keep an initial EDB instance DB_0 fixed and describe the dynamic changes on the database exclusively by update request sets. An interpretation I assigns a set of pairs of consistent update request

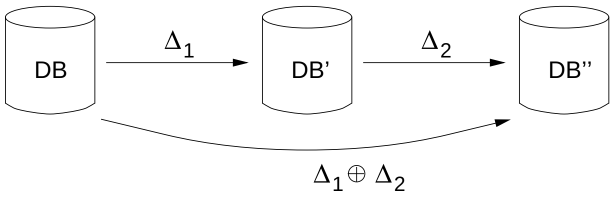

Figure 2: Sequential composition of update request sets

sets, i.e. a subset of $\mathcal{D} \times \mathcal{D}$, to each ground update goal φ. The first component of each pair $(\Delta_C, \Delta) \in I(\varphi)$ refers to a (hypothetical) current state DB_{Curr} reachable from DB_0 by executing Δ_C. In this state, φ is to be evaluated. The second component refers to an update request set that would lead to the next intermediate database state DB_{Next} if applied to the current state DB_{Curr}. Intuitively spoken, Δ collects the updates produced by the goal φ. Every pair (Δ_C, Δ) corresponds to an allowed state transition from DB_{Curr} to DB_{Next}

$$(\quad DB_0 \ \stackrel{\Delta_C}{\longrightarrow} \ ) \quad DB_{Curr} \ \stackrel{\Delta}{\longrightarrow} \ DB_{Next}$$

If $I(\varphi)$ contains multiple pairs with the same first component Δ_C, φ has a non-deterministic update interpretation. Note that non-deterministic *choice* is *not* involved at the level of the logical semantics. However, an implementation has to perform a choice operation when materializing one of the new possible database states.

Definition 2.5 [Interpretation of Update Goals] Let an interpretation I_{UP} of the ground definable update atoms be given. Then we define the interpretation I of the ground update goals inductively as follows:

(1) EDB/IDB atom

Let $q(\vec{t})$ be an EDB or an IDB atom.

For all $\Delta_C, \Delta \in \mathcal{D}$ define:

$$(\Delta_C, \Delta) \in I(q(\vec{t})) \ :\Longleftrightarrow$$
$$(DB_0 \oplus_E \Delta_C) \models q(\vec{t}) \text{ and } \Delta = \{?q(\vec{t})\}$$

(2) Basic update atom

For all $\Delta_C, \Delta \in \mathcal{D}$ define:

$$(\Delta_C, \Delta) \in I(INS \ \ r(\vec{t})) \ :\Longleftrightarrow \ \Delta = \{+r(\vec{t})\}$$
$$(\Delta_C, \Delta) \in I(DEL \ r(\vec{t})) \ :\Longleftrightarrow \ \Delta = \{-r(\vec{t})\}$$

(3) Definable update atom

Let $p(\vec{t})$ be a definable update atom.

For all $\Delta_C, \Delta \in \mathcal{D}$ define:

$$(\Delta_C, \Delta) \in I(p(\vec{t})) \ :\Longleftrightarrow \ (\Delta_C, \Delta) \in I_{UP}(p(\vec{t}))$$

(4) Sequential conjunction

For all $\Delta_C, \Delta \in \mathcal{D}$ define:

$(\Delta_C, \Delta) \in I(\varphi : \psi) \;:\Longleftrightarrow$

there exist $\Delta_1, \Delta_2 \in \mathcal{D}$ such that:

$(\Delta_C, \Delta_1) \in I(\varphi)$ and $(\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi)$ and $\Delta = \Delta_1 \oplus \Delta_2$

□

Definition 2.5 essentially formalizes what kind of update request sets Δ are necessary to satisfy the corresponding update goal. In operational terms this amounts to the definition of the update request sets Δ that are generated during the evaluation. Note how basic update atoms are related to basic update requests and how the sequential conjunction is related to the sequential composition of update request sets. For each relevant EDB/IDB atom, a (fine-grained) read tag is collected. It is worth mentioning that the sequential conjunction is associative (see [WFF98b]), thus we can omit precedence brackets in the examples.

Note that we define the semantics of update goals based on ground atoms. To enable a sensible evaluation, the basic update atoms and the definable update atoms must become bound by EDB/IDB atoms. This requires some safety properties of the update program and a corresponding sideways information passing strategy used by the evaluation engine. However, we will not consider such topic in this paper. Here it suffices to assume that the sequential goals can be evaluated from left to right, i.e. that no sideways information passing is necessary from a future state into a previous state.

Our next step is to find a sensible interpretation of the definable update atoms. We call an interpretation I_{UP} a *model* of the given update program P_{UP}, iff $I(\varphi) \subseteq I(p(\vec{t}))$ holds for every ground instance $p(\vec{t}) \leftarrow \varphi$ of an update rule $r \in P_{UP}$, where I is the extension of I_{UP} according to Definition 2.5. Every update program normally has a variety of models and we intend to find a minimal model, which allows only the least possible state transitions. Informally, this means that the ULTRA semantics should not imply changes that are not specified by the program. This is in contrast to other notions of minimality, e.g. the minimality required for an implicit revision of facts [MT98].

In [WFF98a, WFF98b] we prove the following theorem.

Theorem 2.6 [Model Existence Theorem] Every update program P_{UP} has a unique minimal model M_{UP}. □

In the ULTRA system, a transaction is invoked by a ground top-level update goal, which we call *transaction goal*. When a transaction goal is issued by the user, evaluation has to be performed as a *(top-level) transaction* to guarantee the ACID properties [BHG87]. In particular, it must be ensured that the data accessed by a transaction is not affected simultaneously by other transactions *(isolation)*, and that either all changes caused by the transaction are applied to the database, or none of them at all *(atomicity)*. To achieve this, the execution of all basic update requests as well as read access to the database has to be certified by the system. If the execution of an operation could possibly compromise the ACID properties, the operation cannot be certified – so the system may decide to delay it, to abort the whole transaction, or to do something else.

Example 2.7 [Personal Calendar (Cont.)] Suppose that in our calendar example two concurrent transactions have been invoked using the *do_insert* update goal. Assume further that both try to insert appointments on the same day. Unconstrained execution of the two transactions could result in a situation where both appointments are assigned to the same time, if both transactions look up the free time slots first, before any of them performs its updates. This leads to an inconsistent database state, which is also reflected by a violation of the ACID properties. The system must detect this conflict and refuse to certify the operations of at least one of the transactions. □

The logical semantics of a transaction goal is based on the minimal model of the update program.

Definition 2.8 Let φ be an update goal that does not contain any free variables. Let M_{UP} denote the unique minimal model of an update program P_{UP} (w.r.t. DB_0 and P_{IDB}) and M the extension of M_{UP} to arbitrary update goals.

A *possible transition* for the transaction goal φ in an ULTRA database (DB_0, P_{IDB}, P_{UP}) is every update request set $\Delta \in \mathcal{D}$ such that

$$(\emptyset, \Delta) \in M(\varphi).$$

□

Before a transaction invoked by a transaction goal can be committed, one of the possible transitions must have been materialized, i.e. its update requests must have been executed and committed on the persistent EDB. In the presence of independent concurrent transactions, the ULTRA system must control not only the write operations but also the read access of the transactions. The read tags contain sufficient information about data relevant to each transaction. They must be certified to guarantee the ACID properties (see [WFF98a, WFF98b] for details). If there exist multiple possible transitions $\Delta^{(1)}, \ldots, \Delta^{(n)}$ for a transaction goal φ, the transaction goal will be called *non-deterministic*, see Figure 3. The actual choice of one transition, however, is not reflected by the logical semantics. In the architecture described in [WFF98a], a choice was performed after the evaluation of the goal φ, whereas in this paper we propose a "depth-first" evaluation strategy which permanently tries to find one successful branch. Note that this corresponds to the well-known PROLOG strategy [MW88a]. Similar freedom as for the choice can be found for the materialization of the transitions. The logical semantics does not state anything about materialization time, although it deals with multiple database states.

Example 2.9 [Personal Calendar (Cont.)] Recall Example 2.2 and consider the two transaction goals $\varphi_1 := do\_insert(mon, \text{"Review"})$ and $\varphi_2 := do\_delete(mon, 12)$.

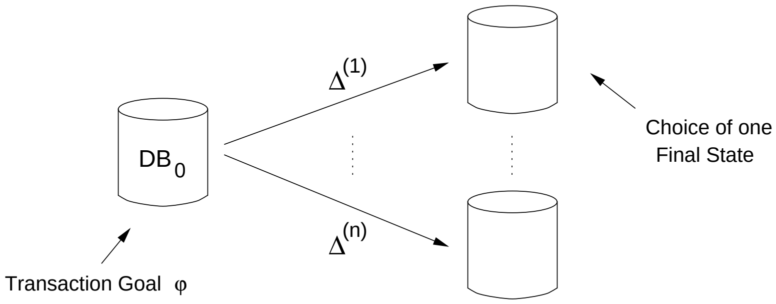

Figure 3: Possible transitions for a (non-deterministic) transaction goal

The possible transitions for φ_1 are the update request sets:

$$\begin{aligned}\Delta_1^{(1)} &= \{?free(mon, 14),\ ?ic,\ \ -entry(mon, 14, \text{“-”}),\\ &\quad +entry(mon, 14, \text{“Review”})\}\\ \Delta_1^{(2)} &= \{?free(mon, 15),\ ?ic,\ \ -entry(mon, 15, \text{“-”}),\\ &\quad +entry(mon, 15, \text{“Review”})\}\end{aligned}$$

They correspond to the possibilities of inserting the “review” appointment into either the 2pm or the 3pm slot. The 1pm slot cannot be occupied, as this would cause ic to fail. As we can see, do_insert is a non-deterministic update operation.

φ_2 has only one possible transition:

$$\begin{aligned}\Delta_2 &= \{?entry(mon, 12, \text{“Discussion”}),\\ &\quad -entry(mon, 12, \text{“Discussion”}),\ \ +entry(mon, 12, \text{“-”})\}\end{aligned}$$

Let us clarify why Δ_2 is indeed a possible transition for φ_2. Recall the semantics given in Definition 2.5.

The following holds regardless of the interpretation I_{UP} of definable update atoms and regardless of a given update request set $\Delta_C \in \mathcal{D}$:

$$\begin{aligned}(\emptyset, \{?entry(mon, 12, \text{“Discussion”})\}) &\in I(entry(mon, 12, \text{“Discussion”}))\\ (\Delta_C, \{-entry(mon, 12, \text{“Discussion”})\}) &\\ &\in I(DEL\ entry(mon, 12, \text{“Discussion”}))\\ (\Delta_C, \{+entry(mon, 12, \text{“-”})\}) &\in I(INS\ entry(mon, 12, \text{“-”}))\end{aligned}$$

The first assertion refers to the initial state DB_0, since the truth value of the atom $entry(mon, 12, \text{“Discussion”})$ is essential, whereas the other assertions hold w.r.t. arbitrary intermediate states represented by Δ_C. We can apply case (4) of Definition 2.5 twice and get:

$$\begin{aligned}(\emptyset, \Delta_2) \in I(&entry(mon, 12, \text{“Discussion”}) :\\ &DEL\ entry(mon, 12, \text{“Discussion”}) : INS\ entry(mon, 12, \text{“-”}))\end{aligned}$$

Note that

$$\begin{aligned}\Delta_2 &= \{?entry(mon, 12, \text{“Discussion”})\}\\ &\oplus \{-entry(mon, 12, \text{“Discussion”})\}\\ &\oplus \{+entry(mon, 12, \text{“-”})\}\end{aligned}$$

holds.

In [WFF98a, WFF98b] we have shown that the minimal model M_{UP} of the update program can be computed by a bottom-up fixpoint iteration. This is similar to the computation of the least Herbrand model of a definite logic program [Llo87]. Applying the rule which defines the update predicate *do_delete* we can derive

$$(\emptyset, \Delta_2) \in M_{UP}(do\_delete(mon, 12))$$

which completes the semi-formal proof that Δ_2 is a possible transition for φ_2.

The derivation of the possible updates for φ_1 can be done in a similar fashion, but is omitted in this paper.

Note the interesting fact, that although all slots may be explored during an evaluation of φ_1 (in order to find a free slot), from a semantical point of view only the read access $free$ w.r.t. the modified slot is essential and must be certified. For instance, if $\Delta_1^{(1)}$ is chosen for materialization, another transaction may concurrently update the 10am, 11am, or 3pm slot. In contrast, a modification of the 2pm slot may affect the truth value of $free(mon, 14)$, such that in the modified state the solution $\Delta_1^{(1)}$ may be invalid. Thus the 2pm slot must be kept unchanged. The other slots are related with *ic* and must be kept unchanged as well. Technically, the certification can be done by the scheduler (see Section 4), which knows from a predefined compatibility relation that a read tag $?free(mon, 14)$ is in conflict with a deletion $-entry(mon, 14, \text{“-”})$, etc. When for instance the read tag $?free(mon, 14)$ is approved, a deletion request $-entry(mon, 14, \text{“-”})$ belonging to a concurrent transaction will be rejected and vice versa. □

3 The ULTRA Architecture

Recall that in the ULTRA concept, transactions are invoked by *(top-level) transaction goals* φ that are resolved against a given *update program* P_{UP}. The execution of a transaction consists of two types of processing: the *evaluation* of an EDB/IDB query (for binding variables and computing the update requests/read tags) and the *application* of selected update requests to the EDB. In the architecture described in [WFF98a], the execution was strictly divided into two subsequent phases doing exclusively evaluation and application, respectively. Thus, first all possible transitions of the transaction goal had to be computed without changing the physical EDB instance, and second one possible transition could be materialized. This execution method had been derived from the formal construction of the model-theoretic semantics. However, an operational semantics completely based on deferred updates has several drawbacks.

First, it raises the need for hypothetical reasoning when referring to intermediate states: as the operations leading to an intermediate state are known but not carried out yet, the effects are not visible and thus must be computed by a reasoning component. For this purpose, a sound and complete axiomatization has to be provided for the basic operations. Although the hypothetical reasoning turns out to be easy for the pure insertion/deletion model, it becomes complicated or even impossible, when more complex basic operations, e.g. calls to stored SQL procedures, are allowed. Furthermore, the reasoning process may dynamically become more expensive with every subsequent state, since it always starts at the (physical) initial state. Axiomatization of effects of actions and (hypothetical) reasoning about these actions was investigated in more detail by Reiter [Rei95] (situation calculus) and Kowalski [Kow92] (event calculus).

A second practical problem results from performing a transaction in two strictly separated phases (evaluation and materialization). Such a system does not show a continuous behaviour and thus is not suitable to be extended by interactive components. For example, let a multimedia session be modeled using an extended ULTRA language which provides some additional basic operations for user interaction, e.g. a display and a wait action. When the user now initiates a new session by submitting the "start" goal, it is obviously not sensible that each display action is deferred to the end of the session. Instead, in such interactive applications, it is more natural that logical reasoning and physical changes (actions) are interleaved. Semantically, the reaction on an external event can be considered as the evaluation of an intermediate state which cannot be handled hypothetically.

Another problem results from the bottom-up evaluation approach used in the current prototype, although it is combined with top-down techniques as described in [ZF97]. As the evaluation phase computes all possible updates, while only one will finally be selected for materialization, a lot of unnecessary work may be done. Especially in the presence of non-deterministic specifications with much hypothetical reasoning involved, even small examples may not be tractable anymore. The problem can be handled using a lazy evaluation strategy or backtracking, such that only a few branches are explored. Then, the evaluation can stop as soon as a successful branch has been detected.

Note that the proper handling of non-determinism, e.g. by a backtracking strategy, is necessary and not just an option for the modified operational semantics described below, which performs the updates immediately: Whenever the system reaches a choice point [MW88a], i.e. a point where there are several possible ways to continue the evaluation, one possible update branch must be selected for further exploration and the updates on that branch must be performed on the EDB. In case of logical failures – or even failures resulting from system errors or conflicts with other transactions – the branch must be rolled back to the last choice point. This requires update compensation (a general form of recovery) [KLS90] and an underlying transaction model which enables at least multiple checkpoints [BHG87, GR93]. To be open to extend the system to intra-transaction-parallelism, we propose a nested transaction model [BBG89, Mos85]. Note that a committed-choice strategy [Sha87] would not be adequate.

These requirements for a revised evaluation technique – no hypothetical reasoning, interleaving of logical evaluation and physical execution of basic update operations, backtracking to reduce unnecessary work, and a flexible transaction model – lead us to an improved ULTRA architecture. The components and their interaction are depicted in Figure 4.

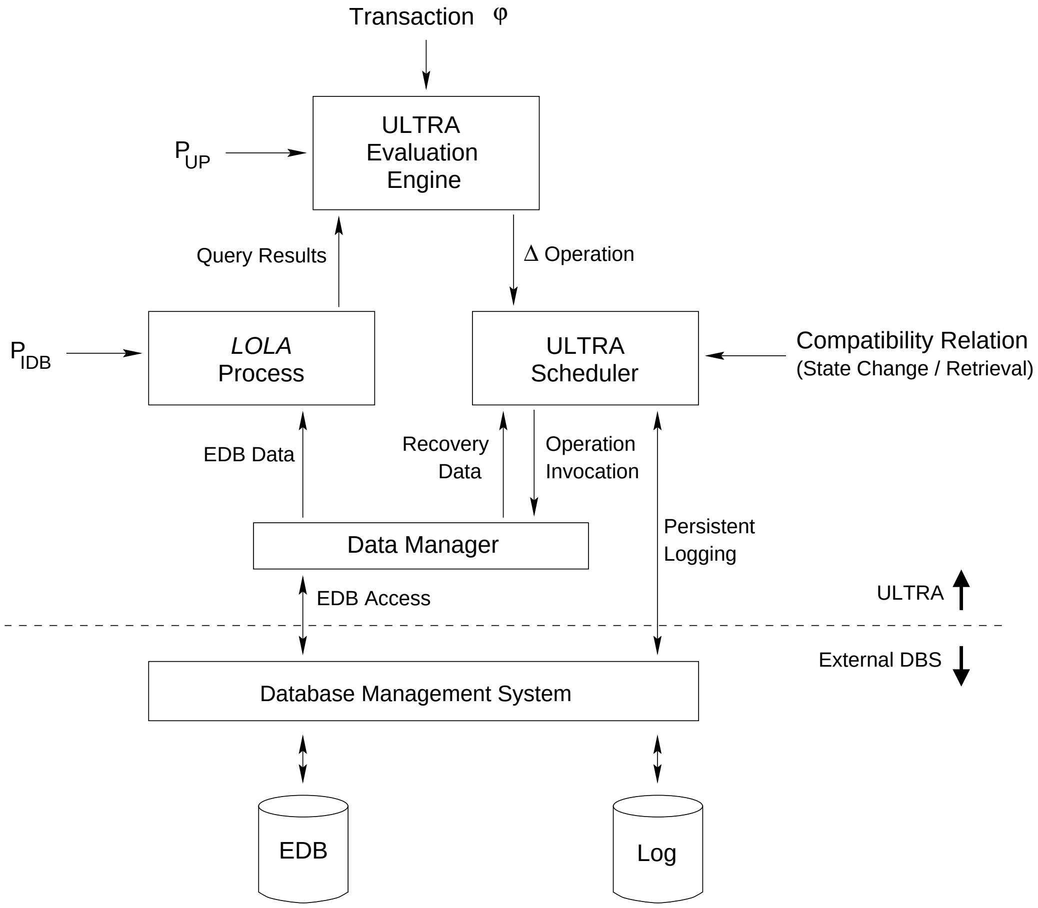

Figure 4: Architecture of the ULTRA system

The new architecture is based on five components: The ULTRA *evaluation engine* accepts top-level transaction goals φ and evaluates them w.r.t. a given update program P_{UP}. EDB and IDB goals are evaluated by a *LOLA process* based on the IDB program P_{IDB}. The basic update operations of a query are sent to the ULTRA *scheduler* which orders the operations from various transactions in such a way that they appear to run in isolation. The scheduler forwards operations to the *data manager* for execution on the underlying external *database system*, which persistently stores the EDB and a copy of the scheduler's log. The external database management system only has to enable flat transactions on the complete storage, i.e. EDB and log. Local transactions, i.e. transactions that are not invoked by the scheduler, are not allowed, otherwise multidatabase techniques [BGMS92] would be required to ensure global consistency. In this paper we take it for granted that messages are neither lost nor corrupted. The communication protocol, however, lies outside the scope of this paper.

Besides the fact that this architecture satisfies the given requirements, it features some advantages due to the loose coupling of the components. For example, the *LOLA* process could be replaced by another logic database system, such as XSB [SSW94]. We are allowed to use different evaluation techniques and strategies in the components, e.g. a PROLOG-like top-down strategy for the update goals, and bottom-up handling of IDB rules. The external database system can be chosen freely as well. Even the scheduler can select from various protocols suitable for nested transactions, e.g. [AFL$^+$88, BBG89, FLMW90, LMWF94]. In particular, the scheduler does not depend on a sophisticated and efficient transaction support from the external database system.

4 The ULTRA Scheduler

As the ULTRA scheduler is one of the most prominent features of our new architecture, let us have a closer look at it. A description of the new evaluation engine follows in Section 5.

The scheduler receives basic update requests and read tags from the evaluation engine, together with additional transaction control commands. Transaction control commands can state the begin, commit, or abort of a (sub-) transaction. In the current version of ULTRA, the update requests correspond to insertions ("+") and deletions ("−") of tuples. Read tags ("?") must be certified to exclude that the *LOLA* process has read invalid data. Yet, the scheduler can be provided with information about arbitrary (semantically rich) operations and thus allows an easy extension of the system to a more general, not just insertion/deletion-oriented ULTRA language.

The ULTRA system implements a *nested transaction* system [BBG89, Mos85] on top of a conventional external DBS. When the user issues a top-level transaction, it is decomposed into a number of subtransactions by the ULTRA evaluation engine (cf. Section 5), each of which may in turn consist of other subtransactions. The leaf subtransactions of a transaction tree contain basic operations to be applied to the database (see Figure 5). These operations are sent to the external DBS as separate, independent transactions and are committed immediately. Consequently, the transaction hierarchy is partly *open* nested [WS92]. The main advantage of the open hierarchy is the fact that the transactions built from basic operations are fully controlled by the ULTRA scheduler and not affected by the external transaction manager, which has to deal only with rather short transactions. The price of the flexibility is a bigger overhead for the recovery.

It should be mentioned that the external database may become temporarily inconsistent due to the early commit of external actions in the open transaction hierarchy. However, we assume that each compensating operation will eventually succeed (maybe after some retries), such that the consistency will be eventually restored. Further, we disallow transactions on the external database system which are not executed under control of the ULTRA scheduler.

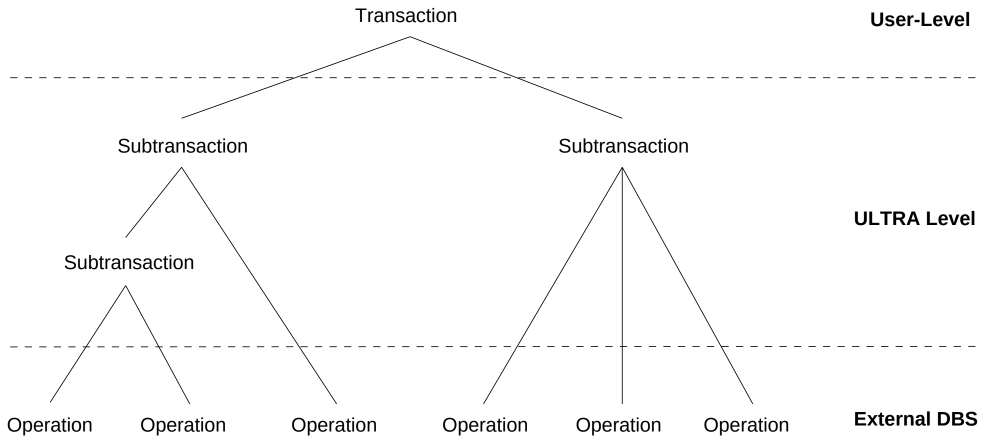

Figure 5: Nested transactions in the ULTRA system

The task of the scheduler is to interleave the operations stemming from various independent transactions in such a way that the resulting schedule is equivalent to a serial, non-interleaved execution of the transactions *(serializability)* [BHG87]. Besides this, it has to ensure that the changes performed by committed transactions are actually reflected by the database, while all effects from aborted transactions are removed from the database *(recoverability)* [BHG87]. Even if a system error occurs, these properties must be guaranteed.

Let us first have a look at the recoverability [BHG87] requirements. As the architecture described here relies on immediate updates that are committed by the external DBS, we do not have to take care of redoing operations after a system crash as long as we can guarantee that pending subtransactions are submitted safely to the external DBS, but must concentrate on the undoing of aborted transactions – aborted either from the outside due to logical reasons (backtracking), or by the scheduler itself due to conflicts, system errors, etc. As all basic operations are executed on the DBS as committed transactions, we use *compensation* [KLS90] to undo these operations. This means that for each basic operation o, a compensating action o^{-1} which removes all observable effects of o from the database has to be executed. For example, the compensating action of an insertion is a deletion and vice versa. However, compensation is not that simple at all. Usually, additional information about the initial state is needed to compensate an operation. In the ULTRA approach, insertions and deletions behave as assertions about the next state. The system, however, has to record whether a proper insertion or deletion takes place. If compensation is needed, the compensating action o^{-1} must be parameterized with the additional recovery information. Then it can undo the effect of the forward operation o (see Figure 6).

To undo the effects of a subtransaction, the corresponding compensating actions must be scheduled backwards on the DBS, up to the beginning of the subtransaction. For this purpose a *log* is kept that contains all basic operations together with recovery information, e.g. whether an insertion really inserted a new tuple or it was already contained in the EDB before the update. Note that the external DBS can be used to implement the *persistent copy* of the log, which is necessary to ensure recoverability in case of system errors. To guarantee that all logged operations are also successfully executed and vice versa, execution and logging either can be merged into a single transaction on the DBS, or a protocol like two phase commit [BHG87] has to be used. The persistent log must contain at least all (committed) basic operations and their specific recovery information, however, it may contain some additional information to enable auditing, debugging, etc.

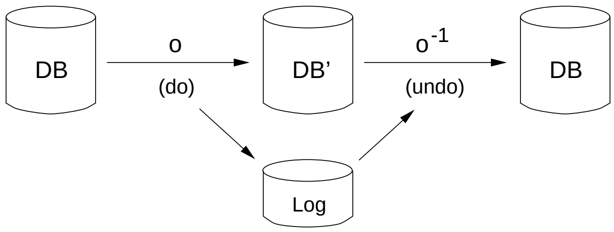

Figure 6: Operation and compensation

The recovery method presented so far may look a bit trivial at a first glance. This is mainly due to the simplicity of the basic operations used in the ULTRA approach described here. If semantically rich basic operations, e.g. calls to stored procedures of the underlying DBS or actions involving interaction with the "real world", are performed by the scheduler, this form of recovery turns out to be the only viable solution. In this case, for each basic operation o a corresponding compensating operation o^{-1} must be provided. E.g. the compensation of a "print document" action could be to advise a human operator to put the print-out into the shredder or to print a remark on the next document asking the recipient to ignore the former print-out.

The second task of the scheduler is to guarantee serializability [BHG87] when several independent transactions are executed concurrently. To do so, it must check compatibility between basic operations, compensating operations and read access. As read access is done independently by the $LOLA$ process (see also Section 5), the scheduler is informed about the relevant read access by the incoming read tags which are generated by the evaluation engine (cf. Definition 2.5, case (1)). Because the read tags may refer not only to EDB, but also to IDB predicates, the scheduler needs to know the IDB program P_{IDB} to check for conflicts with the basic update operations. The determination of conflicts is a hard problem indeed (see [Elk90] for instance) and lies outside the scope of this paper. So, let us suppose that the scheduler knows a *compatibility matrix* over the set of all (ground) basic update requests and all read tags. Two basic update requests are called *write-compatible*, if the execution order has no effect on the resulting state; a read tag is called *compatible* with an update request, if the execution of the update request cannot change the information in the read tag, such that read access and update can be exchanged. For instance, like in [WFF98a], read/write conflicts could be easily defined at the predicate level using the dependency graph of P_{IDB}. The compatibility matrix approach can be used for arbitrary basic operations, not just for insertions and deletions.

When an operation is submitted to the scheduler, it checks whether or not it is admissible, or if it is to be delayed or rejected. If the operation is admissible, it is passed to the data manager, that executes the corresponding transaction on the external DBS and returns the information needed for recovery, which is in turn written into the persistent log together with given information about the operation. If the scheduler detects a conflict, the operation is either delayed for later execution, or it is rejected. In the latter case, the corresponding subtransaction is aborted. Then it is up to the higher levels, i.e. the ULTRA evaluation engine, to decide whether to retry it, to choose another possible branch, to abort the whole transaction, etc. Note that a scheduler that delays operations raises the danger of deadlock and therefore has to take care of its detection or avoidance. However, the scheduler is not bound to use delays by the overall system architecture, but is free to use any protocol suitable to handle nested transactions, e.g. [AFL$^+$88, BBG89, FLMW90, LMWF94].

5 Evaluation and Immediate Updates

In this section we describe the ULTRA evaluation engine and show how a transaction is executed. As usual, the evaluation engine takes a transaction goal and performs an evaluation w.r.t. the fixed update program P_{UP}. In contrast to our former approach [WFF98a], the evaluation is restricted in the sense that at most information that belongs to the *current* state is computed. When a subformula β of a composite goal refers to a state that would result from applying the updates related to another subformula α, the formula β is kept as a *continuation goal*, until the updates of α have been materialized (see also Figure 7). After the logical reasoning w.r.t. the current state, the evaluation will terminate providing variable bindings, ground basic update requests/read tags, and new goals (the continuations). This is illustrated in Example 5.1. When the logical evaluation stops, the evaluation engine encounters a new choice point like in PROLOG [MW88a]. It creates a new subtransaction and sends one of the computed update requests/read tags to the scheduler. After the scheduler returns a success message, the corresponding continuation goal is evaluated in the new state like a top-level query. If finally the goal is completely resolved, the evaluation engine can send a global commit to the scheduler and stop. If the evaluation fails logically or due to errors detected by the scheduler (e.g. conflicts with other transactions, deadlocks, etc.), the transaction must be rolled back to the last choice point by aborting the corresponding subtransaction. The evaluation engine can choose another branch, retry the old one, or cascade the aborts upwards within the subtransaction hierarchy, possibly up to the top-level.

Read access to EDB/IDB relations is performed by a *LOLA* process [ZF97]. On the other hand, any engine which is able to compute all answers of an EDB/IDB query, for example XSB [SSW94], can be used as well. However, our prototype relies on the *LOLA* system. Whenever an EDB/IDB goal is to be evaluated, the goal is transmitted to the *LOLA* process which computes and returns all corresponding answers. When accessing

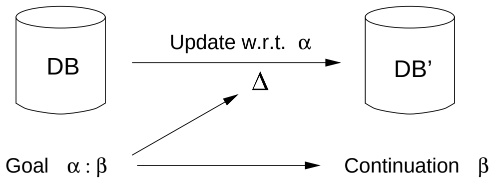

Figure 7: Goals and continuations

the EDB relations, data must be fetched from the underlying database system. This has to be done in a low-level transaction in order to ensure that all data fetched during one evaluation belong to the same database state. Independently from the evaluation in *LOLA*, the scheduler will check later if the read access has taken place in a valid state or if there was a read/write conflict with a concurrent transaction.

Example 5.1 [Personal Calendar (Cont.)] Assume that the top-level transaction goal $\varphi_1 := do\_insert(mon, \text{"Review"})$ of Example 2.9 is fed into the evaluation engine. The goal requires to assign the "review" task to a free time slot on Monday. When the evaluation is started, a new top-level transaction must be opened by sending the corresponding request to the scheduler. The scheduler will create the transaction and ensure that the ACID properties [BHG87] are maintained.

Knowing the update program P_{UP} the evaluation engine can resolve the goal

$$do\_insert(mon, \text{"Review"})$$

by generating the resolvent:

$$free(mon, S) \ : \ DEL\ entry(mon, S, \text{"-"}) \ : \ INS\ entry(mon, S, \text{"Review"}) \ : \ ic$$

As we can see, this is a sequential goal, and according to Definition 2.5, case (4), everything to the right of the first conjunction sign ":" refers to a *intermediate* state which is not materialized, yet. In the architecture described in this paper, consequently, the ULTRA evaluation engine only has to care about the subgoal $free(mon, S)$, which refers to the *current* state. The rest will be kept unresolved, until the intermediate state becomes materialized.

So the evaluation engine must find bindings for the variable S and produce read tags for the retrieval subgoal. This is done by sending the query $free(mon, S)$ to the *LOLA* process. It will be evaluated, and the result tuples $(mon, 13)$, $(mon, 14)$, and $(mon, 15)$ will be returned. The evaluation engine produces the read tags

$$?free(mon, 13),\ ?free(mon, 14),\ ?free(mon, 15)$$

– according to the logical semantics (see Definition 2.5) – and the corresponding continuation goals:

$$DEL\ entry(mon, 13, \text{"-"}) \ : \ INS\ entry(mon, 13, \text{"Review"}) \ : \ ic$$
$$DEL\ entry(mon, 14, \text{"-"}) \ : \ INS\ entry(mon, 14, \text{"Review"}) \ : \ ic$$
$$DEL\ entry(mon, 15, \text{"-"}) \ : \ INS\ entry(mon, 15, \text{"Review"}) \ : \ ic$$

Since there are no further rules defining *do_insert* the "pure" evaluation terminates at this point. It is time to change the current state.

The evaluation engine creates a new subtransaction by sending a request to the scheduler. This subtransaction will serve as a rollback sphere (cf. spheres of control [Dav78]), which is used to enable local rollback without aborting the whole transaction (see also [GR93] for this use of the nested transaction concept). As we have three alternative update branches (see Figure 8), backtracking may be necessary, and corresponding provisions are made by starting the new subtransaction. If a branch fails in the future, the corresponding subtransaction will be aborted and rolled back, which corresponds to backtracking. In contrast to the backtracking in PROLOG [MW88a], which does not undo changes performed by assert/retract, the updates belonging to the failed branch are compensated in our model.

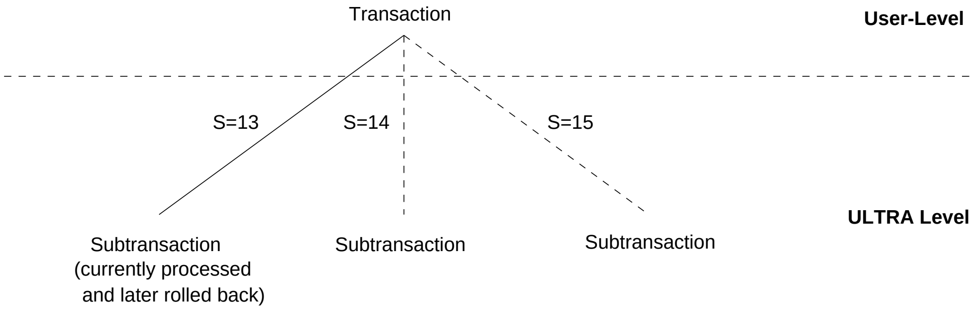

Figure 8: Choice point and subtransactions

Assume that the evaluation engine selects the branch $S = 13$ for further exploration as indicated in Figure 8. First, the read tag $?free(mon, 13)$ must be sent to the scheduler for certification. After some time, the scheduler will have checked that no other transaction has manipulated the EDB in a way that affects the truth value of the atom $free(mon, 13)$. Now the evaluation engine can proceed and evaluate the first continuation goal. The evaluation of the subgoal $DEL\ entry(mon, 13,$ "-"$)$ is trivial. So, an update request

$$-entry(mon, 13, \text{“-”})$$

is generated, while the new continuation goal

$$INS\ entry(mon, 13, \text{“Review”})\ :\ ic$$

is left. As this is deterministic, there is no need to create a choice point and a new subtransaction.

Let the deletion be scheduled and finally performed, let the continuation goal be evaluated – without doing something special –, and let the next insertion be scheduled and successfully performed as well.

Now the evaluation engine reaches the concluding integrity check. It sends the query goal ic to the $LOLA$ process which sees the deletion and the insertion made on the

EDB and thus recognizes that ic has become false. This causes a logical failure in the evaluation engine, as no result tuple is returned.

Due to this failure, the ULTRA evaluation engine decides that the updates made since the last choice point must be rolled back. This is done by simply sending an abort message (w.r.t. the currently active subtransaction) to the scheduler. To undo the subtransaction, the scheduler consults the log and executes compensating operations on the EDB: it deletes the newly inserted tuple (mon, 13, "Review") from the $entry$ relation and then inserts the tuple (mon, 13, "-") which has been deleted before. Finally, it reports the successful compensation to the evaluation engine, which initiated the abort.

The recovery information states that the insertion/deletion have been proper, thus the compensation is done by a (proper) deletion/insertion. The certification of the read access did not have any effect on the EDB, so there is no need for a compensation action for it here.

The evaluation engine can now choose another branch, e.g. $S = 14$. As for the branch $S = 13$, a new subtransaction is opened to mark the choice point again, and within this transaction the read tag $?free(mon, 14)$ is certified and the update requests $-entry(mon, 14,$ "-") and $+entry(mon, 14,$ "Review") are executed on the EDB.

This time, the integrity constraint ic remains valid, and after the read tag $?ic$ is certified, the evaluation engine encounters an empty continuation goal. The top-level transaction can be committed and the evaluation engine can stop. As all updates have been already executed – and even committed (cf. Section 4) – on the EDB, there is no need for a special materialization phase.

Semantically, the update result corresponds to the possible transition $\Delta_1^{(1)}$ in Example 2.9. Note that the update branch $S = 15$ has not been considered in the evaluation. This form of incomplete evaluation can save much time for non-deterministic specifications, provided that usually one solution is found early. □

The evaluation engine can react individually whenever a failure is encountered. The scheduler performs a rollback to the last choice point, and – usually – a new branch will be tried. However, depending on the type of failure (logical failure, conflict abort, system error abort, communication error, time-out, etc.) the evaluation engine can alternatively retry a branch or cascade the abort to an earlier choice point (or even abort everything). For instance, a single or double retry might be advantageous in presence of a system or communication error. The abort of the whole transaction is sensible, if too many conflict aborts have been reported by the scheduler, such that the transaction will probably fail anyway.

In the example we considered only the logical evaluation of a single transaction. However, the evaluation engine can be made multi-threaded in order to evaluate multiple independent transactions in parallel. To guarantee the isolation property, all ULTRA evaluation threads must access the same scheduler. Also the underlying $LOLA$ process can actually be a rather complex server. It only must be ensured that every evaluation of an EDB/IDB query is performed w.r.t. one single state, i.e. the relevant data must not change suddenly, which would render the computed results dirty.

6 Conclusion

After a short draft of the ULTRA language and its model-theoretic semantics, we have identified some requirements for an operational semantics. Then we have designed a flexible architecture for performing transactions specified in the ULTRA language. The architecture contains several loosely coupled components, which can be configured and tuned independently. In particular, the scheduler can behave either more optimistic, more aggressive, or more delaying. We claim that the architecture can even be adapted to evaluate similar update concepts with updates in the rule body [Che97, MW88b, MBM97], in particular the rule-based fragment of (Concurrent) Transaction Logic [BK94, BK96].

In the new architecture, hypothetical reasoning can be avoided, the logical evaluation of update queries and the materialization of updates can be interleaved, and the non-determinism can be handled efficiently.

Our next steps include the extension of the ideas presented above to the full ULTRA concept [WFF98a, WFF98b] (including concurrent conjunction and bulk updates) and their investigation at the theoretical level. In parallel, a thorough examination of the problems that arise from recursive update rules is necessary. Especially the scheduler must be able to handle a hierarchy of transactions whose depth is unknown in advance. In addition, the ULTRA semantics itself will be extended to arbitrary basic operations, which the new architecture can already accommodate.

References

[AFL+88] J. Aspnes, A. Fekete, N. A. Lynch, M. Merritt, and W. E. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proc. 14th Int. Conf. on Very Large Data Bases*, pages 431–444, 1988.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, 1989.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multi-database transaction management. *VLDB Journal*, 1:181–239, 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BK94] A. J. Bonner and M. Kifer. An overview of Transaction Logic. *Theoretical Computer Science*, 133:205–265, 1994.

[BK96] A. J. Bonner and M. Kifer. Concurrency and communication in Transaction Logic. In *Proc. Joint Int. Conf. and Symp. on Logic Programming (JICSLP '96), Bonn, Germany*, pages 142–156, 1996.

[Che95] W. Chen. Declarative updates of relational databases. *ACM Transactions on Database Systems*, 20(1):42–70, 1995.

[Che97] W. Chen. Programming with logical queries, bulk updates, and hypothetical reasoning. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):587–599, 1997.

[Dav78] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[Elk90] C. Elkan. Independence of logic database queries and updates. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 154–160, 1990.

[FLMW90] A. Fekete, N. A. Lynch, M. Merritt, and W. E. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41:65–156, 1990.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, 1996.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 95–106, 1990.

[Kow92] R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Germany, 2nd edition, 1987.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[MBM97] D. Montesi, E. Bertino, and M. Martelli. Transactions and updates in deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):784–797, 1997.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[MT98] V. W. Marek and M. Truszczynski. Revision programming. *Theoretical Computer Science*, 190:241–277, 1998.

[MW88a] D. Maier and D. S. Warren. *Computing with Logic: Logic programming with Prolog*. Benjamin/Cummings, Menlo Park, CA, 1988.

[MW88b] S. Manchanda and D. S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[Rei95] R. Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[Sha87] E. Shapiro, editor. *Concurrent Prolog (Collected Papers)*, volume 1. MIT Press, 1987.

[SSW94] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 442–453, 1994.

[vGRS91] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(2):620–650, 1991.

[WF97] C. A. Wichert and B. Freitag. Capturing database dynamics by deferred updates. In *Proc. Int. Conf. on Logic Programming (ICLP '97), Leuven, Belgium*, pages 226–240, 1997.

[WFF98a] C. A. Wichert, B. Freitag, and A. Fent. Logical transactions and serializability. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 133–163. Springer-Verlag, Berlin, Germany, 1998.

[WFF98b] C. A. Wichert, B. Freitag, and A. Fent. Logical transactions and serializability. Technical Report MIP-9807, Universität Passau (FMI), 1998. Available in the WWW: `http://www.fmi.uni-passau.de/~freitag/papers/`.

[WS92] G. Weikum and H. J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[ZF97] U. Zukowski and B. Freitag. The deductive database system *LOLA*. In *Proc. 4th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR '97), Dagstuhl, Germany*, pages 375–386, 1997.