A Logical Framework for the Specification of Transactions

Carl-Alexander Wichert
Burkhard Freitag
Alfred Fent

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

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

Universität Passau
Technical Report MIP-0002

Abstract

In this paper a logical framework called ULTRA is presented that can serve as the basis for a variety of update languages, among others extensions of the database language SQL, accommodating nested update queries and universal quantification over updates. We give a complete definition of the syntax and semantics of the ULTRA language based on concepts of logic databases.

A natural candidate for an instantiation of the ULTRA framework is a database update language which we discuss in this paper as one example. However, the framework is not tied to database applications and can also deal with states and actions of other external systems such as robot control systems or workflow engines. The semantics of an update goal is defined as a structured transition. Depending on the selected execution model, the materialization of a transition can be either deferred or performed immediately.

1 Introduction and Motivation

The concept of logic databases can serve as a clear and expressive foundation for various kinds of information systems. Rule-based languages including negation/aggregation or nested term structures offer a modeling power that lies above that of the well-known database language SQL [DD97]. Today, the static semantics and the evaluation of queries are well understood. The appropriate handling of dynamic behaviour in this context, however, is still a research issue. The main task is to develop concepts that allow the specification of (transactional) changes at a high logical level while featuring declarativity and compositionality. In this case, as known from the data retrieval task, the mapping from the logical to the physical layer can be performed and optimized transparently by a (suitably extended) database management system. This will lead to considerable improvements within the software development process. Although there exist many different concepts to integrate updates into logical languages, these approaches – regarded in isolation – cannot capture the semantics used in the database community.

In [WF97a, WF97b, WFF98b, WFF98c] we presented the rule-based update language ULTRA, whose semantics is based on the concept of deferred updates. 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 operations in a modular fashion. Constructs for concurrent composition, sequential composition, and bulk updates are provided, that subsume the classical database operations which are also supported by procedural extensions of SQL (see [DD97] for details). The semantics of an update goal w.r.t. a fixed database state is defined in terms of update request sets which collect insertion and deletion requests for ground EDB tuples and thus describe (hypothetical) transitions to other database states. This form of *deferred updates* was already used in Heraclitus [GHJ96], U-Datalog [MBM97], and the update calculus of Chen [Che95]. As two or more update request sets can be better combined than two or more different database extensions, the concept turns out to be well-suited for the representation of bulk updates, i.e. updates simultaneously performed for a specific set of parameter instances. We are also able to formulate a concurrent conjunction which is compositional, i.e. the semantics of the concurrent conjunction φ, ψ is defined in terms of the semantics of both subformulas φ and ψ. Compositionality is important for modularization and avoids the well-known problems of interleaving parallelism found in various programming languages, e.g. Java. From the basic notion of interpreting formulas we can derive a declarative, model-theoretic semantics for update programs. This distinguishes ULTRA from other practical extensions of database languages found in various database systems that only have operational meanings but no overall semantics.

The main contribution of this paper is the definition of the syntax and semantics of a logical framework, the instances of which extend logic programming languages, relational database languages, and logical action calculi.

- As a logic programming language, the ULTRA framework provides fully integrated declarative update and action constructs.
- Database-oriented instances of the ULTRA framework combine rule-based specification with reasoning about states (data retrieval) and transactional execution of updates.
- Action languages derived from the ULTRA framework can benefit from its intrinsic transactional properties under backtrackable action execution.

The major advantage of the results of this paper as compared to those presented by the authors in [WF97a, WF97b, WFF98b, WFF98c] is their generality, as ULTRA is not tied to specific basic operations, but lifted to the level of a generic framework for update languages. The framework approach and the seamless integration of transactional concepts known from database theory are also the main contributions as compared to other approaches mentioned above and in Section 5.1.

The concepts presented in this paper generalize our previous work in a significant way: The language of [WFF98b] was restricted to simple insertions and deletions as basic operations, and its semantics was tied to classical logic databases. Yet, it would be desirable to generalize the results to a greater family of update languages. Therefore, in this paper we present the generic ULTRA approach, that can serve as the foundation of a variety of specific logical update languages. The generic ULTRA approach will not only abstract from states and basic operations but also from the representation and the composition of deferred updates. To develop a semantics for a specific update language, which may deal with external operations, non-trivial basic operations in databases, operations on main-memory data structures, or even a combination of all these features, what remains to be done is to provide an adequate transition system including functions for the sequential and the concurrent composition of transitions. If these functions have monoid properties and some additional conditions are satisfied, all the properties which we are going to prove for our ULTRA framework, in particular those concerning the model-theoretic semantics of update programs, also hold for the instance. Note that at the operational level, the transitions can serve as the foundation of logs [BHG87, GR93].

In this paper we mainly concentrate on an insertion/deletion-based database language to illustrate the ULTRA framework. However, the ULTRA concept can be used as a framework for a great variety of update languages. This is exemplified by an alternative instantiation which is well-suited for the (mostly sequential) programming of external operations – as illustrated by a variation of the famous robot world example that can be found e.g. in [LRL+97, Rei95]. Despite major differences between the world of a robot arm and a database system, both update languages behave as instances of our ULTRA framework, see Figure 1.

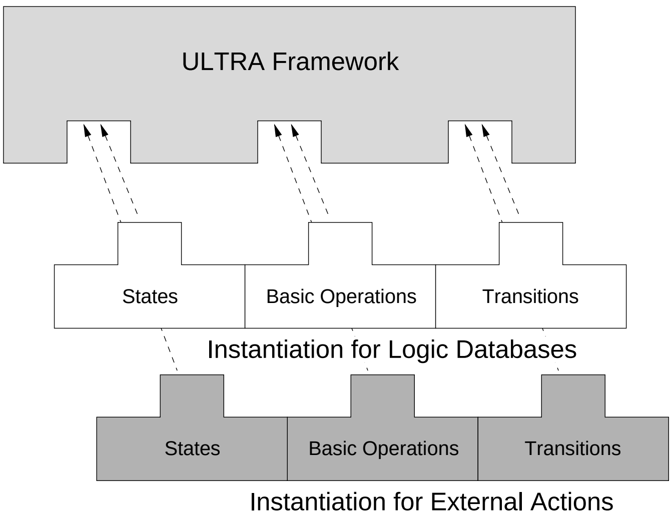

Figure 1: Instantiation of the ULTRA framework

From the evaluation point of view, a complex state transition specified in the ULTRA language can and (often) should be considered as a transaction. In particular, the ACID properties, which are well-known in database theory [BHG87, BN97, GR93], should hold. In the database-oriented instance of ULTRA, the atomicity and durability properties are easy to guarantee. We sketch a suitable two-phase strategy which is based on operationally deferred updates and hypothetical reasoning. If arbitrary basic operations are integrated into the language, other evaluation strategies, e.g. those proposed in [FWF00, WFF98a], will have to be used. Some additional work is also necessary to guarantee the isolation of independent transactions that are processed in parallel (see [WFF98b, WFF98c] for details). Note, however, that operational semantics and transaction processing strategies are outside the scope of this paper.

The rest of the paper is organized as follows: In Sections 2 and 3 we formalize syntax and semantics of the generic update language ULTRA and provide two specific instances of the framework: one instance is devoted to logic databases, the other to (sequential) external operations. The technical presentations are illustrated by non-trivial running examples, namely a calendar tool and a driver for a simple robot arm. We focus on transactional aspects in the ULTRA context and give an outline of a two-phase transaction processing strategy in Section 4, discuss some related work on dynamics in Section 5, and conclude the paper by Section 6. In the appendices, one can find the detailed proofs of assertions made in Section 3 as well as the complete running examples.

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

2 The Update Language ULTRA

2.1 The Generic ULTRA Language

In this section we present the generic ULTRA language, whose syntax is based on the syntax of first-order predicate logic [Llo87].

Definition 2.1 [Predicate Classes] We distinguish a set of *DB predicates* ($Pred_{DB}$), a set of *basic update predicates* ($Pred_{BU}$), and a set of *definable update predicates* ($Pred_{DU}$). □

The DB predicates refer to observable state information, whereas the basic update predicates refer to executable (atomic) update operations. In the ULTRA-based database language defined in Section 2.2, the basic operations will be simple insertions or deletions of tuples of base relations. However, other atomic operations like fixed SQL statements, calls to stored database procedures, or even external operations can be integrated. Moreover, the state information can comprise more than just classical base relations and views. For instance, return values of basic operations or external events can be modeled using auxiliary DB predicates. The definable update predicates can be regarded as names of complex update operations that may be executed as transactions. Their meaning is defined by an update program, i.e. by a set of update rules (see below) written in the ULTRA language.

Definition 2.2 [Terms, Atoms, DB Literals] A *term* is an arity-conform composition of function symbols, constants, and variables taken from given alphabets. A term without variables is called *ground*.

We assume that the alphabet of constants contains a special constant *all*.

Atoms are of the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and p is an n-ary predicate. We can distinguish between *DB atoms*, *basic update atoms*, and *definable update atoms* depending on which set of predicates p belongs to. *DB literals* are DB atoms $q(t_1, \ldots, t_n)$ or negated DB atoms $NOT\ q(t_1, \ldots, t_n)$. The groundness property of terms can be generalized to atoms, literals, and formulas defined below in the natural way. □

Technically, the reserved constant *all* is needed for the representation of non-ground terms as ground terms at the semantical level in certain cases. The constant *all* is important for algebraic properties and must not be used at the syntactical level, i.e. in programs or queries.

Definition 2.3 The set of all ground terms is called the *Herbrand universe* and is denoted by $\mathcal{U}$. The set of all ground DB atoms is called the *Herbrand base* and is denoted by $\mathcal{B}$. The set of all ground basic update atoms is called the *basic update base* and is denoted by $\mathcal{B}_{BU}$. The set of all ground definable update atoms is called the *definable update base* and is denoted by $\mathcal{B}_{DU}$. □

In the following we use the abbreviation $\vec{t}$ for a finite sequence $t_1, \ldots, t_n$ of terms. Similarly, we write $\vec{X}$ for a finite sequence $X_1, \ldots, X_n$ of variables and $\vec{all}$ for a repetition of the constant *all*.

After having recalled some preliminaries, we now define the specific elements of the ULTRA language. Its basic elements are called update literals.

Definition 2.4 [**Update Literals**] The set of *update literals* is defined by the following cases:

1. Every DB literal is an *update literal*.
2. NOP is an *update literal* ("no operation").
3. Every basic update atom $u(\vec{t})$ is an *update literal*.
4. Every definable update atom $p(\vec{t})$ is an *update literal*.

□

Update literals specify retrieval (DB literals), atomic modifications (basic update atoms), and calls to defined complex operations (definable update atoms). Update literals do not need to be ground, i.e. they may contain variables. However, they should not contain the reserved constant *all*.

Update literals can be composed to form more complex update formulas. First, we define the general notion of update formulas, a subset of which forms the set of update goals that we will introduce later.

Note that different from other approaches, e.g. [Che97], we do not allow negated basic or definable update atoms. Negation is permitted only at the retrieval level.

Definition 2.5 [**Update Formulas**] The set of *update formulas* is defined inductively by the following cases:

1. Every update literal is an *update formula*.
2. Let φ and ψ be update formulas. The *concurrent conjunction* φ, ψ and the *sequential conjunction* $\varphi : \psi$ are *update formulas*.
3. Let φ and ψ be update formulas. The *disjunction* $\varphi \vee \psi$ is an *update formula*.
4. Let φ be an update formula and $\vec{X}$ be a finite sequence of variables. Then $\exists \vec{X} \varphi$ is an *update formula*.
5. Let A be a DB atom, let φ be an update formula, and let $\vec{X}$ be a finite sequence of variables. The *bulk quantification* $\# \vec{X} [A \mapsto \varphi]$ is an *update formula*.
6. Let φ and ψ be update formulas. The *implication* $\varphi \rightarrow \psi$ is an *update formula*.
7. Let φ be an update formula and $\vec{X}$ be a finite sequence of variables. Then $\forall \vec{X} \varphi$ is an *update formula*.

If necessary we use square brackets "[...]" to indicate the operator bindings in a composite formula.

□

In Definition 2.5 we have defined a concurrent and a sequential conjunction for specifying simultaneous and successive update operations, respectively, and a bulk quantification for specifying set-oriented updates in terms of single updates. The intuitive reading of the bulk quantification is "for all $\vec{X}$ such that A holds perform update φ". The concurrent conjunction and the bulk quantification specify simultaneous operations that can be performed in parallel or in an interleaved fashion. From the logical point of view, however, each thread will refer to local states and will not be aware of the other threads. The resulting state will finally be derived by accumulating the local changes. At a first sight this looks like uncontrolled concurrency. However, suitable isolation properties can be defined and exploited to ensure the mutual exclusion of the parallel threads. We believe that our approach does not sacrifice generality too much while avoiding the semantical problems of interleaving parallelism.

The disjunction and the existential quantification enable the compact specification of non-deterministic updates. Further, the existential quantification helps to deal with local variables inside a bulk quantifier. However, disjunction and existential quantification are not essential for the rule-based language, as their semantics can be simulated by auxiliary rules as in classical logic programming [MW88a]. The implication and the universal quantification are used to construct update rules for the definition of complex update operations.

From a semantical point of view, the existential quantification and the bulk quantification can be regarded as a generalization of the disjunction and the concurrent conjunction, respectively.

Definition 2.6 [Update Goals] An *update goal* is an update formula not containing the implication $\rightarrow$ or the quantifier $\forall$, i.e. the set of update goals is defined by cases 1 to 5 of Definition 2.5. □

Update goals form rule bodies or top-level update queries (see Examples 2.15 and 2.18).

Definition 2.7 A variable occurring inside a formula is *free* unless being in the scope of a quantifier ($\forall$, $\exists$, or $\#$). An update formula is called *ground*, if it does not contain any free variables. □

Definition 2.8 For an update formula φ we denote by $\varphi[X/t]$ the new formula φ' that results from replacing simultaneously all free occurrences of the variable X by the term t.

Let $X_1, \ldots, X_n$ be a sequence of disjoint variables and $t_1, \ldots, t_n$ be a sequence of terms. $\varphi[\vec{X} \,/\, \vec{t}\,]$ denotes the simultaneous substitution of each variable X_i by the corresponding term t_i. □

Definition 2.9 [Update Rules] An *update rule* is a universal closure $\forall(U \rightarrow p(\vec{t}\,))$, also denoted by $p(\vec{t}\,) \leftarrow U$, where $p(\vec{t}\,)$ is a definable update atom and U is an update goal. Rules without a head $\leftarrow U$ denote *(top-level) queries*. An update rule with an empty body is called an *update fact*. The empty body corresponds to the update literal NOP. □

Definition 2.10 [Update Program] An *update program* P_{UP} is a set of update rules. □

An update program specifies the meaning of the definable update atoms occurring in the rule heads. It can be compared with a set of procedure definitions in classical programming languages. However, the ULTRA notion is less operational and more convenient for programming related to (logic) databases and information processing.

Up to now we have defined the basic concepts of ULTRA, in particular the notion of update rules and update programs. A final ULTRA instance must provide the necessary alphabets of constants, function symbols, and various predicates. It also may provide additional syntactical elements that will be relevant for the semantics, e.g. a program for computing the truth interpretation in each state. The instances defined below, however, are more abstract, they just describe some additional constraints that must hold for the final instances. The final instances are implicitly given by the example applications, e.g. a calendar manager or a robot interface. Note that the instantiation will be more interesting at the semantical level, which is treated in Section 3.

2.2 Instantiating the Framework: ULTRA for Logic Databases

In this section we show that the database language which has been proposed in [WFF98b] as a stand-alone concept is in fact an instance of the generic ULTRA language of Section 2.1.

As usual in logic databases we distinguish an *extensional database (EDB)* comprising some (persistent) base relations, e.g. a relational database, and an *intensional database (IDB)* defined by a set of normal deductive rules, i.e. DATALOG with function symbols and negation [Llo87]. Consequently, the set of DB predicates as defined in Definition 2.1 is partitioned as follows.

Definition 2.11 [**DB Predicates, DB Atoms**] The set of DB predicates $Pred_{DB}$ is partitioned into a set of *EDB predicates* and a set of *IDB predicates*.

Every DB atom is either called an *EDB atom* or an *IDB atom* depending on its predicate symbol. □

Definition 2.12 [**EDB Instance**] Let $\mathcal{B}$ be the given Herbrand base. A set DB of ground EDB atoms $r(\vec{t}) \in \mathcal{B}$ is called an *EDB instance* over $\mathcal{B}$. □

An EDB instance assigns truth values to the EDB atoms as usual and represents a database state. The truth value of the IDB atoms can be derived from the IDB program w.r.t. the state semantics chosen (see Section 3.3 for details).

Definition 2.13 [**IDB Rules, IDB Program**] *IDB rules* are universal closures of implications built from an IDB head atom $q(\vec{t})$ and a body $L_1, \ldots, L_n$ of DB literals (denoted by $q(\vec{t}) \leftarrow L_1, \ldots, L_n$). IDB rules with an empty body are called *IDB facts*.

An *IDB program* P_{IDB} is a set of IDB rules. □

In the following we assume that two basic operations are available in the logic database instance of the ULTRA framework: the insertion and the deletion of an EDB atom, i.e. of a tuple in an EDB relation. So we provide corresponding basic update atoms $INS \ldots$ and $DEL \ldots$ for every EDB atom $r(\vec{t})$. Note that we do not consider basic updates on the IDB (view updates) in this paper.

Definition 2.14 [**Basic Update Atoms**] The set of basic update atoms consists of elements of the form $INS\, r(\vec{t})$ and $DEL\, r(\vec{t})$, where $r(\vec{t})$ is an EDB atom. Note that this also fixes the basic update base $\mathcal{B}_{BU}$. □

An extended logic database (deductive database) consists of three user-definable components: a persistent EDB, an IDB program P_{IDB}, and an update program P_{UP}.

To illustrate the properties and capabilities of the ULTRA language and the formal results presented in this paper, we use a simplified version of a personal calendar. In the examples only part of the full program, which can be found in Appendix B, is shown.

Example 2.15 [**Personal Calendar**] In the calendar model used in our examples, appointments have a unique identifier and may occupy one or more consecutive time slots. Time slots may be of any length, but we assume that a slot represents one hour. The calendar is based on two EDB relations: a relation $entry(Day, Slot, ID)$ which associates a time slot $Slot$ on day Day with an appointment identifier ID and a relation $description(ID, Text)$ that contains the descriptive text $Text$ for an appointment with identifier ID. The reserved identifier 0 represents free time slots. In a real world implementation, free slots would of course not be recorded in the database but rather be computed, either by joining the *entry* relation with a relation representing all possible slots per day, or by using a computed predicate that enumerates the possible slots. For the sake of simplicity, we consider only one fixed week.

In the rest of the paper we always refer to the EDB instance DB_0 shown in Table 1. The constant *mon* refers to the Monday of the fixed week. This EDB instance can be interpreted as follows: The owner of the calendar has a meeting with Mr. Dean on Monday from 12pm to 2pm, she wants to visit the hairdresser on Monday at 3pm, etc. The time slots on Monday from 10am to 12pm and from 2pm to 3pm are not reserved, yet. To keep the tables short we omit the entries for the other days.

In this setting, possible basic update atoms are $DEL\ entry(mon, 10, 0)$, $INS\ entry(mon, 10, 23)$, and $INS\ description(23, \text{"Presentation"})$. They specify the deletion of the tuple $(mon, 10, 0)$ from the *entry* relation, the insertion of the tuple $(mon, 10, 23)$ into the *entry* relation, and the insertion of the tuple $(23, \text{"Presentation"})$ into the *description* relation, respectively. Together, this amounts

entry	*Day*	*Slot*	*ID*
	mon	9	21
	mon	10	0
	mon	11	0
	mon	12	7
	mon	13	7
	mon	14	0
	mon	15	8
	mon	16	10

description	*ID*	*Text*
	7	Meeting Mr. Dean
	8	Hairdresser
	10	Review
	21	Call Mr. Miller

Table 1: Sample EDB instance DB_0

to inserting a new appointment "Presentation" on Monday from 10am to 11am into the calendar database.

Let us now have a closer look at the update program P_{UP}. In our calendar example there is – among others – the update rule

$$\begin{array}{lcl} do\_insert(D,S,L,T) & \leftarrow & \mathsf{newid}(ID),\ do\_allocate(D,S,L,ID), \\ & & INS\ description(ID,T) \end{array}$$

which specifies the insertion of a new entry with descriptive text T, starting at time slot S on day D and having a duration of L slots. The definition of the *do_insert* predicate consists of three components: newid is a (built-in) predicate that returns a new identifier ID. From the semantical point of view it can be regarded as a predicate which is true for exactly one constant, however the constant varies between independent evaluations. Such a feature is provided in many database systems to avoid concurrency problems when searching for unused key values. *do_allocate* is a recursive auxiliary update predicate which allocates L consecutive time slots needed for identifier ID in relation *entry* (see Appendix B for details). As before, $INS\ description(ID,T)$ inserts the descriptive text of the new appointment into the relation *description*.

Since the subgoals $do\_allocate(D,S,L,ID)$ and $INS\ description(ID,T)$ refer to different EDB relations, they can be evaluated simultaneously and thus are combined by concurrent conjunction ",". Note that also the subgoal newid(ID) is connected by concurrent conjunction. From the logical point of view it can be evaluated concurrently with the rest. A lazy evaluation method, for instance, may work with an open value of ID, until it is clear that the the allocation of the slots is possible. However, in classical implementations newid(ID) would be implicitly scheduled to be evaluated first such that it can produce a binding for ID. This kind of partial sequentialization can be achieved by a suitable sideways information passing strategy, but we will not consider such topic in this paper.

For the deletion of an entry, a predicate *do_delete* can be defined that uses an auxiliary predicate *do_deallocate*. The latter predicate may be defined as follows using the bulk quantifier, which says that for all entries in *entry* with appointment identifier ID the corresponding slot has to be marked free.

$$\begin{array}{lcl} do\_deallocate(ID) & \leftarrow & \#D,S \\ & & [\ entry(D,S,ID)\ \mapsto \\ & & \quad [\ DEL\ entry(D,S,ID),\ INS\ entry(D,S,0)\ ]\ ] \end{array}$$

Appointments are frequently moved from one time interval to another. This can be achieved using the update predicate *do_move* shown below which moves an appointment with identifier ID to day D and time slot S. Again, *do_move* is built upon the predefined auxiliaries *do_allocate* and *do_deallocate*. The predicate *duration_of* counts the number L of slots reserved for the appointment ID, it is defined by an IDB rule (see Appendix B).

$$\begin{array}{lll} do\_move(ID, D, S) & \leftarrow & [\ duration\_of(ID, L),\ do\_deallocate(ID)\ ]\ : \\ & & do\_allocate(D, S, L, ID) \end{array}$$

Here we need the sequential conjunction ":" to specify that the second subgoal is to be evaluated in the (hypothetical) database state that results from completing the operation specified by the subgoals on the left. In our example this is necessary to ensure that an entry does not block its own movement. So, first all time slots assigned to that particular entry are freed, and then the allocation of the slots at the new starting time is attempted. For *do_move* to succeed, both subgoals of the sequential conjunction must succeed. □

For the development of isolation concepts we have to regard the retrieval dependencies within the IDB program. The definitions are as usual, cf. [ABW88].

Definition 2.16 [Dependency Graph] Let P_{IDB} be an IDB program, and let p and q be EDB or IDB predicates. *p depends directly* on q, denoted by $p \leftarrow_{P_{IDB}} q$, iff there exists a rule in P_{IDB} such that p is the head predicate and q occurs in the body. The relation $\leftarrow_{P_{IDB}}$ on the set $Pred_{DB}$ of DB predicates defines the edges of the *dependency graph* of P_{IDB}. □

Note that the relation $\leftarrow_{P_{IDB}}$ only refers to the retrieval part, i.e. EDB and IDB predicates. The predicate dependencies given by the update program, i.e. by the update rules, are not considered here.

The set of EDB predicates a predicate q depends on will be used to detect read/write conflicts between two transactions.

Definition 2.17 Let P_{IDB} be an IDB program and q be an EDB or IDB predicate. We define

$$Def_E[P_{IDB}](q) := \{r \mid r \text{ is an EDB predicate } \wedge\ q \leftarrow^*_{P_{IDB}} r\},$$

where $\leftarrow^*_{P_{IDB}}$ denotes the reflexive and transitive closure of the relation $\leftarrow_{P_{IDB}}$. □

Whenever a predicate q is accessed by a transaction, the predicates contained in $Def_E[P_{IDB}](q)$ will be marked as read as if they were accessed instead of q. Then it is easy to detect conflicts between read access and basic update operations on the EDB.

2.3 Instantiating the Framework: ULTRA for External Operations

While in Section 2.2 we have shown how a logic database language supporting updates can be derived from the ULTRA framework by instantiation, we now focus on a rather different application domain: we will show how ULTRA can be used for the specification of complex external operations. At the syntactical level, no refinements of the ULTRA framework are necessary. So, we can proceed with an example. Note that our example and the semantical concepts defined in Section 3.4 are simplified due to space limitations.

Example 2.18 [Robot World] Consider a robot arm working over a range of n positions $p_1, \dots, p_n$. The robot arm can move to a certain position P, where it can try to pick up or put down a block. Corresponding to these basic operations, let the basic update predicates *move*, *pickup*, and *putdown* be given. Let *move* be unary with the position as its only parameter, let the other predicates be nullary. We assume that *pickup* and *putdown* cause (successful) idle movements, whenever there is no block to operate on or there are conflicts between multiple blocks. In other words, the robot arm is not aware of what it is really doing, but rather just performs the predefined movements. However, let us assume that the robot arm is equipped with a sensor that checks whether it actually holds a block or it is empty. Syntactically, the state of the sensor is modeled by a nullary predicate *empty* (a DB predicate in the sense of Section 2.1).

We are now going to specify a more sophisticated operation to pick up a block at some position P. As a precondition we require that the robot arm is empty, as a postcondition that it is not empty, i.e. that it actually has picked up a block and has not just made an idle movement to the floor. Note that for the sake of the example, we do not model any knowledge about the environment of the robot, e.g. where the blocks are placed. In particular, no persistent database storage is involved. In our example, the ULTRA system can just let the robot arm perform the basic operations *move*, *pickup*, and *putdown* and check the sensor by querying the *empty* predicate. In this setting, the desired operation can be specified by the following update goal:

$$empty \ : \ move(P) \ : \ pickup \ : \ NOT\ empty$$

Even if the simplified robot arm works sequentially, it is imaginable that the invocation of an action and the question about the sensor can be performed simultaneously. As we will see in Section 3.4, this can also be reflected by the formal semantics of ULTRA. So we may decide to program the complex pickup operation as follows using the concurrent conjunction, too:

$$pickup\_at\_position(P) \quad \leftarrow [\ empty,\ move(P)\ ] \ : \ pickup \ : \ NOT\ empty$$

The complete robot example featuring some other operations can be found in Appendix C. □

3 Semantics of Formulas and Programs

In this section we present the model-theoretic semantics of ULTRA, which is based on the concept of *deferred updates*. Essentially, updates are not just considered as side effects which occur during the evaluation of an update goal. Referring to a fixed initial state, each update goal determines (one or more) *possible* transitions that may or may not be materialized later. Not earlier than at *materialization time* does a possible transition cause an actual transition leading to some new state.

For the logical semantics, it is legitimate to call all states except the initial state *hypothetical*. However, for the operational semantics, which is not in the focus of this paper, several implications arise: obviously, retrieval which is necessary to provide variable bindings cannot be deferred but must be done immediately during an evaluation. When referring to hypothetical states, the retrieval must be either based on hypothetical reasoning without changing the physical state – as proposed in Section 4 –, or a selection of computed transitions must be physically executed with an undo option, cf. [FWF00, WFF98a]. The latter point of view also applies for the robot example (cf. Example 2.18), if the robot world is considered as a black box. In this case, the robot arm must actually perform the actions, such that its sensors can investigate the resulting intermediate state. In case of a failure, the actions must be undone, i.e. the robot arm and its environment must return to a previous state. This form of recovery is not necessary in the database approach which can be semantically and *operationally* based on deferred updates, but on the other hand requires hypothetical reasoning. Of course, it would also be possible to model enough knowledge about the effects of further actions than insert and delete to enable hypothetical reasoning in other contexts. Reiter [Rei95], for instance, provides a viable axiomatization technique in terms of the situation calculus. Axioms that describe the effects of some external actions, e.g. the movements of the robot arm, could be integrated into an ULTRA evaluation engine. Unfortunately, the semantics of external actions is typically more difficult than the semantics of database operations like insert and delete. The frame problem causes further intractabilities for bigger sets of basic operations and observable predicates. Consequently, there exist many questions and problems w.r.t. an operational semantics, and its design is not a simple, straight-forward task. The semantics we present in this section, however, is independent from the operational model, and the latter can and should be designed and optimized w.r.t. a more specific instantiation of the ULTRA framework.

To be able to detect conflicts between immediate retrieval and deferred updates, the semantics allows to assign logging transitions to every DB atom. It must be guaranteed by semantical properties that a certification of the corresponding (immediate) read access is possible when the materialization of the logging transition takes place. Operationally, deferred updates with logged read operations can be well combined with optimistic synchronization protocols [BHG87]. The read-isolation problem is not investigated in this paper due to space limitations. We refer the reader to [WFF98b, WFF98c].

3.1 Preliminaries and Preconditions

Before we can define the semantics of update formulas, we have to consider some preliminaries.

It may be necessary to deal with three-valued interpretations for the DB predicates, for instance, when their interpretation is determined by deductive rules containing negation [AB94].

Definition 3.1 [**Three-Valued Interpretation**] A *three-valued interpretation* I over the Herbrand base $\mathcal{B}$ assigns to each ground DB atom $A \in \mathcal{B}$ one of the truth values "true" (denoted by $I \models A$), "false" (denoted by $I \models \neg A$), or "unknown".

$\mathcal{I}^3_{\mathcal{B}}$ denotes the set of all three-valued interpretations I over the Herbrand base $\mathcal{B}$. □

Next, we introduce some objects and constructs needed in the definition of the interpretation domain for the update formulas (see Section 3.2). In the following definition we formalize the components of a transition system and the algebraic properties that must hold. The transition system is one of the open parameters of the ULTRA framework.

Definition 3.2 [**Transition System**] A *transition system* is a tuple $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ where

- $\mathcal{S}$ is a set of *states*,
- $\mathcal{T}$ is a set of *transitions*,
- $\mathcal{T}_{Cons} \subseteq \mathcal{T}$ is set of transitions which are called *consistent*,
- $\oplus_E$ denotes an *execution* semantics

  $$\oplus_E : \mathcal{S} \times \mathcal{T}_{Cons} \to \mathcal{S}$$

  such that $s \oplus_E \Delta$ represents the state s' resulting from the execution of Δ starting in the state s,
- $\Delta_\varepsilon \in \mathcal{T}_{Cons}$ denotes a special *neutral* transition,
- $\sqcup$ is a *concurrent composition* function

  $$\sqcup : \mathcal{T} \times \mathcal{T} \to \mathcal{T},$$
- $\bigsqcup$ is a *concurrent composition* function for multi-sets of transitions,
- $\oplus$ is a *sequential composition* function

  $$\oplus : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$$

such that following algebraic properties hold:

1. For the concurrent composition $\sqcup$, the following holds:

   (a) $\sqcup$ is commutative.
   (b) $\sqcup$ is associative.
   (c) Δ_ε is a neutral element for $\sqcup$.

   In other words, $(\mathcal{T}, \sqcup, \Delta_\varepsilon)$ forms a commutative monoid.

2. For the sequential composition $\oplus$, the following holds:

   (a) $\oplus$ is associative.
   (b) Δ_ε is a neutral element for $\oplus$.

   In other words, $(\mathcal{T}, \oplus, \Delta_\varepsilon)$ forms a monoid.

3. Let $\Delta_1, \Delta_2 \in \mathcal{T}$ be arbitrary transitions. Then the following holds:

   (a) $\Delta_1 \in \mathcal{T}_{Cons} \;\wedge\; \Delta_2 \in \mathcal{T}_{Cons} \;\Longleftarrow\; \Delta_1 \sqcup \Delta_2 \in \mathcal{T}_{Cons}$
   (b) $\Delta_1 \in \mathcal{T}_{Cons} \;\wedge\; \Delta_2 \in \mathcal{T}_{Cons} \;\Longrightarrow\; \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons}$
   (c) $\Delta_2 \in \mathcal{T}_{Cons} \;\Longleftarrow\; \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons}$

4. Let $s \in \mathcal{S}$ be a state, and let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be consistent transitions. Then the following holds:

   (a) $(s \oplus_E \Delta_1) \oplus_E \Delta_2 = s \oplus_E (\Delta_1 \oplus \Delta_2)$

   For every state $s \in \mathcal{S}$,

   (b) $s \oplus_E \Delta_\varepsilon = s$

   holds, i.e. Δ_ε is neutral for $\oplus_E$.

5. For arbitrary multi-sets T over $\mathcal{T}$ and arbitrary transitions $\Delta \in \mathcal{T}$ the equality

   (a) $\bigsqcup(\{\Delta\} \uplus T) = \Delta \sqcup \bigsqcup T$

   holds. (We write $\uplus$ for the union of multi-sets.) Further the following holds for the empty multi-set:

   (b) $\bigsqcup \emptyset = \Delta_\varepsilon$

$\square$

Transitions are semantical objects to represent the changes between two states. Every consistent transition leads from a given current state to a next state when it is executed. A transition which is not consistent does not need to be executable in any state and thus may not represent state changes. Later we will restrict ourselves to consistent transitions, however, the more general concept helps to simplify the formal treatment of the specific ULTRA instances.

Example 3.3 For the database language of Section 2.2, the states will be defined as the various EDB instances, and the transitions will be defined as sets which contain insertion and deletion requests for EDB tuples. A consistent transition must not specify the simultaneous insertion and deletion of the same tuple. See Section 3.3 for more details.

When using the ULTRA framework for external operations as sketched in Section 2.3, the states will be defined as the states of the external system, and consistent transitions will be defined as lists of ground basic update atoms. The execution $\oplus_E$ simply models the state change that results from performing the external actions corresponding to the basic update atoms in the listed order. See Section 3.4 for more details. $\square$

Transitions must be composable by a concurrent composition and a sequential composition, such that complex combinations of transitions can be expressed by a single transition. The composition functions may be defined for inconsistent transitions or yield inconsistent ones. To be able to formulate the semantics of the bulk quantification, we also need a concurrent composition for (possibly infinite) multi-sets of transitions.

The algebraic properties required in Definition 3.2 are justified by experiences in the real world. They are also necessary to obtain several expected properties of the semantics of the update formulas (see Section 3.2). Property 3 is important, since we will restrict the semantics of update formulas to consistent transitions. Property 5 states that the concurrent composition of multi-sets fits with the concurrent composition of two transitions. This is important, as $\bigsqcup$ is not defined inductively by $\sqcup$. In contrary, $\bigsqcup$ is an additional parameter and has to be defined for infinite multi-sets as well. Note that we do not define further requirements at the generic framework level. For example, we do not formalize dependencies between the concurrent and the sequential composition of transitions. When the framework is instantiated, further properties may be identified, which can be exploited for an operational semantics.

Definition 3.4 [**Conformity**] Two or more consistent transitions are called *conforming* with each other, if their concurrent composition is consistent. □

Next, the basic update atoms and the DB atoms will be related to the given transition system. Further, the interpretation of the DB atoms will be specified for each state. Informally speaking, the parameters defined below serve as a bridge between the syntactical parameters of the ULTRA framework and the transition system.

Definition 3.5 [**DB Interpretation**] A mapping

$$I_{DB} : \mathcal{S} \to \mathcal{I}^3_{\mathcal{B}},$$

which assigns a set of true and false DB atoms (observations) to each state, is called a *DB interpretation.* □

Definition 3.6 [**Logging Transition Assignment**] A mapping

$$Log : \mathcal{B} \to \mathcal{T}_{Cons},$$

which assigns a consistent transition to each ground DB atom, is called a *logging transition assignment*. □

Definition 3.7 [**Update Transition Assignment**] A mapping

$$Upd : \mathcal{B}_{BU} \to \mathcal{T}_{Cons},$$

which assigns a consistent transition to each ground basic update atom, is called an *update transition assignment.* □

The DB interpretation I_{DB} provides a state-dependent meaning for the DB atoms. The mappings Log and Upd are used to assign semantical counterparts – in terms of consistent transitions – to the (syntactical) update literals. Note that the assignments are state-independent. This is adequate, since the execution semantics $\oplus_E$ already handles the state-dependence. The logging transitions are used to record what state information has been queried and thus simply serve as marks. They should typically not change the state when they are executed. However, this property is not relevant for the semantics. If it is not satisfied, not only the basic update atoms, but also the DB atoms may become afflicted with side effects.

3.2 Interpretation of Update Formulas

We will now define the interpretation of update formulas. The semantics will be defined w.r.t. an arbitrary but fixed *initial* state $s_0 \in \mathcal{S}$. However, for the sake of readability we do not parameterize the constructs introduced in the following with this state. Any (non-initial) state $s \in \mathcal{S}$ is represented by a transition $\Delta \in \mathcal{T}_{Cons}$, such that the execution of Δ in the state s_0 would lead to s.

$$s_0 \quad \xrightarrow{\Delta} \quad s$$

We call s a *hypothetical* state, as it does not need to become a physical state. Note that s_0 can be represented by the neutral transition Δ_ε.

The semantics of ULTRA is not based on relations between different states like in a dynamic logic [KT90], but rather on *deferred* transitions. An interpretation I is a mapping from the set of ground update formulas to the power-set of $\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$, i.e. $I(\varphi) \subseteq \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$ for every ground formula φ. The first component of each pair $(\Delta_C, \Delta) \in I(\varphi)$ points to a (hypothetical) current state s_{Curr} (reachable from s_0) in which φ is to be evaluated, the second component refers to a transition that would lead to the next state s_{Next} if applied to the current state s_{Curr}. Every pair (Δ_C, Δ) corresponds to an allowed state change.

$$(\quad s_0 \quad \xrightarrow{\Delta_C} \quad) \quad s_{Curr} \quad \xrightarrow{\Delta} \quad s_{Next}$$

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 states.

As defined below, I will be an extension of an interpretation I_{UP} of the definable update atoms. In other words, if an interpretation I_{UP} of the definable update atoms is given, the interpretation I of all ground update formulas can be derived according to Definition 3.8. In Section 3.5 we will characterize a particular interpretation I_{UP} derived from the update program P_{UP}.

Note that we define the semantics of quantifiers over a replacement of variables by ground terms. This is correct, because we use the Herbrand pre-interpretation [Llo87], where every domain element can be represented by a ground term of $\mathcal{U}$.

Definition 3.8 [Interpretation of Update Formulas] Let $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ be a transition system with initial state $s_0 \in \mathcal{S}$. Let $I_{DB} : \mathcal{S} \to \mathcal{I}^3_{\mathcal{B}}$ be a DB interpretation, $Log : \mathcal{B} \to \mathcal{T}_{Cons}$ be a logging transition assignment, and $Upd : \mathcal{B}_{BU} \to \mathcal{T}_{Cons}$ be an update transition assignment. Let I_{UP} be an interpretation of the definable update atoms, i.e. a mapping

$$I_{UP} : \mathcal{B}_{DU} \to 2^{\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}}.$$

We define the *interpretation I of update formulas* as an extension of I_{UP} to arbitrary ground update formulas inductively as follows. Note that only consistent transitions $\Delta \in \mathcal{T}_{Cons}$ are considered.

Base cases:

1. DB literal (DB)

   Let $q(\vec{t}) \in \mathcal{B}$ be a DB atom.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(q(\vec{t}))$ $:\Longleftrightarrow$
   $I_{DB}(s_0 \oplus_E \Delta_C) \models q(\vec{t})$ and $\Delta = Log(q(\vec{t}))$

   $(\Delta_C, \Delta) \in I(NOT\ q(\vec{t}))$ $:\Longleftrightarrow$
   $I_{DB}(s_0 \oplus_E \Delta_C) \models \neg q(\vec{t})$ and $\Delta = Log(q(\vec{t}))$

2. NOP literal (NOP)

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(NOP) \;:\Longleftrightarrow\; \Delta = \Delta_\varepsilon$

3. Basic update atom (BU)

   Let $u(\vec{t}) \in \mathcal{B}_{BU}$ be a basic update atom.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(u(\vec{t})) \;:\Longleftrightarrow\; \Delta = Upd(u(\vec{t}))$

4. Definable update atom (DU)

   Let $p(\vec{t}) \in \mathcal{B}_{DU}$ be a definable update atom.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

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

Inductive cases:

1. Concurrent conjunction (CCj)

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

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

   there exist $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ such that:
   $(\Delta_C, \Delta_1) \in I(\varphi)$ and $(\Delta_C, \Delta_2) \in I(\psi)$ and $\Delta = \Delta_1 \sqcup \Delta_2$

2. Sequential conjunction (SCj)

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

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

   there exist $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ such that:
   $(\Delta_C, \Delta_1) \in I(\varphi)$ and $(\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi)$ and $\Delta = \Delta_1 \oplus \Delta_2$

3. Disjunction (Dj)

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

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

   $(\Delta_C, \Delta) \in I(\varphi)$ or $(\Delta_C, \Delta) \in I(\psi)$

4. Existential quantification (Ex)

   Let φ be an update formula, and let $X_1, \ldots, X_n$ be variables such that $\exists \vec{X}\, \varphi$ is ground.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(\exists \vec{X}\, \varphi) \;:\Longleftrightarrow$

   there exists a ground term tuple $(\vec{t}) \in \mathcal{U}^n$
   such that $(\Delta_C, \Delta) \in I(\varphi[\vec{X} \,/\, \vec{t}])$

5. Bulk quantification (Bulk)

   Let A be a DB atom that contains exactly the variables $X_1, \ldots, X_n$,
   let φ be an update formula such that $\varphi[\vec{X} \,/\, \vec{t}]$ is ground for every term tuple $(\vec{t}) \in \mathcal{U}^n$,
   and let $I(\varphi[\vec{X} \,/\, \vec{t}])$ be already defined for every ground tuple $(\vec{t}) \in \mathcal{U}^n$.
   For $\Delta_C \in \mathcal{T}_{Cons}$ let $T_{A,\Delta_C} := \{(\vec{t}) \in \mathcal{U}^n \mid I_{DB}(s_0 \oplus_E \Delta_C) \models A[\vec{X} \,/\, \vec{t}]\}$.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(\#\, \vec{X}\, [A \mapsto \varphi]) \;:\Longleftrightarrow$

   $T_{A,\Delta_C} = \emptyset$ and $\Delta = Log(A[\vec{X} \,/\, \vec{all}\,])$

   or

   $T_{A,\Delta_C} \neq \emptyset$ and there exists a function $f : T_{A,\Delta_C} \to \mathcal{T}_{Cons}$ such that:
   $\forall (\vec{t}) \in T_{A,\Delta_C} : (\Delta_C, f(\vec{t})) \in I(\varphi[\vec{X} \,/\, \vec{t}])$
   and $\Delta = Log(A[\vec{X} \,/\, \vec{all}\,]) \;\oplus\; \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C}} f(\vec{t})$

6. Implication (Impl)

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

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

   $(\Delta_C, \Delta) \in I(\varphi) \Longrightarrow (\Delta_C, \Delta) \in I(\psi)$

7. Universal quantification (Univ)

   Let φ be an update formula, and let $X_1, \ldots, X_n$ be variables such that $\forall \vec{X}\, \varphi$ is ground.

   For all $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ define:

   $(\Delta_C, \Delta) \in I(\forall \vec{X}\, \varphi) \;:\Longleftrightarrow$

   for arbitrary ground term tuples $(\vec{t}) \in \mathcal{U}^n$ the following holds:

   $(\Delta_C, \Delta) \in I(\varphi[\vec{X} \,/\, \vec{t}])$

$\square$

Definition 3.8 essentially formalizes what kind of transitions Δ are necessary to satisfy the corresponding update formulas. In operational terms this amounts to the definition of the transitions Δ that are generated during the evaluation. Note how basic update atoms are related to the update transitions (*Upd*) and how the concurrent/sequential conjunction is related to the concurrent/sequential composition of transitions. Case (Bulk) concerning the bulk quantification should be explained in more detail: consider a fixed current state represented by Δ_C. First, all ground term tuples $(\vec{t})$, such that the instance $A[\vec{X} \,/\, \vec{t}]$ of the atom A is true in the current state, have to be collected in T_{A,Δ_C}. Then either T_{A,Δ_C} is empty and the bulk quantification is successful without any single updates, or for each term tuple $(\vec{t})$ in T_{A,Δ_C}, a corresponding transition (w.r.t. the ground update formula $\varphi[\vec{X} \,/\, \vec{t}]$) must be chosen and incorporated into Δ, which represents the resulting bulk update. The choice is reflected by the function f. In both cases, also logging transitions for the atom A are incorporated into Δ to express the necessary read access to A. Since A is non-ground, all variables of A are replaced by the special constant *all* before the logging transition is assigned. The truth value of the DB atoms in each hypothetical state is given by the possibly three-valued DB interpretation (I_{DB}). We adopt a cautious view, where an undefined truth value leads to a logical failure.

The connectives of the ULTRA language have several algebraic properties, which can be used when rewriting update formulas.

Proposition 3.9 [Algebraic Properties] The following properties hold for the connectives ",", ":", and $\vee$.

1. The concurrent conjunction "," is commutative (a) and associative (b) and has NOP as a neutral element (c).
2. The sequential conjunction ":" is associative (a) and has NOP as a neutral element (b).
3. The disjunction $\vee$ is commutative (a) and associative (b).

Proof: The assertions can be shown using the properties 3.2 required for the composition functions as well as the classical Boolean laws. A complete proof can be found in Appendix A. $\square$

Due to the associativity, we do not need to use precedence brackets in formulas of the form $\varphi_1, \ldots, \varphi_n$, $\varphi_1 : \ldots : \varphi_n$, and $\varphi_1 \vee \ldots \vee \varphi_n$.

3.3 Specific Semantics for Logic Databases

Now we develop the semantics for the database language presented in Section 2.2. We only have to fill the gaps left in the generic ULTRA framework, i.e. we have to define an appropriate transition system as well as the mappings I_{DB}, Log, and Upd which relate it to the syntactical elements of the database language. Despite some minor formal differences, the resulting semantics is exactly the same as given in [WFF98b].

In the context of logic databases, the states will be defined as the EDB instances, and the transitions will be defined as sets which contain insertion and deletion requests for EDB tuples. Such an update request can be considered as an assertion about the next database state. Update request sets store also read tags for EDB relations accessed during derivation and can thus be considered as local logs (cf. [BHG87, GR93]).

Definition 3.10 [Update Request] A *ground basic update request* has the form $+r(\vec{t})$ or $-r(\vec{t})$, where $r(\vec{t}) \in \mathcal{B}$ is a ground EDB atom. We sometimes refer to these update requests as database update requests. □

Intuitively, a ground basic update request specifies the insertion (+) or deletion (−) of an EDB atom, i.e. a tuple in an EDB relation.

Definition 3.11 [Read Tag] A *read tag* has the form $?r$, where r is an EDB predicate. Sometimes we refer to $?r$ as a database read tag. □

The read tags are used to record retrieval operations. Intuitively, a read tag $?r$ expresses that the EDB relation associated with r has (possibly) contributed to the result. Thus, to ensure transaction isolation, the read tags have to be certified [BHG87], before the computed update requests are actually materialized. The certification must check the absence of read/write conflicts with concurrent transactions (see [WFF98b, WFF98c] for details).

Definition 3.12 [Update Request Set, Consistency] An *update request set* Δ is a set of ground basic update requests and read tags.

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

Now we are able to define the required database-specific concepts in order to obtain a concrete interpretation of the update formulas.

Definition 3.13 [States] The set $\mathcal{S}$ of *(database) states* is defined as the set of all EDB instances over the Herbrand base $\mathcal{B}$. □

Definition 3.14 [Transitions] The set $\mathcal{T}$ of *(database) transitions* is defined as the set of all update request sets taken from the Herbrand base $\mathcal{B}$.

The subset $\mathcal{T}_{Cons} \subseteq \mathcal{T}$ is defined as the set of consistent update request sets according to Definition 3.12 ($\mathcal{T}_{Cons}$ equals the set formerly [WF97a, WFF98b] denoted by $\mathcal{D}$).

The *neutral* transition $\Delta_\varepsilon \in \mathcal{T}_{Cons}$ is defined as the empty set $\emptyset$, which is indeed a consistent update request set. □

Definition 3.15 [Execution] The *execution* $\oplus_E$ of a consistent update request set $\Delta \in \mathcal{T}_{Cons}$ w.r.t. the EDB instance $DB \in \mathcal{S}$ is defined by:

$$DB \oplus_E \Delta \quad := \quad \{r(\vec{t}) \mid (r(\vec{t}) \in DB \wedge -r(\vec{t}) \notin \Delta) \vee +r(\vec{t}) \in \Delta\}$$

□

Example 3.16 The execution of the update request set

$$\Delta := \{-entry(mon, 10, 0), +entry(mon, 10, 23)\}$$

w.r.t. the EDB instance DB_0 of Example 2.15 results in the new state:

$$\begin{array}{rcll} DB_0 \oplus_E \Delta & = & DB_0 & \cup \; \{entry(mon, 10, 23)\} \\ & & & \setminus \; \{entry(mon, 10, 0)\} \end{array}$$

□

In logic databases the semantics of the IDB atoms often is given by the *well-founded model* [vG89, vGRS91], which subsumes most standard models defined for restricted program classes, see e.g. [AB94]. It is denoted $WFM(\Phi)$ for a union $\Phi = P_{IDB} \cup DB$ of the (fixed) set of IDB rules P_{IDB} and an EDB instance DB.

Definition 3.17 [DB Interpretation] For logic databases, we define the *DB interpretation*

$$I_{DB} : \mathcal{S} \to \mathcal{I}^3_{\mathcal{B}}$$

by

$$I_{DB}(DB) := WFM(P_{IDB} \cup DB)$$

for all EDB instances $DB \in \mathcal{S}$. □

In our prototype implementation and even in the running example, we make use of stratified aggregation and extra-logical constructs (computed functions or predicates also called built-ins). This is not considered in the pure semantics, but can be integrated under certain conditions, see e.g. [vG92] for well-founded aggregation.

Definition 3.18 [Transition Assignments] We define the mapping *Log* by

$$Log(q(\vec{t})) := \{?r \mid r \in Def_E[P_{IDB}](q)\}$$

for all ground DB atoms $q(\vec{t}) \in \mathcal{B}$ and the mapping *Upd* by

$$\begin{array}{rcl} Upd(INS\ r(\vec{t})) & := & \{+r(\vec{t})\} \\ Upd(DEL\ r(\vec{t})) & := & \{-r(\vec{t})\} \end{array}$$

for all ground EDB atoms $r(\vec{t}) \in \mathcal{B}$. (In the database context, this defines *Upd* for all elements of $\mathcal{B}_{BU}$.) □

Finally, we must define the composition constructs for the transition system.

Definition 3.19 [Concurrent Composition] The *concurrent composition* $\sqcup$ of two update request sets $\Delta_1, \Delta_2 \in \mathcal{T}$ is defined by the set union, i.e.

$$\Delta_1 \sqcup \Delta_2 := \Delta_1 \cup \Delta_2.$$

Similarly, we define the concurrent composition $\bigsqcup$ for a multi-set of update request sets. □

Definition 3.20 [**Write-Compatibility**] Recall Definition 3.4. If two or more consistent update request sets are conforming with each other, we call them also *write-compatible* (cf. [WFF98b]). In Corollary 3.25 we will show that this is legitimate. □

Note that the write-compatibility is an intra-transaction compatibility which is necessary to define a clear semantics for the concurrent conjunction. Within a transaction, a read access does supposedly not conflict with an update. Update goals composed by the concurrent conjunction are evaluated w.r.t. the same current state and specify individual update request sets which are merged in order to express a simultaneous transition leading to a common next state.

Example 3.21 Consistent updates request sets are for example:

$$\begin{array}{rcl}\Delta_1 & = & \{?entry, -entry(mon, 10, 0), +entry(mon, 10, 23)\} \\ \Delta_2 & = & \{+description(23, \text{“Presentation”})\} \\ \Delta_3 & = & \{?entry, -entry(mon, 10, 0), +entry(mon, 10, 23), \\ & & \;+ description(23, \text{“Presentation”})\}\end{array}$$

Δ_3 is the concurrent composition of Δ_1 and Δ_2, i.e. $\Delta_3 = \Delta_1 \sqcup \Delta_2$. Δ_1 and Δ_2 are write-compatible, as Δ_3 is consistent (cf. Definition 3.4).

The following update request sets are consistent, but not conforming with each other:

$$\begin{array}{rcl}\Delta_4 & = & \{?entry, -entry(mon, 10, 0)\} \\ \Delta_5 & = & \{?entry, +entry(mon, 10, 0)\}\end{array}$$

The concurrent composition of Δ_4 and Δ_5 contains both the insertion and deletion of the fact $entry(mon, 10, 0)$ and thus is not a consistent update request set. □

Definition 3.22 [**Sequential Composition**] The *sequential composition* $\oplus$ of two update request sets $\Delta_1, \Delta_2 \in \mathcal{T}$ is defined by:

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

□

Proposition 3.23 Let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be write-compatible update request sets. Then the equality

$$\Delta_1 \oplus \Delta_2 = \Delta_1 \sqcup \Delta_2$$

holds.

Proof: See Appendix A. □

After having defined the specific concepts, we must verify that they satisfy the required algebraic properties of Definition 3.2. This will be done in the following theorem.

Theorem 3.24 [**Algebraic Properties**] The algebraic properties of Definition 3.2 hold for the transition system defined for the extended database language.

Proof: The assertion follows from the definitions of $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$ in this section. A complete proof can be found in Appendix A. □

Corollary 3.25 [Write-Compatibility] Let $DB \in \mathcal{S}$ be an EDB instance, and let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be write-compatible update request sets. Then the following equalities hold:

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

In other words, the execution order of Δ_1 and Δ_2 is not relevant for the resulting state.

Proof: The assertion follows directly from Proposition 3.23 and the algebraic properties proved in Theorem 3.24. □

In Theorem 3.24 we have shown that the concepts defined in this section are legal for the generic ULTRA framework. Thus, we can apply the semantical results of Section 3.2 and get a semantics for the specific database language. It is easy to see that the resulting semantics coincides with the semantics defined in [WFF98b]. Let us now turn to an illustration of the interpretation of update formulas as given in Definition 3.8.

Example 3.26 [Personal Calendar (Cont.)] Consider the update formula

$$\varphi :\equiv entry(mon, 10, 0),\ DEL\ entry(mon, 10, 0),\ INS\ entry(mon, 10, 23)$$

which is an instance of one of the rule bodies defining the update predicate *do_allocate* (see Appendix B). Let the EDB instance DB_0 of Example 2.15 be given as the initial state. By Definition 3.8, cases (DB) and (BU), the following holds regardless of any specific interpretation I_{UP} of definable update atoms:

$$\begin{array}{lcl} (\emptyset, \{?entry\}) & \in & I(entry(mon, 10, 0)) \\ (\emptyset, \{-entry(mon, 10, 0)\}) & \in & I(DEL\ entry(mon, 10, 0)) \\ (\emptyset, \{+entry(mon, 10, 23)\}) & \in & I(INS\ entry(mon, 10, 23)) \end{array}$$

Applying case (CCj) twice we can deduce:

$$(\emptyset, \{?entry, -entry(mon, 10, 0), +entry(mon, 10, 23)\}) \in I(\varphi)$$

Up to now, we only considered the initial state DB_0 as the current state. Next, we consider a hypothetical current state, where the fact $entry(mon, 10, 0)$ has been removed from the EDB:

$$(\{-entry(mon, 10, 0)\}, \{?entry, -entry(mon, 10, 0), +entry(mon, 10, 23)\}) \notin I(\varphi)$$

holds, because $entry(mon, 10, 0)$ is false in the hypothetical state and case (DB) of Definition 3.8 is thus not applicable. □

3.4 Specific Semantics for External Operations

In this section we will give a semantics for the ULTRA instance that is adequate for external operations (cf. Section 2.3). Note that this semantics is essentially restricted to sequential operations and just illustrates how to derive another instance of the ULTRA framework. For the sake of brevity the development of a refined semantics for external operations and a combination with the database world is omitted here. Like in Section 3.3 we have to define a transition system together with the mappings I_{DB}, *Log*, and *Upd*.

We assume that the set of states $\mathcal{S}$ and an execution function

$$do : \mathcal{B}_{BU} \times \mathcal{S} \to \mathcal{S}$$

for basic actions which are identified with the basic update atoms are externally defined. This models the behaviour of the external system. Let further the DB interpretation I_{DB} be given as the projection of the states onto the observable properties represented by the DB atoms.

do and I_{DB} can be axiomatized, e.g. by means of the situation calculus [Rei95]. In this case, hypothetical reasoning is an option for the operational semantics. However, an axiomatization is not necessary, if the external operations can be performed and possibly undone and if $I_{DB}(s)$ can be queried directly in every physical state s of the external system. The latter case is reasonable for our robot example.

Example 3.27 [**Robot World (Cont.)**] In our running example, the states in $\mathcal{S}$ will be the states of the robot arm and its environment, i.e. the blocks world.

If the robot world is currently in state $s \in \mathcal{S}$, then $do(pickup, s)$ denotes the state that results from executing the operation *pickup*. Of course $s = do(pickup, s)$ may hold, since idle operations are possible. But usually $do(pickup, s)$ denotes a state where a block which was on the floor in s has been picked up by the robot arm.

In our simplified example, the only observable item is the sensor of the robot arm. Assume that the robot arm is empty in state s and a block is lying on the floor at the current position, then the semantics I_{DB} of the sensor states that $I_{DB}(s) \models empty$ and $I_{DB}(do(pickup, s)) \models \neg empty$ holds. □

Based on $\mathcal{S}$ and *do* we now specify a transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ suitable to model the semantics of ULTRA formulas in the external world.

Definition 3.28 [**Transitions**] The set $\mathcal{T}_{Cons}$ of *consistent transitions* is defined as the set $\mathcal{B}^*_{BU}$ that consists of all finite lists of ground basic update atoms.

We define

$$\mathcal{T} := \mathcal{T}_{Cons} \cup \{\Delta_\perp\},$$

where $\Delta_\perp$ denotes a special inconsistent transition.

The *neutral* transition $\Delta_\varepsilon \in \mathcal{T}_{Cons}$ is defined as the empty list $[\,]$.

A multi-set T over $\mathcal{T}$ is called *neutral*, if it contains at most the neutral transition $[\,]$. □

Definition 3.29 [**Execution**] Let $\circ$ denote the classical list concatenation.

The *execution* $\oplus_E$ of a consistent transition $\Delta \in \mathcal{T}_{Cons}$ w.r.t. the external state $s \in \mathcal{S}$ is defined recursively by:

$$s \oplus_E \Delta := \begin{cases} do(u(\vec{t}), s) \oplus_E \Delta', & \text{if } \Delta = [u(\vec{t})] \circ \Delta' \text{ for a ground basic update atom } u(\vec{t}) \\ s, & \text{if } \Delta = [\,] \end{cases}$$

□

Definition 3.30 [**Concurrent Composition**] The *concurrent composition* $\sqcup$ of two transitions $\Delta_1, \Delta_2 \in \mathcal{T}$ is defined by:

$$\Delta_1 \sqcup \Delta_2 := \begin{cases} \Delta_1, & \text{if } \Delta_2 = [\,] \\ \Delta_2, & \text{if } \Delta_1 = [\,] \\ \Delta_\perp, & \text{otherwise} \end{cases}$$

In analogy, we define the concurrent composition $\bigsqcup$ for a multi-set T of transitions:

$$\bigsqcup T := \begin{cases} [\,], & \text{if } T \text{ neutral} \\ \Delta, & \text{if } T = \{\Delta\} \uplus T' \text{ for a neutral multi-set } T' \\ \Delta_\perp, & \text{otherwise} \end{cases}$$

□

Definition 3.31 [**Sequential Composition**] Let $\circ$ denote the classical list concatenation.

The *sequential composition* $\oplus$ of two transitions $\Delta_1, \Delta_2 \in \mathcal{T}$ is defined by:

$$\Delta_1 \oplus \Delta_2 := \begin{cases} \Delta_1 \circ \Delta_2, & \text{if } \Delta_1 \neq \Delta_\perp \text{ and } \Delta_2 \neq \Delta_\perp \\ \Delta_\perp, & \text{otherwise} \end{cases}$$

□

Obviously, the functions $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$ are well-defined w.r.t. recursion and case distinction. $\oplus_E$ models the subsequent execution of the collected basic actions, and $\oplus$ is essentially defined as the list concatenation. The operations $\sqcup$ and $\bigsqcup$ need some further explanation. In our simplified model, we assume that the concurrent composition is allowed, i.e. consistent, if at most one of the composed transitions is not equal to $[\,]$. Otherwise we consider it as inconsistent. Informally speaking, we disallow simultaneous invocations of external actions. This implicitly requires to compose update goals using only the sequential conjunction. However, as we will define below, the DB atoms do not produce any logging transitions different from $[\,]$. Consequently, the concurrent conjunction of DB literals and other update atoms may yield valid results. The bulk quantification, however, is rather useless, as it is designed for the concurrent composition of multiple update transitions. Recall that this instantiation of the ULTRA framework only serves for illustration purposes. A full-blown instantiation that allows a combination of parallel and sequential tasks needs more complex transition structures and composition functions. It is left out in this paper due to space constraints.

Let us finally define the missing transition assignments.

Definition 3.32 [**Transition Assignments**] We define the mapping *Log* by

$$Log(q(\vec{t})) := [\,]$$

for all ground DB atoms $q(\vec{t}) \in \mathcal{B}$ and the mapping *Upd* by

$$Upd(u(\vec{t})) := [u(\vec{t})]$$

for the ground basic update atoms $u(\vec{t}) \in \mathcal{B}_{BU}$. □

After having defined the specific objects, we must verify the algebraic properties required by Definition 3.2. This will be done in the following theorem.

Theorem 3.33 [**Algebraic Properties**] The algebraic properties of Definition 3.2 hold for the transition system defined for external operations.

Proof: The assertion follows from the definitions of $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$ in this section. The proof is omitted in this paper. □

By Theorem 3.33 we have shown that the objects defined in this section are legal for the generic ULTRA framework. Let us illustrate the meaning of some goals in the context of our running example.

Example 3.34 Consider the update formula

$$\varphi :\equiv [\ empty,\ move(p_3)\ ]\ :\ pickup\ :\ NOT\ empty$$

which is an instance of the rule body defining the update predicate *pickup_at_position* (see Example 2.18 or Appendix C). Let the robot world be in a state s_0, where the robot arm is at position p_1 and empty, and where one block is lying on the floor at position p_3. We want to interpret the goal φ w.r.t. the initial state s_0.

We define the following auxiliary states:

$$\begin{array}{lcllcl} s_1 & := & s_0 \oplus_E [move(p_3)] & = & do(move(p_3), s_0) \\ s_2 & := & s_0 \oplus_E [move(p_3), pickup] & = & do(pickup, s_1) \end{array}$$

By Definition 3.8, cases (DB) and (BU), the following holds regardless of any specific interpretation I_{UP} of definable update atoms:

$$\begin{array}{lcl} ([\,],[\,]) & \in & I(empty) \\ ([\,],[move(p_3)]) & \in & I(move(p_3)) \\ ([move(p_3)],[pickup]) & \in & I(pickup)) \\ ([move(p_3), pickup],[\,]) & \in & I(NOT\ empty) \end{array}$$

Note that the first assertion and the second assertion refer to the initial state s_0, where *empty* is true. The other assertions refer to the hypothetical states s_1 and s_2, which are semantically represented by $[move(p_3)]$ and $[move(p_3), pickup]$, respectively. As the block at position p_3 has been picked up in state s_2 the truth value of the DB atom *empty* has become false, i.e. $I_{DB}(s_2) \models \neg empty$.

Applying case (CCj) once and case (SCj) twice we can deduce:

$$([\,],[move(p_3), pickup]) \in I(\varphi)$$

Intuitively, this result states that whenever the robot world is in state s_0 (represented by $[\,]$), the formula φ has a solution with a side effect expressed by the action list $[move(p_3), pickup]$. □

3.5 Semantics of Update Programs

Let an update program P_{UP} be given, and let $s_0 \in \mathcal{S}$ be an arbitrary but fixed initial state. In this section we characterize the (minimal) models of P_{UP}.

As stated in Section 3.1, the interpretation of DB atoms and basic update atoms is given from outside (by I_{DB}, *Log*, and *Upd*). So we just have to find an interpretation I_{UP} of the definable update atoms that respects the intended semantics of P_{UP}. Recall that in this setting, an interpretation I of the ground update formulas is completely determined by I_{UP}. We will therefore identify I_{UP} and the corresponding interpretation I of ground update formulas and also write $I_{UP}(\varphi)$ for arbitrary ground update formulas φ. Note that $I_{UP}(\varphi)$ may depend on s_0.

Definition 3.35 [Models of an Update Program] An interpretation I_{UP} of definable update atoms is a *model* of an update program P_{UP}, iff for each rule $r \in P_{UP}$

$$I_{UP}(r) = \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$$

holds. Recall that rules denote universally closed implications. □

An interpretation I_{UP} is a model of an update program P_{UP}, iff every rule of P_{UP} is true w.r.t. I_{UP}. According to Definition 3.8 this holds, iff $I_{UP}(\varphi) \subseteq I_{UP}(\psi)$ holds for every ground instance $\varphi \to \psi$ of a rule in P_{UP}. Informally speaking, this means that each rule head allows at least all state changes specified by the corresponding rule body. The formal definition states that every rule must be interpreted as valid w.r.t. arbitrary pairs of transitions, where each pair corresponds to a state change (cf. Section 3.2).

In the following we have to deal with various interpretations I_{UP} over the given definable update base $\mathcal{B}_{DU}$.

Definition 3.36 $\mathcal{I}$ is defined as the set of all interpretations $I : \mathcal{B}_{DU} \to 2^{\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}}$ (over the definable update base $\mathcal{B}_{DU}$). Further define $I_\perp, I_\top : \mathcal{B}_{DU} \to 2^{\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}}$ by

$$I_\perp(p(\vec{t})) := \emptyset \quad \text{and} \quad I_\top(p(\vec{t})) := \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. □

Definition 3.37 [Interpretation Ordering] The ordering relation $\leq$ on interpretations is defined by:

$$I_1 \leq I_2 \quad :\Longleftrightarrow \quad I_1(p(\vec{t})) \subseteq I_2(p(\vec{t})) \text{ for all } p(\vec{t}) \in \mathcal{B}_{DU}$$

□

Intuitively, $I_1 \leq I_2$ holds, if I_2 allows at least the same state changes for a definable update atom as I_1. However, I_1 may be more restrictive.

Lemma 3.38 [Lattice of Interpretations] The set $\mathcal{I}$ of all interpretations together with its ordering relation $\leq$ (cf. Definition 3.37) forms a complete lattice.

The bottom element and the top element of $\mathcal{I}$ are the constant mappings $I_\perp$ and $I_\top$, respectively.

Proof: The assertions follow from similar properties of the power-set lattice. □

Remark 3.39 [Trivial (Maximal) Model] Every update program P_{UP} has at least one trivial model, i.e. the maximal interpretation $I_\top$.

Proof: The assertion follows directly from Definitions 3.8 and 3.35. □

Definition 3.40 [Minimal Model] A model $M \in \mathcal{I}$ is *minimal*, iff

$$M' \leq M \Longrightarrow M' = M$$

holds for any other model $M' \in \mathcal{I}$. □

Lemma 3.41 Let $\mathcal{J} \subseteq \mathcal{I}$ be a non-empty set of interpretations over $\mathcal{B}_{DU}$. Define a new interpretation I by

$$I(p(\vec{t})) := \bigcap_{I' \in \mathcal{J}} I'(p(\vec{t}))$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. Then

$$I(U) \subseteq \bigcap_{I' \in \mathcal{J}} I'(U)$$

holds for every ground update goal U.

Proof: The assertion can be proved by structural induction. See Appendix A for details. □

Lemma 3.42 [Model Intersection Property] Let P_{UP} be an update program, let $\mathcal{M} \subseteq \mathcal{I}$ be a non-empty set of models of P_{UP}. Define a new interpretation M by

$$M(p(\vec{t})) := \bigcap_{M' \in \mathcal{M}} M'(p(\vec{t}))$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. Then M is a model of P_{UP}.

Proof: The assertion follows easily from Lemma 3.41 (see Appendix A). □

Theorem 3.43 [Existence of a Unique Minimal Model] Let P_{UP} be an update program. Then P_{UP} has a unique minimal model M_{UP}.

Proof: P_{UP} has at least one model due to Remark 3.39. Let $\mathcal{M}$ be the non-empty set of all models of P_{UP}. Define M_{UP} by

$$M_{UP}(p(\vec{t})) := \bigcap_{M' \in \mathcal{M}} M'(p(\vec{t}))$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. By Lemma 3.42 M_{UP} is a model of P_{UP}. Due to the definition of M_{UP}, $M_{UP} \leq M'$ holds for all models $M' \in \mathcal{M}$. The minimality and uniqueness of M_{UP} follow directly. □

Now we will have a look at the minimal model of the update programs in our running examples. Recall Theorems 3.24 and 3.33, which allow us to use all results proved for the generic ULTRA language also for the database language and the programming language for external operations.

Example 3.44 [Personal Calendar (Cont.)] Consider the update rules

$$\begin{array}{lcl} \mathit{do\_allocate}(D,S,1,ID) & \leftarrow & \mathit{entry}(D,S,0), \\ & & DEL\ \mathit{entry}(D,S,0),\ INS\ \mathit{entry}(D,S,ID) \\ \mathit{do\_allocate}(D,S,L,ID) & \leftarrow & L > 1,\ \mathit{do\_allocate}(D,S,1,ID), \\ & & S1 = S+1,\ L1 = L-1, \\ & & \mathit{do\_allocate}(D,S1,L1,ID) \end{array}$$

of our calendar example (see Appendix B) and recall Example 3.26, in particular the definition:

$$\varphi :\equiv \mathit{entry}(mon,10,0),\ DEL\ \mathit{entry}(mon,10,0),\ INS\ \mathit{entry}(mon,10,23)$$

Because

$$(\emptyset, \{?\mathit{entry}, -\mathit{entry}(mon,10,0), +\mathit{entry}(mon,10,23)\}) \in I(\varphi)$$

holds for arbitrary interpretations $I \in \mathcal{I}$ (see Example 3.26), it follows that

$$\begin{array}{l} (\emptyset, \{?\mathit{entry}, -\mathit{entry}(mon,10,0), +\mathit{entry}(mon,10,23)\}) \\ \quad \in M(\mathit{do\_allocate}(mon,10,1,23)) \end{array}$$

holds for every model M of P_{UP}. Finally, have a look at the following negative example. Although there are many models M of P_{UP} for which e.g.

$$\begin{array}{l} (\{-\mathit{entry}(mon,10,0)\}, \{?\mathit{entry}, -\mathit{entry}(mon,10,0), +\mathit{entry}(mon,10,23)\}) \\ \quad \in M(\mathit{do\_allocate}(mon,10,1,23)) \end{array}$$

holds, it does not hold for the minimal model M_{UP}, because it is not justified by any rule body (see Example 3.26). The statement refers to a state change for the goal $\mathit{do\_allocate}(mon,10,1,23)$ which is not specified by the programmer and thus is excluded by the minimality condition. □

Example 3.45 [Robot World (Cont.)] Consider the update rule

$$\mathit{pickup\_at\_position}(P) \quad \leftarrow [\ \mathit{empty},\ \mathit{move}(P)\ ]\ :\ \mathit{pickup}\ :\ NOT\ \mathit{empty}$$

of our robot world example (see Example 2.18 and Appendix C) and recall Example 3.34, in particular the definition:

$$\varphi :\equiv [\ \mathit{empty},\ \mathit{move}(p_3)\ ]\ :\ \mathit{pickup}\ :\ NOT\ \mathit{empty}$$

Because

$$([\,], [move(p_3), pickup]) \in I(\varphi)$$

holds for arbitrary interpretations $I \in \mathcal{I}$, it follows that

$$([\,], [move(p_3), pickup]) \in M(pickup\_at\_position(p_3))$$

holds for every model M of P_{UP}, in particular for the minimal model M_{UP}. □

We now provide an inductive fixpoint characterization of the unique minimal model of a given update program. The well-known results for definite programs [Llo87] essentially hold for the update programs in ULTRA, too.

Definition 3.46 [Immediate Consequence Operator] Let P_{UP} be an update program. We define an *immediate consequence operator* $T_{P_{UP}} : \mathcal{I} \to \mathcal{I}$ on the set of all interpretations over $\mathcal{B}_{DU}$ as follows: let $I \in \mathcal{I}$ be an arbitrary interpretation, then define

$$\begin{array}{rcl} T_{P_{UP}}(I)(p(\vec{t})) & := & \{ \;\; (\Delta_C, \Delta) \in \mathcal{T}_{Cons} \times \mathcal{T}_{Cons} \mid \\ & & \quad\;\; \text{there exists a ground instance } U \to p(\vec{t}) \\ & & \quad\;\; \text{of a rule } r \in P_{UP}, \text{ such that } (\Delta_C, \Delta) \in I(U) \;\; \} \end{array}$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. □

To be able to apply the fixpoint theory we must show the monotonicity of the immediate consequence operator.

Lemma 3.47 [Monotonicity] The immediate consequence operator $T_{P_{UP}}$ of Definition 3.46 is monotonic.

Proof: The proof is straight-forward: let two interpretations $I_1, I_2 \in \mathcal{I}$ with $I_1 \leq I_2$ be given. Lemma 3.41 applied to the set $\{I_1, I_2\} \subseteq \mathcal{I}$ can serve as an auxiliary to show that $I_1(U) \subseteq I_2(U)$ holds for arbitrary ground goals U, in particular for all instances of the rule bodies of P_{UP}. $T_{P_{UP}}(I_1) \leq T_{P_{UP}}(I_2)$ follows directly from Definition 3.46. □

Lemma 3.48 Let P_{UP} be an update program and $I \in \mathcal{I}$ be an arbitrary update interpretation. Then the following holds:

$$I \text{ is a model of } P_{UP} \iff T_{P_{UP}}(I) \leq I$$

Proof: The assertion follows directly from Definitions 3.35 and 3.46. □

Definition 3.49 The *ordinal powers* $T_{P_{UP}} \uparrow \gamma$ of the immediate consequence operator of an update program P_{UP} are defined as usual (see [Llo87]). □

Theorem 3.50 [Fixpoint Characterization] Let P_{UP} be an update program and M_{UP} be its unique minimal model. Then M_{UP} is the least fixpoint of $T_{P_{UP}}$, and there exists a closure ordinal $\alpha_{P_{UP}}$, such that $M_{UP} = T_{P_{UP}} \uparrow \alpha_{P_{UP}}$.

Proof: Using the properties of the power-set lattice, one can show that

$$glb(\mathcal{J})(p(\vec{t})) = \bigcap_{I \in \mathcal{J}} I(p(\vec{t}))$$

holds for all $p(\vec{t}) \in \mathcal{B}_{DU}$ and arbitrary non-empty subsets $\mathcal{J}$ of the lattice $\mathcal{I}$ of interpretations.

Now recall the construction of M_{UP} and apply Lemma 3.48. We have:

$$M_{UP} = glb \; \{ \; I \in \mathcal{I} \mid T_{P_{UP}}(I) \leq I \; \}$$

The assertion follows from the theorem of Knaster and Tarski (see [Llo87] for details). Note that the monotonicity property of Lemma 3.47 is essential. □

Remark 3.51 If the Herbrand universe $\mathcal{U}$ is finite or P_{UP} uses bulk quantification exclusively at the level of basic updates (e.g. in the fashion $\#\vec{X}\,[A \mapsto u(\vec{t})]$, where u is a basic update predicate (compare to updates in SQL)), it is possible to show that the immediate consequence operator $T_{P_{UP}}$ is continuous. In this case, $\alpha_{P_{UP}} \leq \omega$ holds [Llo87], i.e. the minimal model of P_{UP} can be generated by the iterated application of $T_{P_{UP}}$ at most to the first limit ordinal. In the general case, however, $T_{P_{UP}}$ may be not continuous. □

Definition 3.52 Let P_{UP} be an update program, and let $s_0 \in \mathcal{S}$ be the fixed initial state. Then

$$M_{UP}[P_{UP}, s_0]$$

denotes the (unique) minimal model of P_{UP} w.r.t. the state s_0. □

4 Transactional Execution of ULTRA Operations

Complex operations specified in the ULTRA language should be performed as transactions. The ACID properties, which are well-known in database theory [BHG87, BN97, GR93], play an important role to guarantee a well-defined and consistent behaviour – even in case of concurrency or failures. In the ULTRA context, a new top-level transaction is invoked by a top-level update query. The interpretation of the query goal yields possible transitions, which can transform the initial state s_0 to a desired future state.

Definition 4.1 [Possible Transitions] Let P_{UP} be an update program and let $s_0 \in \mathcal{S}$ be the fixed initial state. Let Q be an update query of the form $\leftarrow U$, where U does not contain any free variables.

All transitions $\Delta \in \mathcal{T}_{Cons}$ such that $(\Delta_\varepsilon, \Delta) \in M_{UP}[P_{UP}, s_0](U)$ are called *possible transitions* for the update query Q. □

To commit a transaction invoked by a top-level query, one possible transition of the query goal has to be materialized, such that a new physical state $s_0' \in \mathcal{S}$ will be reached. The logical semantics does not care about when and how the choice of a possible transition will take place. This is left to the operational semantics. We only require that whenever a transaction commits, the physical execution of a possible transition must be complete. However, a transaction may be aborted, but then it must not cause a (persistent) state change. A transaction will logically fail, if there exists no possible transition.

Example 4.2 [Personal Calendar (Cont.)] We provide a predicate $do\_insert\_on\_day(D, L, T)$ which can be used to insert an appointment having a description T and a duration of L slots on a day D without specifying the starting time. Instead, all those time slots an appointment of the requested duration can be assigned to are looked up, then the corresponding database entry is inserted at a free position. The defining rule reads as follows:

$$do\_insert\_on\_day(D, L, T) \quad \leftarrow \quad free(D, S, L) \;:\; do\_insert(D, S, L, T)$$

Now suppose the update query

$$Q \quad \equiv \quad \leftarrow \textit{do\_insert\_on\_day}(mon, 1, \text{“Call Mr. Martin”})$$

is issued in database state DB_0 of Example 2.15. Using the IDB predicate $free$ (see Appendix B), it is deduced that the time slots on Monday at 10am, 11am, or 2pm can hold an appointment of one hour. Therefore, the following three possible transitions for query Q can be generated, assuming the assigned identifier is 28:

$$\begin{aligned}
\Delta^{(1)} &= \{?entry, -entry(mon, 10, 0), +entry(mon, 10, 28), \\
&\quad + description(28, \text{“Call Mr. Martin”})\} \\
\Delta^{(2)} &= \{?entry, -entry(mon, 11, 0), +entry(mon, 11, 28), \\
&\quad + description(28, \text{“Call Mr. Martin”})\} \\
\Delta^{(3)} &= \{?entry, -entry(mon, 14, 0), +entry(mon, 14, 28), \\
&\quad + description(28, \text{“Call Mr. Martin”})\}
\end{aligned}$$

□

Example 4.3 [Robot World (Cont.)] Recall Example 3.45. Let the goal $\textit{pickup\_at\_position}(p_3)$ be submitted as a query, while the robot world is in the same state s_0 as described above. Then there exists one possible transition:

$$\Delta = [move(p_3), pickup]$$

If the robot world is in a state where the robot arm is not empty or there is no block lying at position p_3, the same query will have no possible transition. □

As free variables in DATALOG queries usually have a set-oriented interpretation, which is different from the provided non-deterministic update semantics, we do not allow free variables in top-level update queries. This seems a little odd at a first glance, because in conventional database applications pure retrieval transactions are of course also necessary. However, the latter can be easily integrated into the overall approach provided they produce update request sets which contain read tags according to the queried information.

Definition 4.4 [Retrieval Queries] Let Q be a retrieval query of the form $\leftarrow\ q(\vec{t})$ about a DB atom $q(\vec{t})$ which may contain the variables $\vec{X}$. Then we assign to Q the "possible transition" $\Delta := Log(q(\vec{t})[\vec{X}\,/\,\vec{all}\,])$. □

Like update transactions, a pure retrieval transaction also has to materialize its (unique) possible transition. However, in every ULTRA instance for conventional databases, this materialization does not perform any visible updates on the state. Instead, only isolation checks with other transactions will be done (see [WFF98b, WFF98c] for details). These checks can be seen as part of the materialization.

Example 4.5 [Personal Calendar (Cont.)] $browse(D, S, T)$ (see Appendix B) is an IDB predicate that relates a description T to the appointment on day D occupying time slot S. *browse* is defined as the natural join of the *entry* and the *description* relation. The following retrieval query Q inquires the agenda for entries on Monday at 12pm:

$$Q \quad \equiv \quad \leftarrow browse(mon, 12, T)$$

This retrieval query produces the binding "Meeting Mr. Dean" for the free variable T. In addition, the update request set

$$\Delta = \{?entry,\ ?description\}$$

is generated. Both read tags contained in Δ have to be checked before the query result is returned to the top-level. □

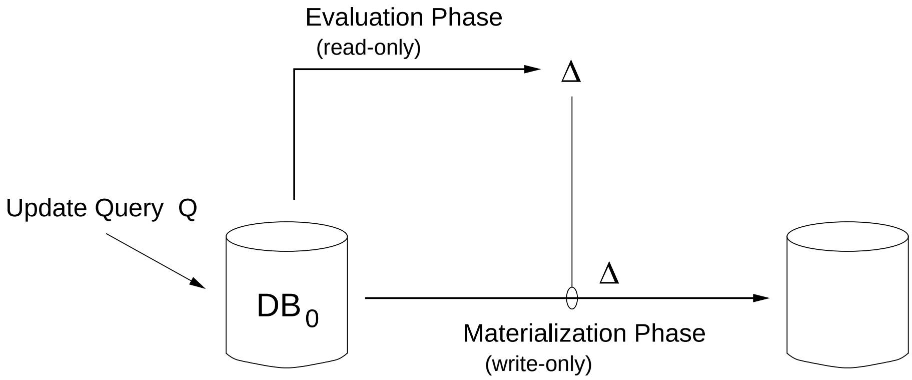

Figure 2: Evaluation and materialization phase

A straight-forward method to execute ULTRA transactions is the following two-phase strategy. The transactions invoked by top-level update queries are processed in two phases: an *evaluation phase*, where no updates are performed on the database, followed by a *materialization phase*. Figure 2 illustrates this operational semantics. As the initial state is kept unchanged during the evaluation phase, all references to intermediate states must be handled by hypothetical reasoning. This requires an axiomatization of observable facts and basic operations. The ULTRA instance tailored to logic databases is a suitable candidate for the two-phase strategy, while the robot interface is not, unless the robot world is axiomatized. The two-phase method can easily guarantee the atomicity and durability properties. Additionally, isolation properties can be guaranteed as shown in [WFF98b, WFF98c]: The read tags – or, more general, the logging transitions – must be considered, before the execution of the computed basic actions may start. The system must check that the read access during the evaluation phase has not been invalidated by updates of concurrent transactions. The overall execution method corresponds to an optimistic transaction protocol [BHG87] similar to ODL [HD91].

A prototype system for the database-oriented instance of ULTRA has been implemented on top of the deductive database system *LOLA* [ZF97]. As the transaction processing is not integrated into the *LOLA* deduction engine, the *LOLA* system can be easily replaced by other systems, e.g. XSB [RSS+97]. Recall that within the two-phase approach the evaluation w.r.t. intermediate states, that are not materialized yet, is done by hypothetical reasoning. The insertion/deletion model allows this in an efficient way due to the algebraic properties of the update request sets. Technically, the reference to an intermediate state is handled similarly as in the situation calculus [Rei95]. The calculation of new transitions is explicitly encoded into the evaluable program. A compiler, which is called before query processing time, performs the adequate transformation of the IDB program P_{IDB} and the update program P_{UP} into a common logic program that can be evaluated by the *LOLA* engine.

5 Related Work

In the following subsections we discuss some work related to the ULTRA project. First, we cite a collection of concepts for rule-based update specification and other concepts dealing with database dynamics. A brief, informal description and classification is provided for each approach. The second subsection is devoted to the standard database language SQL and its procedural extensions. In the last subsection we refer to monads in functional programming languages and mention some similarities with the ULTRA approach.

5.1 Logical Foundations of Updates

Transaction Logic [BK94] is one of the concepts which have significantly inspired our own work. It forms a logical framework for the representation of updates in which the sequential composition of updates is handled explicitly and arbitrary basic updates can be integrated. Transaction Logic has no explicit construct for bulk updates. Of course, the effect of bulk updates can be obtained using recursive rules (see e.g. [MW88b] for an example). In [BKC93] a relational assignment operator for copying an IDB relation into the EDB is proposed. However, this operation is atomic and outside the scope of Transaction Logic. Thus, bulk updates cannot be composed from existing single updates. The general concept of Transaction Logic can also be used for reasoning about updates and update programs, see [BK98a] for details. Concurrent Transaction Logic [BK96] provides an explicit concurrency construct, by which one can specify that subtransactions are to be performed in an interleaved fashion. This form of concurrency, which is also found in various programming languages, e.g. Java, leads to verification problems for composite systems. In particular, modular programming becomes difficult because of the unconstrained interaction of the components. Although the (Concurrent) Transaction Logic approach can be considered as declarative, the evaluable programs have a similar semantics as programs written in classical, imperative programming languages. In contrast, ULTRA contains more abstract constructs and a novel semantics for concurrent updates that supports compositionality. While the update semantics of Transaction Logic is tightly defined in terms of state paths, the ULTRA framework leaves out the exact structure of the transition objects. Thus, it is possible to design and tune an instance w.r.t. the given environment and operational issues. Under some minor restrictions, the sequential version of Transaction Logic and the ULTRA instance presented in Sections 2.3 and 3.4 have the same modeling power.

U-Datalog [MBM97] has made the first attempt to perform bulk updates by aggregation of success paths in the resolution tree of a query. However, this approach cannot handle sequential composition or other operations than insertion and deletion of EDB tuples. Sequential updates can only be specified as complex transactions at the top-level. Due to a rigorous aggregation, one always gets a bulk update effect and cannot specify update alternatives as in Transaction Logic and ULTRA. In addition, the aggregation of the update requests does not distinguish between logical conjunction and disjunction.

The expressibility of the language DLP proposed by Manchanda and Warren [MW88b] is comparable to that of the rule fragment of Transaction Logic restricted to basic insertions and deletions. Like in our approach, the implementation uses a non-immediate update semantics. The formal semantics, however, is based on a dynamic logic in which a formula is interpreted by pairs of database states. Consequently, the same problems in specifying bulk updates arise as in Transaction Logic [BK94], and concurrent conjunction cannot be specified. Cronau [Cro90] has collected some methods for transaction processing in a DLP-like update language.

Chen defines an update calculus and a corresponding update algebra [Che95]. The latter is an extension of the relational algebra and – similarly to our approach – deals with increments, concurrent/sequential composition, and consistency constraints. In contrast to our work, the update calculus is based on abduction and minimal changes of a relational database, where the basic update atoms are considered as assertions for the next state. It should be noted that neither IDB rules (or relational views), nor update rules are considered in Chen's update calculus. But Chen has also worked on the integration of deferred updates with the logic programming paradigm and developed a concept [Che97] that has turned out to be similar to the ULTRA language described in [WFF98b] as well as in Sections 2.2 and 3.3 of this paper. However, Chen's main goal is to define a "well-founded semantics" for update programs by tailoring van Gelders's alternating fixpoint procedure [vG89] to operate on a structure built over update request sets, whereas in the ULTRA approach, the emphasis lies in the integration of arbitrary basic operations and in a transactional foundation. Up to now we have not considered to permit negation of basic or definable update atoms.

An alternative rule paradigm considers rules as condition-action specifications. Basic updates in a rule head are triggered, when the condition of the rule body is satisfied. There exist many different semantics depending on how the rules (set-oriented/tuple-oriented, deterministically/non-deterministically, simultaneously/successively, etc.) are evaluated, see e.g. [AV88]. Some approaches like Statelog [LHL95] and the concept of Zaniolo [Zan93] define a clear semantics based on temporal logic programming [AM89] by incorporating states into the rule formalism. Ludäscher, May, and Lausen [LML96] present an extension of Statelog by update procedures and sequential composition. The language further abstracts from states. In essence, its semantics encodes a top-down control into a bottom-up, data/event-driven evaluation framework. A similar effect can be obtained in the ULTRA context by using a suitable magic set transformation [BR91].

Kakas, Mancarella [KM90] and Bry [Bry90] use abductive reasoning for translating intensional updates into changes on the EDB. They do not explicitly describe update methods but derive them from view definitions and integrity constraints. In contrast, we use special rules which *explicitly* describe the update operations, although their evaluation has an abductive flavour. As proposed in [MW88b], suitable rules for performing view updates can be generated by a precompiler. In this setting, the possibility of expressing non-deterministic update operations is useful. In [ALS98] a logic database consists of IDB rules and update rules. The latter rules do not express updates, but are used to infer positive and negative elementary conclusions which must not be inconsistent. The update rules are also abductively used to trigger updates for maintaining the consistency. To get a deterministic semantics in this case, each body of an update rule must not contain more than one literal. For the IDB a well-founded model approach [vGRS91] is adopted.

Some other approaches dealing with updates in deductive databases should be mentioned: Naish, Thom, and Ramamohanarao [NTR87] show the problems that arise from non-declarative update constructs in logic programming and propose a clean concept based on deferred updates. Naqvi and Krishnamurthy [NK88] develop another rule-based update language, whose semantics unfortunately appears to be difficult and imposes many impractical constraints for the programmer.

So far, we have presented related work dealing with the logic-based *specification* of updates and transactions rather than with *reasoning* about actions. Consequently, the frame problem and other problems that arise at the axiomatization of effects are not an issue, neither in ULTRA, nor in the other approaches. Nevertheless, reasoning about actions becomes relevant for the operational semantics sketched in Section 4. In this case, the following concepts can serve as a basis. Reiter [Rei95] describes possible actions and its effects in the situation calculus and addresses the frame problem. In the language GOLOG [LRL+97], procedural structures and complex operations are introduced, their semantics is described by macro expansion and second order constructs. Like in our approach, all states generated by a sequence of actions are represented by an increment w.r.t. a certain initial state. The procedures that lead to the state transitions, however, are not specified in a logic language. Kowalski [Kow92] uses the event calculus for characterizing dynamics, which has some similarities with the situation calculus but behaves better for hypothetical reasoning. In [SWM93] a dynamic logic for verifying database updates is developed. However, the updates are programmed in a language like Embedded SQL.

For a more detailed comparison of work on the field of updates and change, we refer the reader to a recent survey article by Bonner and Kifer [BK98b].

5.2 The Database Language SQL

Today, most database management systems provide a variant of SQL as a data definition and data manipulation language (see [DD97] for a detailed description of the SQL/92 standard). The retrieval capabilities, i.e. the SELECT operation, together with the possibility to define views yield a programming environment similar to a classical deductive database system, although the SQL/92 standard does not allow recursive views and nested term structures as known from deductive databases. On the other hand, however, SQL features some “impure” constructs like sorting, committed choice, and constraints.

Simple update operations combining retrieval and tuple-oriented manipulations on base tables are directly expressible in SQL by the statements INSERT, DELETE, and UPDATE. Such updates implicitly behave set-oriented and can be simulated by ULTRA goals using the (explicit) bulk quantification.

Complex database operations are usually programmed in a host language accessing an SQL interface. This technique is called Embedded SQL. Nevertheless, several database systems and even the SQL/92 standard consider procedural extensions of the basic language. As an interesting feature, bulk updates can be implemented by (implicit) iteration over a cursor. This syntactically resembles the bulk quantification of ULTRA, although it does not have a declarative semantics. The procedural extensions usually allow to store program objects inside the database (Persistent Stored Modules/Procedures), which may facilitate availability and security issues. Transaction commands, however, must be explicit. A procedure itself does not describe a transaction, and the begin of its execution does not force the creation of a rollback sphere. In contrast, ULTRA implicitly considers complex operations as transactions, the programmer does not need to use transaction commands. Transaction handling is part of the evaluation process and must be performed in harmony with the modularization constructs. The latter is difficult in SQL, because hierarchical or even recursive programs do not fit well with the flat transaction model that is commonly provided.

The extensions of SQL form powerful languages, whose semantics, however, is mostly defined at the operational level, and specific extensions may have different implementations for the same constructs. The partly verbose and partly implicit structure of SQL is quite unsuitable for a formal treatment.

5.3 Functional Programming and Monads

In the area of functional programming, a powerful programming paradigm, called *monadic programming*, has been developed. As sketched in [Wad95a], it is easy to build a framework founded on an abstract data type. An instance of the framework can be created by providing a concrete implementation for the abstract data type. The data type must satisfy a few algebraic laws, such that it behaves as a *monad*. Note that this form of instantiation enables modular and polymorphic programming. The principles of the monadic programming technique have been implicitly adopted during the development of the generic ULTRA concept.

Monadic programming can even be exploited to enrich functional programming languages by “impure” constructs, in particular by I/O operations, without losing the declarative semantics of the programs, i.e. for a given program, it is possible to find a meaning that does not depend on the evaluation strategy. In [Wad95b] Wadler describes a useful I/O monad. The semantics of an I/O function can be compared with a function computing a list of basic I/O operations, which in turn realize the desired side effect, if they are executed on the I/O system. This approach is similar to the ULTRA semantics, which is also (semantically) based on deferred updates. Monadic programming together with a sophisticated lazy evaluation strategy can lead to immediate I/O, i.e. to an interleaving of evaluation and I/O. Such a strategy has been implemented for the language Haskell [JWA+92]. The goal-oriented transaction processing strategy described in [FWF00] shows strong similarities and allows for immediate updates in the ULTRA context. Consequently, we think that some results developed in the field of functional programming can be adopted for the ULTRA concept, and vice versa. It should be mentioned that in functional programming, mainly the higher-order features have a significant impact on the theoretical and operational semantics, whereas in logic programming, several problems which result from non-determinism and logical failure have to be solved.

6 Conclusion

In this paper we have presented the syntax and logical semantics of the rule-based update language ULTRA. We have described ULTRA as a generic concept which can serve as a framework for various transaction specification languages. To illustrate this framework character, we have reconsidered the extended database language of [WFF98b] as an instance of the framework. Further, we have presented another instance tailored to arbitrary external operations. This instance has been illustrated by a robot world example, which is significantly different from a database example. Although operational aspects are outside the scope of this paper, we have sketched a transaction processing strategy that works in two independent phases – *evaluation* and *materialization*. This strategy integrates smoothly with the logical semantics and is applicable at least for database-oriented instances of the framework.

One topic of ongoing work is the development of other instances of the ULTRA framework which are suitable for arbitrary (external) basic actions and which provide transactional features at the same time. The main task lies in the development of implementable structures that meet the algebraic properties of Definition 3.2, such that the model-theoretic semantics presented in this paper can be applied. The representation of transitions by sets of basic update requests turns out to be limited, since – as opposed to the simple insertion/deletion model – semantical properties like idempotence and commutativity cannot be assumed for the basic operations. The representation by lists suffers from algebraic properties needed for a non-trivial concurrent conjunction and for bulk updates. We suppose that transition systems built on the concept of partially ordered multi-sets of basic operations [Pra86] will provide the desired modeling power. However, a lot of technical details concerning consistency, infinity, execution semantics, etc. have to be investigated first and the formal treatment will need ample space.

To improve performance in the database context and to enable external actions without axiomatizations, a dedicated operational semantics based on immediate updates with an undo option is being developed [FWF00, WFF98a]. The operational semantics and the forthcoming system architecture is based on an open nested transaction model [BBG89, Mos85, WS92]. We do not exclude a combination of both evaluation paradigms studied so far. The deferred materialization combined with hypothetical reasoning could be adequate to deal with internal (database) states, while the immediate materialization might be necessary for the interaction with the external world. Of course, a finer-grained tuning of the operational model in combination with a meta-language is imaginable. Recall that the model-theoretic semantics presented in this paper does not impose any restrictions on the materialization task.

References

[AB94] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19–20:9–71, 1994.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[ALS98] M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in Datalog programs. *Journal of Logic and Computation*, 8(6):745–775, 1998.

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. 7th ACM Symp. on Principles of Database Systems*, pages 240–250, 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.

[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.

[BK98a] A. J. Bonner and M. Kifer. Results on reasoning about updates in Transaction Logic. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 164–194. Springer-Verlag, Berlin, Germany, 1998.

[BK98b] A. J. Bonner and M. Kifer. The state of change: A survey. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 1–36. Springer-Verlag, Berlin, Germany, 1998.

[BKC93] A. J. Bonner, M. Kifer, and M. Consens. Database programming in Transaction Logic. In *Proc. 4th Int. Workshop on Database Programming Languages, New York City*, pages 309–337, 1993.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[BR91] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–299, 1991.

[Bry90] F. Bry. Intensional updates: Abduction via deduction. In *Proc. 7th Int. Conf. on Logic Programming (ICLP '90), Jerusalem, Israel*, pages 561–575, 1990.

[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.

[Cro90] H. J. Cronau. A transaction concept for deductive databases. Dissertation, Universität Dortmund, 1990.

[DD97] C. J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, 4th edition, 1997.

[FWF00] A. Fent, C. A. Wichert, and B. Freitag. Logical update queries as open nested transactions. In G. Saake, K. Schwarz, and C. Türker, editors, *Transactions and Database Dynamics*, volume 1773 of *LNCS*, pages 46–67. Springer-Verlag, Berlin, Germany, 2000. To appear.

[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.

[HD91] U. Halici and A. Dogac. An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17(7):712–724, 1991.

[JWA+92] S. L. P. Jones, P. L. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnson, R. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the functional programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), 1992.

[KM90] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 650–661, 1990.

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

[KT90] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[Lef94] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. *New Generation Computing*, 12(2):131–160, 1994.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proc. 7th Int. Conf. on Management of Data (COMAD '95), Pune, India*, 1995.

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

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. In *Proc. Int. Workshop on Logic in Databases (LID '96), Pisa, Italy*, 1996.

[LRL+97] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.

[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.

[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.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. 7th ACM Symp. on Principles of Database Systems*, pages 251–262, 1988.

[NTR87] L. Naish, J. A. Thom, and K. Ramamohanarao. Concurrent database updates in PROLOG. In *Proc. 4th Int. Conf. on Logic Programming (ICLP '87), Melbourne, Australia*, volume 1, pages 178–195, 1987.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

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

[RSS+97] P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In *Proc. 4th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR '97), Dagstuhl, Germany*, pages 430–440, 1997.

[SWM93] P. Spruit, R. Wieringa, and J. J. Meyer. Dynamic Database Logic: the first-order case. In U. W. Lipeck and B. Thalheim, editors, *Modelling Database Dynamics*, pages 103–120. Springer-Verlag, Berlin, Germany, 1993.

[vG89] A. van Gelder. The alternating fixpoint of logic programs with negation (extended abstract). In *Proc. 8th ACM Symp. on Principles of Database Systems*, pages 1–10, 1989.

[vG92] A. van Gelder. The well-founded semantics of aggregation. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pages 127–138, 1992.

[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.

[Wad95a] P. Wadler. How to declare an imperative. In *Proc. Int. Symp. on Logic Programming (ILPS '95), Portland, Oregon*, pages 18–32, 1995.

[Wad95b] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer-Verlag, Berlin, Germany, 1995.

[WF97a] 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.

[WF97b] C. A. Wichert and B. Freitag. Capturing database dynamics by deferred updates. Technical Report MIP-9704, Universität Passau (FMI), 1997. Available in the WWW: `http://daisy.fmi.uni-passau.de/papers/`.

[WFF98a] C. A. Wichert, A. Fent, and B. Freitag. How to execute ULTRA transactions. Technical Report MIP-9812, Universität Passau (FMI), 1998. Available in the WWW: `http://daisy.fmi.uni-passau.de/papers/`.

[WFF98b] 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.

[WFF98c] 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://daisy.fmi.uni-passau.de/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.

[Zan93] C. Zaniolo. On the unification of active databases and deductive databases. In *Proc. 11th British Nat. Conf. on Databases, Keele, UK*, pages 23–39, 1993.

[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.

Appendix A

Here we will give the detailed proofs that have been omitted in Section 3.

Proposition 3.9 The following properties hold for the connectives ",", ":", and $\vee$.

1. The concurrent conjunction "," is commutative (a) and associative (b) and has NOP as a neutral element (c).

2. The sequential conjunction ":" is associative (a) and has NOP as a neutral element (b).

3. The disjunction $\vee$ is commutative (a) and associative (b).

Proof: The proof is straightforward mainly using Definition 3.8 and the algebraic properties required in Definition 3.2. Let φ, ψ, and χ be arbitrary ground update formulas. For two given update formulas, we show the equality of their interpretation by comparing the elements $(\Delta_C, \Delta) \in \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$.

1. First, we show the properties of the concurrent conjunction ",".

(a) Commutativity

$$\begin{array}{ll}
& (\Delta_C, \Delta) \in I(\varphi, \psi) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\underset{(3.2)}{\Longleftrightarrow} & \text{there exist } \Delta_2, \Delta_1 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta = \Delta_2 \sqcup \Delta_1 \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\psi, \varphi)
\end{array}$$

(b) Associativity

$$\begin{array}{ll}
& (\Delta_C, \Delta) \in I(\ [\varphi, \psi], \chi\ ) \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_{1-2}) \in I(\varphi, \psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \text{ and } \Delta = \Delta_{1-2} \sqcup \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_1, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \\
& \text{and } \Delta_{1-2} = \Delta_1 \sqcup \Delta_2 \text{ and } \Delta = \Delta_{1-2} \sqcup \Delta_3 \\
\underset{(3.2)}{\Longleftrightarrow} & \text{there exist } \Delta_1, \Delta_{2-3}, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \\
& \text{and } \Delta_{2-3} = \Delta_2 \sqcup \Delta_3 \text{ and } \Delta = \Delta_1 \sqcup \Delta_{2-3} \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3} \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_{2-3}) \in I(\psi, \chi) \text{ and } \Delta = \Delta_1 \sqcup \Delta_{2-3} \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\ \varphi, [\psi, \chi]\ )
\end{array}$$

(c) Neutral element

$$\begin{array}{ll}
& (\Delta_C, \Delta) \in I(\varphi, NOP) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(NOP) \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta_2 = \Delta_\varepsilon \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\underset{(3.2)}{\Longleftrightarrow} & (\Delta_C, \Delta) \in I(\varphi)
\end{array}$$

We only have shown that $I(\varphi, NOP) = I(\varphi)$ holds. The equation $I(NOP, \varphi) = I(\varphi)$ follows from the commutativity of ",".

2. Next, we show the properties of the sequential conjunction ":".

(a) Associativity

$$
\begin{array}{cl}
& (\Delta_C, \Delta) \in I(\ [\varphi : \psi] : \chi\ ) \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_{1-2}) \in I(\varphi : \psi) \text{ and } (\Delta_C \oplus \Delta_{1-2}, \Delta_3) \in I(\chi) \text{ and } \Delta = \Delta_{1-2} \oplus \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_1, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \text{ and } (\Delta_C \oplus \Delta_{1-2}, \Delta_3) \in I(\chi) \\
& \text{and } \Delta_{1-2} = \Delta_1 \oplus \Delta_2 \text{ and } \Delta = \Delta_{1-2} \oplus \Delta_3 \\
\underset{(3.2)}{\Longleftrightarrow} & \text{there exist } \Delta_1, \Delta_{2-3}, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \\
& \text{and } ((\Delta_C \oplus \Delta_1) \oplus \Delta_2, \Delta_3) \in I(\chi) \\
& \text{and } \Delta_{2-3} = \Delta_2 \oplus \Delta_3 \text{ and } \Delta = \Delta_1 \oplus \Delta_{2-3} \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3} \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_{2-3}) \in I(\psi : \chi) \text{ and } \Delta = \Delta_1 \oplus \Delta_{2-3} \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\ \varphi : [\psi : \chi]\ )
\end{array}
$$

(b) Neutral element

$$
\begin{array}{cl}
& (\Delta_C, \Delta) \in I(\varphi : NOP) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(NOP) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta_2 = \Delta_\varepsilon \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\underset{(3.2)}{\Longleftrightarrow} & (\Delta_C, \Delta) \in I(\varphi)
\end{array}
$$

$$
\begin{array}{cl}
& (\Delta_C, \Delta) \in I(NOP : \varphi) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& (\Delta_C, \Delta_1) \in I(NOP) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\varphi) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
& \Delta_1 = \Delta_\varepsilon \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\varphi) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\underset{(3.2)}{\Longleftrightarrow} & (\Delta_C, \Delta) \in I(\varphi)
\end{array}
$$

3. The proof of the properties (a) and (b) of the disjunction $\vee$ is trivial. □

Proposition 3.23 Let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be write-compatible update request sets. Then the equality

$$\Delta_1 \oplus \Delta_2 = \Delta_1 \sqcup \Delta_2$$

holds.

Proof: The following equivalences hold for arbitrary insertion requests $+r(\vec{t})$:

$$
\begin{array}{cl}
& +r(\vec{t}) \in \Delta_1 \oplus \Delta_2 \\
\Longleftrightarrow & (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2) \ \vee \ +r(\vec{t}) \in \Delta_2 \\
\underset{(subsmpt.)}{\Longleftrightarrow} & +r(\vec{t}) \in \Delta_1 \ \vee \ +r(\vec{t}) \in \Delta_2 \\
\Longleftrightarrow & +r(\vec{t}) \in \Delta_1 \sqcup \Delta_2
\end{array}
$$

Have a closer look at the equivalence marked “subsmpt.”. ‘$\Rightarrow$’ is trivial, as the condition on the left hand side is stronger than the condition on the right. However, due to the assumption of write-compatibility, the converse ‘$\Leftarrow$’ also holds, as the deletion request $-r(\vec{t})$ cannot be contained in Δ_2, if $+r(\vec{t}) \in \Delta_1 \sqcup \Delta_2$ holds.

As the definitions are symmetric, similar equivalences hold for deletion requests $-r(\vec{t})$.

In both compositions, the read tags are merged like in the usual set union. Thus,

$$?r \in \Delta_1 \oplus \Delta_2 \iff ?r \in \Delta_1 \cup \Delta_2 \iff ?r \in \Delta_1 \sqcup \Delta_2$$

holds for arbitrary read tags $?r$. This completes the proof of the equality of $\Delta_1 \oplus \Delta_2$ and $\Delta_1 \sqcup \Delta_2$. $\square$

Theorem 3.24 The algebraic properties of Definition 3.2 hold for the objects defined for the ULTRA instance tailored to logic databases.

Proof: The proof essentially relies on the definitions of $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$.

1. Recall that the concurrent composition $\sqcup$ is defined as the set union. Further, $\Delta_\varepsilon = \emptyset$ holds. Properties (a) to (c) required for $\sqcup$ thus follow from the properties of the set union.

2. Next, we show the properties required for $\oplus$.

(a) Let arbitrary update request sets $\Delta_1, \Delta_2, \Delta_3 \in \mathcal{T}$ be given.

The following equivalences hold for arbitrary insertion requests $+r(\vec{t})$:

$$\begin{array}{cl}
 & +r(\vec{t}) \in (\Delta_1 \oplus \Delta_2) \oplus \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \oplus \Delta_2 \ \wedge \ -r(\vec{t}) \notin \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & [\ (\ (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2) \vee +r(\vec{t}) \in \Delta_2) \wedge \\
 & \ -r(\vec{t}) \notin \Delta_3\ ] \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \vee \\
 & (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \vee \\
(subsmpt.) & (+r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \in \Delta_3 \wedge -r(\vec{t}) \notin \Delta_3) \vee \\
 & (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \wedge (-r(\vec{t}) \notin \Delta_2 \vee +r(\vec{t}) \in \Delta_3) \wedge -r(\vec{t}) \notin \Delta_3) \vee \\
 & (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \wedge \neg\ [\ (-r(\vec{t}) \in \Delta_2 \wedge +r(\vec{t}) \notin \Delta_3) \vee -r(\vec{t}) \in \Delta_3\ ]\ ) \vee \\
 & (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \notin \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_3 \\
\iff & (+r(\vec{t}) \in \Delta_1 \ \wedge \ -r(\vec{t}) \notin \Delta_2 \oplus \Delta_3) \ \vee \ +r(\vec{t}) \in \Delta_2 \oplus \Delta_3 \\
\iff & +r(\vec{t}) \in \Delta_1 \oplus (\Delta_2 \oplus \Delta_3)
\end{array}$$

Have a closer look at the equivalence marked "subsmpt.". '$\Rightarrow$' is trivial, as the three disjuncts on the left hand side also occur on the right hand side. However, the converse '$\Leftarrow$' also holds, as the disjunct

$$+r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \in \Delta_3 \wedge -r(\vec{t}) \notin \Delta_3$$

which does not occur on the left hand side implies the disjunct

$$+r(\vec{t}) \in \Delta_3.$$

As the definitions are symmetric, similar equivalences hold for deletion requests $-r(\vec{t})$.

The sequential composition $\oplus$ merges the read tags exactly like the usual set union. Thus, the equivalences

$$?r \in (\Delta_1 \oplus \Delta_2) \oplus \Delta_3 \iff ?r \in \Delta_1 \cup \Delta_2 \cup \Delta_3 \iff ?r \in \Delta_1 \oplus (\Delta_2 \oplus \Delta_3)$$

hold for arbitrary read tags $?r$. This completes the proof of the associativity of $\oplus$.

(b) To show that Δ_ε is a neutral element, let $\Delta \in \mathcal{T}$ be arbitrarily chosen.

$$\begin{array}{rcl}
\Delta \oplus \Delta_\varepsilon & = & \{+r(\vec{t}) \mid (+r(\vec{t}) \in \Delta \wedge -r(\vec{t}) \notin \emptyset) \vee +r(\vec{t}) \in \emptyset\} \cup \\
& & \{-r(\vec{t}) \mid (-r(\vec{t}) \in \Delta \wedge +r(\vec{t}) \notin \emptyset) \vee -r(\vec{t}) \in \emptyset\} \cup \\
& & \{?r \mid ?r \in \Delta \ \vee \ ?r \in \emptyset\} \\
& = & \{+r(\vec{t}) \mid +r(\vec{t}) \in \Delta\} \ \cup \ \{-r(\vec{t}) \mid -r(\vec{t}) \in \Delta\} \ \cup \ \{?r \mid ?r \in \Delta\} \\
& = & \Delta \\
\Delta_\varepsilon \oplus \Delta & = & \{+r(\vec{t}) \mid (+r(\vec{t}) \in \emptyset \wedge -r(\vec{t}) \notin \Delta) \vee +r(\vec{t}) \in \Delta\} \cup \\
& & \{-r(\vec{t}) \mid (-r(\vec{t}) \in \emptyset \wedge +r(\vec{t}) \notin \Delta) \vee -r(\vec{t}) \in \Delta\} \cup \\
& & \{?r \mid ?r \in \emptyset \ \vee \ ?r \in \Delta\} \\
& = & \{+r(\vec{t}) \mid +r(\vec{t}) \in \Delta\} \ \cup \ \{-r(\vec{t}) \mid -r(\vec{t}) \in \Delta\} \ \cup \ \{?r \mid ?r \in \Delta\} \\
& = & \Delta
\end{array}$$

3. Next, we show the consistency properties.

(a) The first implication holds, since $\sqcup$ is defined as the set union: if Δ_1 or Δ_2 were not consistent, then $\Delta_1 \sqcup \Delta_2$ would not be consistent, too.

(b) Now let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be consistent update request sets. Assume that $\Delta_1 \oplus \Delta_2$ is not consistent. Then an atom $r(\vec{t}) \in \mathcal{B}$ exists, such that $+r(\vec{t}), -r(\vec{t}) \in \Delta_1 \oplus \Delta_2$. We can reason as follows:

$$\begin{array}{rl}
& +r(\vec{t}) \in \Delta_1 \oplus \Delta_2 \ \wedge \ -r(\vec{t}) \in \Delta_1 \oplus \Delta_2 \\
\Longrightarrow & (\ (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2) \vee +r(\vec{t}) \in \Delta_2\ ) \wedge \\
& (\ (-r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \notin \Delta_2) \vee -r(\vec{t}) \in \Delta_2\ ) \\
\Longrightarrow & (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2 \wedge -r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \notin \Delta_2) \vee \\
& (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \in \Delta_1 \wedge +r(\vec{t}) \notin \Delta_2) \vee \\
& (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \notin \Delta_2 \wedge -r(\vec{t}) \in \Delta_2) \vee \\
& (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \in \Delta_2) \\
\Longrightarrow & (+r(\vec{t}) \in \Delta_1 \wedge -r(\vec{t}) \in \Delta_1) \ \vee \ (+r(\vec{t}) \in \Delta_2 \wedge -r(\vec{t}) \in \Delta_2)
\end{array}$$

The last statement contradicts the precondition saying that Δ_1 and Δ_2 are consistent. Thus, our assumption must be false, which means that $\Delta_1 \oplus \Delta_2$ is consistent.

(c) It is easy to see that $\Delta_2 \subseteq \Delta_1 \oplus \Delta_2$ holds. Consequently, if $\Delta_1 \oplus \Delta_2$ is consistent, Δ_2 must be consistent, too.

4. Next, we show the properties required for $\oplus_E$.

(a) The assertion concerning the exchange of execution $\oplus_E$ and sequential composition $\oplus$ can be proved in a similar fashion as the associativity of $\oplus$ (see property 2). Note that $DB \oplus_E (\Delta_1 \oplus \Delta_2)$ is well-defined, since $\Delta_1 \oplus \Delta_2$ is consistent (see property 3).

(b) To show that Δ_ε is a neutral element, let $DB \in \mathcal{S}$ be arbitrarily chosen.

$$\begin{array}{rl}
& DB \oplus_E \Delta_\varepsilon \\
= & \{r(\vec{t}) \mid (r(\vec{t}) \in DB \wedge -r(\vec{t}) \notin \emptyset) \vee +r(\vec{t}) \in \emptyset\} \\
= & \{r(\vec{t}) \mid r(\vec{t}) \in DB\} \\
= & DB
\end{array}$$

5. Recall that the concurrent composition ($\sqcup$ and $\bigsqcup$) is defined as the set union. Consequently, the properties (a) and (b) required for $\bigsqcup$ can be easily shown. $\square$

Lemma 3.41 Let $\mathcal{J} \subseteq \mathcal{I}$ be a non-empty set of interpretations over $\mathcal{B}_{DU}$. Define a new interpretation I by

$$I(p(\vec{t})) := \bigcap_{I' \in \mathcal{J}} I'(p(\vec{t}))$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. Then

$$I(U) \subseteq \bigcap_{I' \in \mathcal{J}} I'(U)$$

holds for every ground update goal U.

Proof: We show by structural induction that $I(U) \subseteq I'(U)$ holds for arbitrary $I' \in \mathcal{J}$ by comparing their elements $(\Delta_C, \Delta) \in \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$.

Base cases:

Essentially, we apply Definition 3.8.

1. DB literal

   We show the assertion only for a positive DB literal. The proof for a negative DB literal is entirely analogous.

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I(q(\vec{t})) \\ \Longrightarrow & I_{DB}(s_0 \oplus_E \Delta_C) \models q(\vec{t}) \text{ and } \Delta = Log(q(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(q(\vec{t})) \end{array}$$

2. NOP literal

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I(NOP) \\ \Longrightarrow & \Delta = \Delta_\varepsilon \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(NOP) \end{array}$$

3. Basic update atom

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I(u(\vec{t})) \\ \Longrightarrow & \Delta = Upd(u(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(u(\vec{t})) \end{array}$$

4. Definable update atom

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I(p(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(p(\vec{t})) \end{array}$$

   The assertion follows directly from the definition of I.

Induction step:

By the induction hypothesis, $I(\varphi) \subseteq I'(\varphi)$ holds for any direct proper subgoal φ of the composite goals analyzed in the following. Using Definition 3.8 we can reason as follows.

1. Concurrent conjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I(\varphi, \psi) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \underset{(IH)}{\Longrightarrow} & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I'(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I'(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\varphi, \psi) \end{array}$$

2. Sequential conjunction

$$\begin{array}{cl} & (\Delta_C, \Delta) \in I(\varphi : \psi) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \underset{(IH)}{\Longrightarrow} & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I'(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I'(\psi) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\varphi : \psi) \end{array}$$

3. Disjunction

$$\begin{array}{cl} & (\Delta_C, \Delta) \in I(\varphi \vee \psi) \\ \Longrightarrow & (\Delta_C, \Delta) \in I(\varphi) \text{ or } (\Delta_C, \Delta) \in I(\psi) \\ \underset{(IH)}{\Longrightarrow} & (\Delta_C, \Delta) \in I'(\varphi) \text{ or } (\Delta_C, \Delta) \in I'(\psi) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\varphi \vee \psi) \end{array}$$

4. Existential quantification

Consider a quantification $\exists \vec{X}\, \varphi$, where $\vec{X} = X_1, \ldots, X_n$. By the induction hypothesis,

$$I(\varphi[\vec{X} / \vec{t}\,]) \subseteq I'(\varphi[\vec{X} / \vec{t}\,])$$

holds for all instances $\varphi[\vec{X} / \vec{t}\,]$ of the update subgoal φ.

$$\begin{array}{cl} & (\Delta_C, \Delta) \in I(\exists \vec{X}\, \varphi) \\ \Longrightarrow & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\ & (\Delta_C, \Delta) \in I(\varphi[\vec{X} / \vec{t}\,]) \\ \underset{(IH)}{\Longrightarrow} & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\ & (\Delta_C, \Delta) \in I'(\varphi[\vec{X} / \vec{t}\,]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\exists \vec{X}\, \varphi) \end{array}$$

5. Bulk quantification

Consider a bulk quantification $\# \vec{X}\, [A \mapsto \varphi]$, where $\vec{X} = X_1, \ldots, X_n$. By the induction hypothesis,

$$I(\varphi[\vec{X} / \vec{t}\,]) \subseteq I'(\varphi[\vec{X} / \vec{t}\,])$$

holds for all instances $\varphi[\vec{X} / \vec{t}\,]$ of the update subgoal φ.

Let $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ be arbitrarily chosen. The set

$$T_{A,\Delta_C} = \{(\vec{t}\,) \in \mathcal{U}^n \mid I_{DB}(s_0 \oplus_E \Delta_C) \models A[\vec{X} / \vec{t}\,]\}$$

does not depend on any interpretation of update formulas.

The case $T_{A,\Delta_C} = \emptyset$ is trivial:

$$\begin{array}{cl} & (\Delta_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \\ \Longrightarrow & \Delta = Log(A[\vec{X} / \vec{all}\,]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\# \vec{X}\, [A \mapsto \varphi]) \end{array}$$

For $T_{A,\Delta_C} \neq \emptyset$ we get:

$$\begin{array}{cl} & (\Delta_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{t}\,) \in T_{A,\Delta_C} :\ (\Delta_C, f(\vec{t}\,)) \in I(\varphi[\vec{X} / \vec{t}\,]) \\ & \text{and } \Delta = Log(A[\vec{X} / \vec{all}\,]) \ \oplus\ \bigsqcup_{(\vec{t}\,) \in T_{A,\Delta_C}} f(\vec{t}\,) \\ \underset{(IH)}{\Longrightarrow} & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{t}\,) \in T_{A,\Delta_C} :\ (\Delta_C, f(\vec{t}\,)) \in I'(\varphi[\vec{X} / \vec{t}\,]) \\ & \text{and } \Delta = Log(A[\vec{X} / \vec{all}\,]) \ \oplus\ \bigsqcup_{(\vec{t}\,) \in T_{A,\Delta_C}} f(\vec{t}\,) \\ \Longrightarrow & (\Delta_C, \Delta) \in I'(\# \vec{X}\, [A \mapsto \varphi]) \end{array}$$

□

Lemma 3.42 Let P_{UP} be an update program, let $\mathcal{M} \subseteq \mathcal{I}$ be a non-empty set of models of P_{UP}. Define a new interpretation M by

$$M(p(\vec{t})) := \bigcap_{M' \in \mathcal{M}} M'(p(\vec{t}))$$

for all $p(\vec{t}) \in \mathcal{B}_{DU}$. Then M is a model of P_{UP}.

Proof: An interpretation I is a model of P_{UP}, iff every rule is interpreted by $\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$. By Definition 3.8, cases (Impl) and (Univ), this is true for a rule r, iff for every ground instance $U \to p(\vec{t})$ of r the set inclusion $I(U) \subseteq I(p(\vec{t}))$ holds.

Now let $U \to p(\vec{t})$ be a ground instance of a rule $r \in P_{UP}$. Because $\mathcal{M}$ is a set of models of P_{UP}, we know that $M'(U) \subseteq M'(p(\vec{t}))$ holds for every $M' \in \mathcal{M}$. We have to show that $M(U) \subseteq M(p(\vec{t}))$ holds, too.

Applying Lemma 3.41 we can conclude:

$$M(U) \subseteq \bigcap_{M' \in \mathcal{M}} M'(U) \subseteq \bigcap_{M' \in \mathcal{M}} M'(p(\vec{t})) = M(p(\vec{t}))$$

□

Appendix B

Here we show the complete calendar example.

EDB Schema and Instance (DB_0):

$entry(Day, Slot, ID)$ associates a time slot $Slot$ on day Day with an appointment identifier ID.

$description(ID, Text)$ stores the descriptive text $Text$ for appointment ID.

entry	*Day*	*Slot*	*ID*
	mon	9	21
	mon	10	0
	mon	11	0
	mon	12	7
	mon	13	7
	mon	14	0
	mon	15	8
	mon	16	10

description	*ID*	*Text*
	7	Meeting Mr. Dean
	8	Hairdresser
	10	Review
	21	Call Mr. Miller

IDB Program (P_{IDB}):

$free(Day, Slot, L)$ determines whether on day Day there are L free time slots starting at time slot $Slot$.

$$
\begin{array}{lcl}
free(D,S,1) & \leftarrow & entry(D,S,0) \\
free(D,S,L) & \leftarrow & L>1,\ free(D,S,1), \\
 & & S1=S+1,\ L1=L-1, \\
 & & free(D,S1,L1)
\end{array}
$$

$duration\_of(ID,L)$ calculates the duration L of appointment ID by counting the number of time slots that are occupied by this appointment. (The rule contains an aggregation construct [Lef94], which is provided by $LOLA$ [ZF97]. It states that the number L of slots $entry(D,S,ID)$ is counted separately for each identifier ID. Of course, a more complex implementation which is based on recursion and negation could be used instead of the quite natural aggregation.)

$$duration\_of(ID,L) \quad \leftarrow \quad \mathsf{group\_by}\,(\,entry(D,S,ID),\ [ID],\ \mathsf{count}(L)\,)$$

$browse(Day,Slot,Text)$ is the general retrieval predicate that joins the relations $entry$ and $description$.

$$browse(D,S,T) \quad \leftarrow \quad entry(D,S,ID),\ description(ID,T)$$

Update Program (P_{UP}):

$do\_allocate(Day,Slot,L,ID)$ recursively tries to allocate L time slots on day Day, starting at time slot $Slot$, for appointment ID. The predicate fails, if one of the time slots is not free.

$$
\begin{array}{lcl}
do\_allocate(D,S,1,ID) & \leftarrow & entry(D,S,0), \\
 & & DEL\ entry(D,S,0),\ INS\ entry(D,S,ID) \\
do\_allocate(D,S,L,ID) & \leftarrow & L>1,\ do\_allocate(D,S,1,ID), \\
 & & S1=S+1,\ L1=L-1, \\
 & & do\_allocate(D,S1,L1,ID)
\end{array}
$$

$do\_deallocate(ID)$ frees all time slots that are allocated for appointment ID. Here we use the bulk quantifier instead of recursion.

$$
\begin{array}{lcl}
do\_deallocate(ID) & \leftarrow & \#\ D,S \\
 & & [\ entry(D,S,ID)\ \mapsto \\
 & & \quad [\ DEL\ entry(D,S,ID),\ INS\ entry(D,S,0)\ ]\ ]
\end{array}
$$

$do\_insert(Day,Slot,L,Text)$ inserts an appointment with descriptive text $Text$ and duration L on day Day, starting at time slot $Slot$. The insertion fails, if one of the time slots is already allocated for another entry. ($\mathsf{newid}(ID)$ is a built-in predicate that generates a new, unique identifier ID every time it is called. Such an operation is provided in many database systems to avoid concurrency problems when searching for unique keys.)

$$
\begin{array}{lcl}
do\_insert(D,S,L,T) & \leftarrow & \mathsf{newid}(ID),\ do\_allocate(D,S,L,ID), \\
 & & INS\ description(ID,T)
\end{array}
$$

$do\_insert\_priority(Day,Slot,Text)$ inserts a high-priority appointment $Text$ on day Day at time slot $Slot$ without checking whether the time slot is free or already occupied.

$$
\begin{array}{lcl}
do\_insert\_priority(D,S,T) & \leftarrow & \mathsf{newid}(ID), \\
 & & DEL\ entry(D,S,0),\ INS\ entry(D,S,ID), \\
 & & INS\ description(ID,T)
\end{array}
$$

$do\_insert\_on\_day(Day, L, Text)$ looks for time slots on day Day that can hold an appointment of duration L and inserts the entry with description $Text$ at one of these times.

$$do\_insert\_on\_day(D, L, T) \quad \leftarrow \quad free(D, S, L) \ : \ do\_insert(D, S, L, T)$$

$do\_delete(Day, Slot)$ deletes the appointment on day Day at time slot $Slot$ by freeing all allocated time slots and removing the description.

$$\begin{array}{lll} do\_delete(D, S) & \leftarrow & entry(D, S, ID),\ do\_deallocate(ID), \\ & & description(ID, T), \\ & & DEL\ description(ID, T) \end{array}$$

$do\_move(ID, Day, Slot)$ moves the appointment with identifier ID to the new day Day at time slot $Slot$. The predicate fails, if the time slots at the destination are not free.

$$\begin{array}{lll} do\_move(ID, D, S) & \leftarrow & [\ duration\_of(ID, L),\ do\_deallocate(ID)\ ] \ : \\ & & do\_allocate(D, S, L, ID) \end{array}$$

Appendix C

Here we show the complete robot example.

DB Predicate:

empty determines whether the robot arm is empty, i.e. does not hold any block.

Basic Update Predicates:

$move(Pos)$ moves the robot arm to position Pos.

pickup moves the robot arm towards the ground trying to pick up a block. We assume that the operation succeeds, even if no block is actually picked up (idle movement).

putdown moves the robot arm towards the ground trying to drop a block. As for *pickup* we allow an idle movement.

Update Program (P_{UP}):

$pickup\_at\_position(Pos)$ moves the empty robot arm to position Pos and picks up a block (idle movement not successful).

$$pickup\_at\_position(P) \quad \leftarrow \quad [\ empty,\ move(P)\ ] \ : \ pickup \ : \ NOT\ empty$$

$putdown\_at\_position(Pos)$ moves the non-empty robot arm to position Pos and puts down the grabbed block (idle movement not successful).

$$\begin{array}{lll} putdown\_at\_position(P) & \leftarrow & [\ NOT\ empty,\ move(P)\ ] \ : \\ & & putdown \ : \ empty \end{array}$$

$move\_block(Pos1, Pos2)$ moves a block from position $Pos1$ to position $Pos2$. The predicate fails, if actually no proper movement of a block is performed.

$$\begin{array}{lcl} move\_block(P1, P2) & \leftarrow & pickup\_at\_position(P1)\ : \\ & & putdown\_at\_position(P2) \end{array}$$

$test\_position(Pos)$ tests whether there is a block lying at position Pos while the blocks world is left unchanged. The robot arm must be empty before the test.

$$\begin{array}{lcl} test\_position(P) & \leftarrow & [\ empty,\ move(P)\ ]\ :\ pickup\ : \\ & & [\ NOT\ empty,\ putdown\ ]\ :\ empty \end{array}$$