View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OPUS - Volltextserver Universität Passau

Dissertation

ULTRA – A Logic Transaction Programming Language

in englischer Sprache verfaßt von

Carl-Alexander Wichert

zur Erlangung des Dr. rer. nat. an der
Fakultät für Mathematik und Informatik
der Universität Passau

Juni 2000

ULTRA – A Logic Transaction Programming Language

Carl-Alexander Wichert

Contents

1 Introduction

1.1 Motivation

Since several years, information systems have been becoming more and more important. The resource "information" is a significant economical factor for modern business companies, and the management of huge amounts of various information has meanwhile become a standard problem. In earlier times, information systems were restricted to classical office applications like bookkeeping and address management. These applications could be established as small, isolated software products. The increase of memory capacities and the better performance of the hardware has made it possible to store and manipulate information beyond the conventional office data. On the one hand the extent and the heterogeneity of information has increased, and on the other hand the combination of distributed data and the integration of multiple software systems has become an issue. Figure 1 schematically shows an information system built upon one or more "core" databases, which are surrounded by various other components from e-mail services, access to the world-wide web, and graphical user interfaces up to complex software components and external hardware devices.

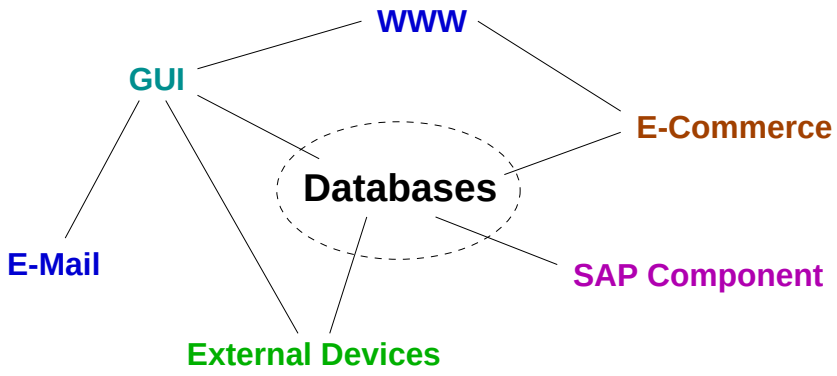

Figure 1: Today's information systems

It is obvious that such complex systems cannot be designed and implemented by one single developer or a small developer team. Thus, it is necessary to find theoretically well-founded standards and methods for software development. These concepts further must enable an *incremental* software development over long time, since complex software is not created at once but refined, improved, and extended with time. Note that software engineering is a broad research field [GJM91, Som96], and we do not claim to provide an overall solution in this thesis. However, we will concentrate on the question of how to specify *complex operations*, i.e. a form of dynamics, in a heterogeneous environment. We are going to develop a *platform- and system-independent language* together with a comprehensive *formal semantics*. By providing means to specify complex operations, we complement the existing concepts related to data modeling and retrieval. Advanced modularization and interface techniques, e.g. object-orientation, can easily be integrated with our approach, as they behave orthogonally. This integration could be a topic of future work.

In the research about database management, the main emphasis was laid on the static aspects, i.e. on the logical modeling and the physical representation of data as well as on the capabilities to retrieve and deduce information. There exist well-founded concepts and languages which enable the definition and exploration of data. We will discuss the SQL language [DD97] for relational

databases as the main representative of these languages. In addition to the static concepts, many approaches for database dynamics, i.e. the change of data with time, have been developed and implemented. But as we will see by a close look, the single approaches are rather limited and cannot easily be combined to an overall concept. The SQL standard, for instance, offers statements to change tuples in a single table, but simultaneous changes in multiple tables cannot be specified. There exist various extensions of SQL that allow procedural combinations of SQL statements, but the semantics is purely operational in contrast to the declarative style of the core language. It is possible to encapsulate multiple statements into transactions that maintain the consistency of the database (see Section 1.2), but the definition of begin and end of the transactions is left to the programmer and does not fit well with the modularization concepts.

Yesterday's database management systems were rather monolithic. It was impossible to communicate with other database systems or external software using the built-in programming paradigms. Of course, database systems could be called via an interface from a program written in a classical language like C. But in this case, transaction and security problems about distributed operations, i.e. operations that do not run within a single database system, had to be solved in the outer programming layer. Modern database architectures are open to act together with other components. Calls to external software are possible e.g. using new extensions of the SQL language, and interfaces are standardized to be platform- and system-independent. In other words, the interoperability features have been improved significantly. However, there still remain many open problems about complex operations and transactions. There exist no common languages for specifying operations at an abstract level with a significant amount of declarativity and with the possibility to do optimizations. All known extensions look like a variant of a procedural language similar to Fortran, C, or Java. They are quite suitable to program algorithms, but do not fit with the basic paradigms of database systems.

1.2 Complex Operations in Information Systems

Now we want to characterize the research topic that has been chosen for this thesis. We are going to illustrate the requirements and the open problems. Let us first have a look at an introductory example.

Example 1.1 [Storage] We model a simplified storage for transport devices like boxes, barrels, buckets, etc. Workers that have access to the storage can take these devices, whenever they have to perform some tasks. The stock of the transport items is checked regularly and, once a day, items that have become low on stock are reordered from a central storage. In the near future, the central storage will provide an e-commerce component which should be used in order to facilitate the ordering and billing process. See Figure 2 for an illustrative overview.

From the technical point of view, there are two main operations in the system: the *delivery* of a single item to a worker and the *reordering* of items having a low stock. Further, we assume that all orders and deliveries have to be logged in a journal for revision purposes. Consequently, the operations work at least on two base tables, one of them storing the stock amounts, the other one storing the changes. When the e-commerce system is involved, the reordering operation becomes more complex and cannot be handled by a database system alone. □

As already mentioned in Section 1.1, we are going to develop a language for the specification of complex operations at an abstract level. This language will be called ULTRA. Now it is time to collect

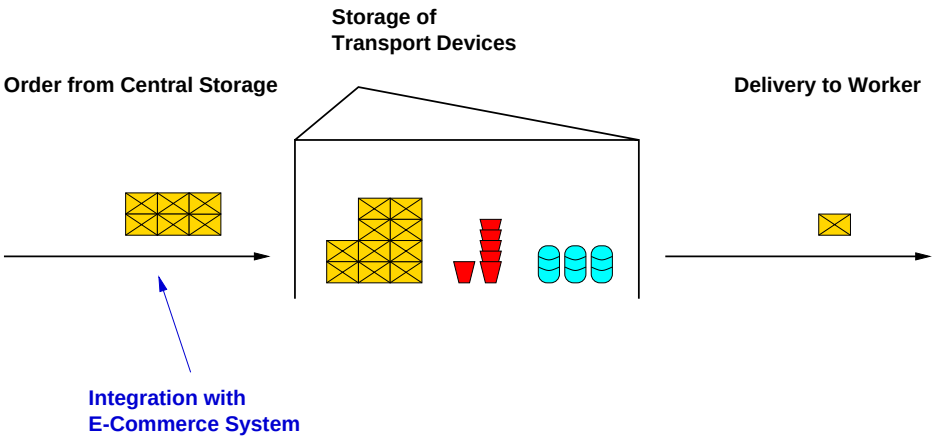

Figure 2: Storage for transport devices

requirements that such a language should satisfy. Since the language is designed for programming in the large, it must enable a *modular* construction of operations under *encapsulation* properties and with the possibility of *reuse*. The meaning of complex operations, i.e. the effects during their execution, can only be completely understood and verified, if the specification language is defined together with a formal semantics. We aim at a *declarative* semantics that shows compositionality and independence of a particular operational model. Based on this foundation, it is possible to develop verification methods for bigger composite systems as well as dedicated execution models and optimization techniques at the operational level. Nevertheless, a declarative approach does not mean that we exclude procedural constructs from the language. *Sequential composition* of predefined operations is one concept known from classical programming languages. A sequential operation naturally originates when one operation has to take changes of other operations into account or when operations are supposed to be performed strictly one after the other. To handle sequential operations at the semantical level, we construct a logical foundation that can incorporate multiple states. From a pragmatic view, it would be desirable to provide also a *concurrent composition* as known from parallel programming languages, e.g. Occam 2 [Wex89]. For instance, the ordering operation of Example 1.1 can be decomposed into two updates on the database and the creation of a mail order, but it would be inadequate to require that these sub-operations are processed sequentially. Thus, the sub-operations should be combined in a concurrent style. Even though the process management has to be very explicit, since there exists no high-level construct for the concurrent composition of statements, today's programming languages like Java allow the specification of concurrency. However, the non-restricted (interleaving) parallelism causes serious problems for the semantics. Although the operational behaviour is rather simple to describe, the overall semantics of composed operations is difficult to define and verify. This is one reason for the fact that parallel programming is a widely open research field. Even in the database context, one of the principles w.r.t. dynamics is the automatic parallelization with a serialization-equivalent effect (cf. Section 2.5). We have chosen a medium approach that features explicit concurrency and guarantees compositionality. Briefly speaking, we allow simultaneous operations as long as they work together in harmony and do not need to communicate with each other. This way, we are not restricted to sequential programs and avoid the semantical problems of parallel programming at the same time. The concurrent composition leaves space for optimizations and is applicable in many database settings, e.g. for separated data objects. Another concept known from relational

databases is the simultaneous update of multiple tuples, called a *bulk update*. We generalize the concepts established in the language SQL and feature the composition of a bulk update from a (possibly already complex) single update. In Example 1.1, the reordering of *all* transport devices that have a low stock is a bulk update. Note that the ordering of a particular item can be considered and implemented as a complex sub-operation. The predefined *basic operations* should not be limited to database operations like insertions and deletions, because the integration with external hard- and software components may require arbitrary basic operations, whose specific semantics is not known at definition time of the program semantics. Thus, we develop an open concept for the semantics and decided to formulate it as a *framework*. This enables a later refinement of the semantics w.r.t. a specific database setting or an external environment.

Operations as described in Example 1.1 typically should behave as transactions. In particular, the ACID properties, which are well-known in database theory [BHG87, BN97, GR93], should hold. These properties require that operations are performed either completely or not at all, that the state of the system is kept consistent, that different operations invoked concurrently do not interfere with each other, and that changes of completed operations are made persistent. In more general words, the ACID properties guarantee a defined and consistent behaviour of the system in case of concurrently running operations and in case of arbitrary failures.

Example 1.2 [Storage (Cont.)] Let us consider the reordering operation of Example 1.1. Provided that the ACID properties are satisfied, it is not possible that a mail order is sent via the e-commerce system, while the entry in the journal or the modification of the stock is omitted. Even if the system crashes, a consistent state will eventually be reached. Next, assume that two workers simultaneously take boxes from the storage and invoke the corresponding bookkeeping operation. Then the ACID properties will ensure that the final amount of boxes is computed and saved well, in particular, it does not come to a lost update. The setting becomes even more complicated, if one of the concurrent operations is aborted in the mean time. □

Transaction models and techniques to guarantee the ACID properties have been studied since a long time. In Section 2.5 we will give a brief overview. With the ULTRA approach we do not invent a new transaction model but a language to specify operations. However we keep an eye on the objective that these operations should be executed as (possibly nested) transactions.

1.3 The ULTRA Approach

Logic programming languages [Llo87], in particular deductive database languages [Das92], are a viable means to describe the static aspects of information systems. They are preferable due to the mathematical and compact syntax, the intuitive clarity, the verifiability with formal methods, etc. 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]. Recall that procedural languages usually allow recursive definitions and modern programming paradigms feature more general data types than just tuples. Thus, it is sensible to develop the ULTRA approach as an extension of the broad concepts of logic (deductive) databases. We claim that the logic language developed in this thesis can easily be tailored to a more user-friendly language like e.g. SQL. The definition of restricted program classes, the augmentation of the language by "syntactical sugar", and the development of precompiling techniques could be practical steps in this direction.

The pure concept of logic databases unfortunately cannot handle dynamics. Thus, a lot of approaches have been developed for the specification of dynamic behaviour. We will give an overview in Section 2.4, where we will also discuss their merits and shortcomings. It turns out that the existing approaches – regarded in isolation – cannot handle the arising problems. One prominent concept, Transaction Logic [BK94, BK96], will be investigated in more detail after the presentation of our own approach.

The main objective of the ULTRA approach is to develop a language concept that allows 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. The separation of physical and logical layers will also lead to considerable improvements within the software development process. After analyzing the existing approaches, we define a new specification language for complex operations, which provides the features discussed in Section 1.2. The language supports classical database operations like insertions and deletions as well as arbitrary other actions (e.g. moving a robot arm, sending an e-mail) as basic operations. Complex operations are defined by update rules that generalize the rules of deductive databases. This technique also resembles the definition of procedures/functions in classical programming languages. Constructs for concurrent composition, sequential composition, and bulk updates are provided in form of logical connectives and quantifiers.

The update language needs a *formal semantics*, such that operations defined in this language get a unique and verifiable meaning. We are trying to keep a high amount of declarativity in order to have space for different evaluation strategies and optimizations. The abstraction from the operational handling turned out to be viable in the setting of relational and deductive databases. We formalize a semantics of update programs founded on the concept of *deferred updates*. A deferred update is represented by a *transition*, which can be considered as an object containing basic actions that have to be performed eventually to accomplish the corresponding operation. As two or more transitions can be better combined than two or more different database or system states, 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 composition which is compositional, i.e. the semantics of the concurrent formula 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 provided by various programming languages. In the simplest form we will discuss, the transitions are sets of insertions and deletions. When read/write conflicts must be dealt with and more general actions are involved, the transitions become more sophisticated. In this case, they must store information about retrieval and order dependencies between basic operations. We define a generic ULTRA framework, which abstracts from the particular notion of states and transitions. To develop a semantics for a specific update language, that 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, the only thing remaining 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, also hold for the instance. In particular, we get 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. Some additional properties concerning transactions and isolation hold, if

further an adequate read-isolation relation on the set of transitions can be constructed.

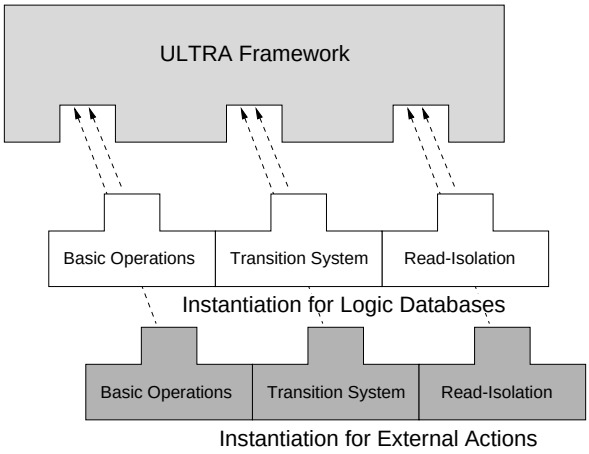

Figure 3: Sample instantiations of the ULTRA framework

Although the ULTRA concept is created with database applications in mind, it can be used as a framework for a great variety of update languages. This is exemplified by presenting two specializations, one of which is conceived to extend logic databases and the other of which is well-suited for the programming of external operations – as illustrated by a variation of the famous robot world example that can be found in many papers, 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 the ULTRA framework, see Figure 3.

1.4 Contributions of this Thesis

To conclude the introduction, we would like to give a concise summary of our research contributions. We will point out the significant increments that we intend to add to the state of the art by working out this thesis.

The main achievement will be the construction of a generic framework for rule-based update languages that serve for the specification of complex operations at an abstract and thus platform- and system-independent level. On the one hand, we allow the modular construction of complex operations using procedural elements known from imperative programming languages, on the other hand, we are able to formulate a logical semantics for these operations. The logical semantics appears compositional and independent of any particular operational model. Consequently, the benefits of declarative languages can be exploited in the context of specification languages that are oriented at the procedural programming paradigm. This is a significant difference to other approaches, which are either defined only at the operational level or provide a declarative paradigm with a limited practical relevance.

The framework we are going to present in this thesis cannot be implemented directly, since it describes requirements but no implementable objects. Therefore, to make the framework operable, an instance that provides the concrete objects and entails the required algebraic properties has to be defined. We will exemplify the instantiation using two application domains for the generic ULTRA concept. However, it should be noted that other instantiations are possible as well. The instances presented in this thesis can be used as a starting point for new instances, i.e. they can

be extended, combined, or revised. This flexibility allows the adaptation of the ULTRA language and its semantics to various other environments. In contrast to ULTRA, related approaches define language and semantics in more concrete terms and thus merely for one application domain, e.g. for a logic database setting.

A further contribution of this thesis is the discussion of problems and possible solutions that become an issue when complex operations have to be executed as transactions. Transaction concepts are well-known in the database community, but they have been neglected in the field of programming languages for long time. We claim to narrow the gap between the different fields in an understandable way, although we leave out many details w.r.t. a concrete operational model for the ULTRA language.

It should be emphasized that the contributions do not arise from the fact that we develop a logic language. We have chosen a logic language, because the syntax is compact and the well-described results in the field of logic databases can serve as the basis for our own work. The ULTRA language defined below has procedural elements and might be considered as non-declarative from a pure logical point of view. As already mentioned in Section 1.3, the mathematical syntax could easily be replaced by a conventional and more user-friedly syntax. The decisive point is the definition of a semantics that assigns to every program a unique meaning which explicitly describes the effects of the programmed operations. This generates a solid foundation for verification techniques, transactional execution strategies, and run-time optimization. Hence, the theoretical work of this thesis can be read as a collection of design principles for a transaction programming language.

2 Open Problems and Research Issues

The main objective of this section is to illuminate the research field and to give reasons for our own work done in the context of this thesis. We will describe how information systems are usually implemented using standard techniques and show some problems that typically occur during that task. Later, we recall the basics of logic (or deductive) databases and briefly discuss important work related to the ULTRA approach. Finally, we cite some work about transaction models and their implementations.

2.1 Relational Databases and SQL

Relational database systems, e.g. ORACLE, are commonly used as the bottom-layer of many information systems. 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 core of SQL consists of a highly declarative retrieval language. After a retrieval task is specified at the logical level (using the `SELECT` statement), the physical retrieval process is automatically defined and optimized by the database management system. This leads to a great efficiency, in particular, when the database designer adds some more information about the physical data layer, e.g. about indexing and clustering. The efficiency and the abstract programming style are some essential reasons to build an information system on top of an existing database software.

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 often are called *bulk updates*. 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. The procedural extensions usually allow to store program objects inside the database (Stored Procedures/Modules), which may facilitate availability and security issues. In contrast to the declarative core language, the procedural extensions are mostly defined at the operational level. Moreover, some constructs may have different semantics dependent on their specific implementations.

The flat transaction model (see Section 2.5) is supported by most database management systems to guarantee that the database is always kept consistent. However, transactions must be handled explicitly using transaction control commands like `COMMIT` and `ROLLBACK`.

Now we continue our introductory example. We will implement the storage application using the concepts of SQL. In particular, we show the problems that arise when complex update operations have to be specified.

Example 2.1 [Storage (Cont.)] Recall the reordering operation of Example 1.1, which should reorder all transport devices that are low on stock. In a relational database one might create a base table *store* with attributes *Item* and *Amount* that relates each transport item *Item* to its current stock *Amount*. The table *store* may have further attributes, e.g. the charging price of an item. The selection of items that are low on stock can be encapsulated into a view *low* having the attribute *Item*. The view definition may look as follows:

```
CREATE VIEW low AS
  SELECT Item FROM store WHERE Amount<10;
```

In this setting, it is easy to implement the stock changes according to a reordering operation by the following update statement:

```
UPDATE store SET Amount=Amount+20
             WHERE Item IN (SELECT Item FROM low);
```

However, this statement alone does not express the whole operation. We also want to log the reorderings in a second table *journal* having at least the attributes *Item* and *Amount*. For instance, if *box* and *bucket* are the low items, then the tuples $(box, 20)$ and $(bucket, 20)$ should be inserted into the *journal* table. Of course, this is possible using an `INSERT` statement that has the `WHERE` clause of the `UPDATE` statement above as a subquery. But this programming style does not meet the well-known requirements for software engineering. First, the condition of the `WHERE` clause is duplicated, which causes redundancy and also may have negative consequences for the evaluation, especially if the constraint $Amount < 10$ of the view is replaced by a more complex constraint. Secondly, the value 20 is coded in both statements, but this could be handled using (global) constants or variables. Thirdly, the set-oriented ordering operation is not modularly constructed out of an ordering operation for single transport items. Although such a sub-operation naturally exists in mind, it cannot be explicitly identified in the composition of the two bulk statements.

An alternative implementation might use the Stored Procedures facilities. The following two procedures implement the reordering operation modularly:

```
PROCEDURE order_low IS
  CURSOR c_low IS SELECT Item FROM low;
  BEGIN
  FOR c_low_rec IN c_low LOOP
    order0(c_low_rec.Item,20);
    END LOOP;
  END order_low;

PROCEDURE order0(i VARCHAR2, a NUMBER) IS
  BEGIN
  UPDATE store SET Amount=Amount+a
               WHERE Item=i;
  INSERT INTO journal VALUES (i,a);
  END order0;
```

We can realize a procedure *order*0 to order a pieces of item i and a procedure *order_low* that performs an iteration over a cursor on the view *low* and calls the procedure *order*0 in each step. It should be mentioned that although the programming style is acceptable, the semantics does not really harmonize with the declarative concepts of the core language. The bulk update feature of SQL is not used, instead a stronger selection takes place in every step (compare the `UPDATE` statements in the two examples). As the updates on both tables are interleaved, it is unlikely that they will be optimized by the database management system. If the view *low* is sorted differently from the base table *store*, an internal optimization becomes difficult, anyway.

One of the requirements discussed in Section 1 is the possibility to perform complex operations as transactions. A procedure itself does not yet describe a transaction. The SQL standard, however, provides a `COMMIT` statement to force the commit of an open transaction. This means that the end of the transaction is reached and all changes made since the last `COMMIT` become persistent. A corresponding `ROLLBACK` statement explicitly aborts a transaction by undoing the recent updates.

To perform operations as transactions, the transaction commands must be integrated into the program code. In the first example, a `COMMIT` must be issued after the two bulk update statements. It is not clear at all that this `COMMIT` operation refers only to the reordering operation. If a previous operation has not been committed, yet, both operations fall into the same transaction sphere. In the procedural environment, it is possible to use transaction commands as program statements. However, this does not fit well with the modularization paradigm, since most systems do not support a nesting of transactions. For instance, if a `COMMIT` statement is inserted into procedure *order*0, the procedure *order_low* cannot be performed as a transaction anymore, because one ordering can complete and commit, while a second ordering can fail and abort. If the `COMMIT` statement is shifted to procedure *order_low*, then more complex operations using *order_low* may suffer from the same problem. Further, the direct call of *order*0 is not processed as a transaction at all, unless the caller provides the control statements needed. It is obviously that the transaction control provided in the SQL standard causes severe problems, which lead to unreadable and error-prone software. Note that these problems become visible even in such small and simplified examples as presented above.

Assume that the storage system is established as described above and we want to combine it with an external e-commerce system. In contrast to the other sub-operations, the sending of mail orders cannot be specified as a bulk update in the SQL core language. If we are lucky, the procedural extensions of the particular database system in use allow the call of external actions. In this case, we can extend the procedure *order*0 in a modular style. In conventional database systems, the only way is to specify the mail orders outside the database system. The following code shows the use of Embedded SQL commands in a procedural language to call the external operation *send_mail_order*(*item*, 20) for all items that are low on stock. The cursor concept facilitates the explicit iteration over a set-oriented query result. Note that additional declarations and statements are necessary to handle the connection to the database and the data transfer.

```
EXEC SQL DECLARE c_low CURSOR FOR
         SELECT Item FROM low;
EXEC SQL OPEN c_low;
REPEAT
   EXEC SQL FETCH c_low INTO item;
   ...
   send_mail_order(item,20);
   ...
UNTIL ...;
EXEC SQL CLOSE c_low;
```

Of course, the Embedded SQL technique can be used in programming languages with modularization constructs. In this case, the procedural extensions of SQL do not need to be used to specify complex operations. Also transaction control is possible from the outside, but it causes the same problems as discussed above.

Let us assume that the reordering operation is encapsulated into a transaction. In any case, the external operation will lie outside the scope of the transaction. Whenever a reordering transaction is committed, it is not clear that the mail order actions have been successful. In the Embedded SQL approach, the database system does not know anything about the call to the e-commerce system. In the other approach it does, but the conventional implementations are so rudimentary that they do not support transactional features for external actions, which is indeed a difficult task (see Section 8 for more details). □

It is undoubted that SQL is a widely used standard language and that its facilities become more and more improved. However, to support programming in the large, several problems, which have been illustrated in Example 2.1, must be solved. In the pure SQL language, updates are not composable to complex operations. The procedural extensions are powerful, but mostly operational and poorly specified. The transaction control features must be used explicitly, they do not harmonize with the modularization concepts, and they are not applicable for external actions.

The language SQL has another disadvantage. It is partly verbose and has a lot of implicit elements. Thus, it is quite unsuitable for a formal treatment.

Hopefully, the ULTRA approach and the results presented in this thesis help to advance the state of the art towards a more declarative update paradigm.

2.2 Imperative Programming Languages

In this section we will refer to imperative programming languages. These languages can be used to implement various applications with or without the integration of given database software. In the example we show Java code, as Java is one major representative of today's programming languages. A priori, an imperative programming language does not provide sophisticated data structures and transactional concepts like a database management system. Modern concepts, e.g. object-orientation and design patterns [GHJV98], facilitate the construction of well-founded and extensible architectures. Nevertheless, the basics must be designed and implemented at least once. The ULTRA approach can be seen as a collection of syntactical and semantical concepts saying how a sophisticated architecture can be built. After the foundations are implemented, the applications itself can be constructed at an abstract level.

In the following, we do not consider a predefined architecture. We show in an example how our storage application can be built from scratch. Note that transactional features are missing. These features would require an enormous additional effort.

Example 2.2 [Storage (Cont.)] Recall Example 1.1, in particular the reordering operation. The application could be implemented as follows using the language Java (see also Figure 4 for a class diagram).

- Data objects:

```
class StoreEntry {
  public String item;
  private int price;
  private int amount;
  public boolean low() { return amount<10; }
  public void update(int a) { amount = amount+a; }
  ...  }
class JournalEntry {
  private String item;
  private String amount;
  ...  }
```

- Operations:

```
class MyApplication {
  List store = new LinkedList();
  List journal = new LinkedList();
  public void order_low() {
    Iterator it = store.iterator();
    while (it.hasNext()) {
      StoreEntry e = (StoreEntry)it.next();
      if (e.low()) { order(e,20); } } }
  public void order(StoreEntry e, int a) {
    e.update(a);
    journal.add(new JournalEntry(e.item,a));
    System.send_mail_order(e.item,a); }
  ...  }
```

The classes *StoreEntry* and *JournalEntry* together with the predefined class *LinkedList* make up the data structure, while the class *MyApplication* contains the desired complex operations as methods. The programming style is similar to that of the procedural extensions of SQL. This is no surprise as the SQL extensions are derived from imperative programming languages. The *LinkedList* data type provides an iterator, which corresponds to a cursor in database languages and avoids an explicit browsing of the internal data structure. This also may be helpful when the list representation is changed, because only the (hidden) iterator has to be revised and not the while loops of the method *order_low*. The class *StoreEntry* provides a method *low*, and every entry object thus contains the information whether it belongs to the view *low* (cf. Example 2.2). The object-orientation of Java enables further modularization and implicit code reuse by inheritance. This, however, is not in the scope of this thesis.

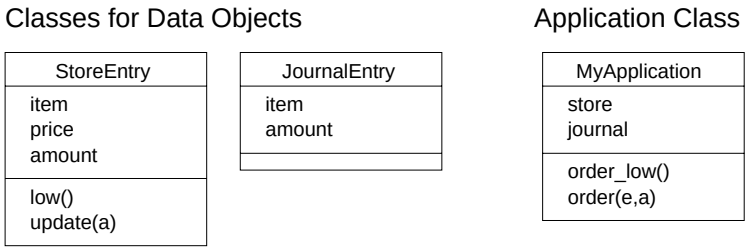

Figure 4: Class diagram of the Java example

The integration of external operations is possible, too. Standard interfaces like CORBA [Sie96] and JDBC [CWH99] facilitate the interoperability with external software. For simple and fixed operations, small wrapper classes might be designed, that transform an abstract procedure call into software-specific calls. □

Imperative programming languages enable a modular and compositional programming of data structures as well as operations. They allow the construction of highly complex architectures, even though the programs are rather operational than declarative. However, the development of a multi-layered architecture that enables a compact high-level programming with a well-defined semantics and meets transactional requirements is not a straight-forward task. For instance, efficient data structures have to be designed, and concurrency and recovery problems have to be solved. In this

light, the ULTRA concept we will present in the following sections can be seen as a proposal of how a sophisticated architecture could be designed. It points out the significant problems and provides viable solutions.

2.3 Logic Databases

Logic (or deductive) databases [Das92, Llo87] can be seen as an extension of relational databases. Essentially, the concepts of relational algebra are augmented by means for recursive programming. Logic databases are usually described in a rule-based language, called Datalog. The language and its semantics are derived from first-order predicate logic [Llo87], although there exist various elements (see below) that are essentially different from the pure logic. The part of the notation that is relevant in the ULTRA context will be defined in detail in Section 3. Here, we will just give a broad overview of the concepts and some illustrative examples.

As depicted in Figure 5, a logic database consists of an *extensional database (EDB)* comprising some (persistent) base relations and an *intensional database (IDB)* defined by a set of deductive rules. The rules correspond to views definitions, and the model-theoretic semantics determines what is true or false in the IDB for a given instance of the EDB.

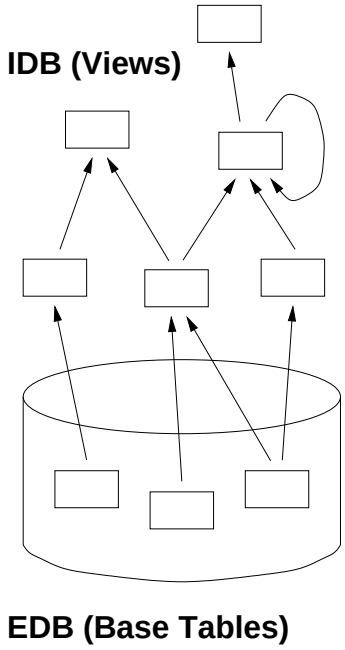

Figure 5: Logic (or deductive) databases

Under the assumption that the rules do not contain any negation, i.e. they are *definite clauses* of the form $p(\ldots) \leftarrow p_1(\ldots),\ \ldots\ , p_n(\ldots)$, every program (together with a given EDB instance) has a unique logical semantics, called the *least Herbrand model*. It can be computed in an iterative way starting with the EDB facts. This strategy is called the *naive bottom-up evaluation*. However, more sophisticated methods have been developed, especially those which do not compute the whole model, when the answer for a restricted query is sought. The methods combine top-down (query-driven) and bottom-up (data-driven) evaluation techniques. One prominent method that encodes top-down elements into a program by rewriting the IDB rules is the *magic set transforma-*

tion [BR91]. The transformed programs are evaluated in a bottom-up fashion. A pure top-down evaluation strategy is the *SLD resolution*, which is known from the language Prolog [MW88a]. See [RU95] for a detailed overview of concepts for efficient query processing in logic databases. Flexible evaluation strategies have also been developed in the *LOLA* project [ZF96, ZF99]. It should be mentioned that these strategies are even capable to compute negation and aggregation semantics as considered below.

The semantics of logic databases becomes more and more complicated, when negation is involved. The negation semantics known from predicate logic is not directly applicable, since it leads to intuitively odd results, which do not fit with the concepts of view definitions. In a pure logic approach, a positive literal in a rule head could be replaced by a negative literal in the body and vice versa. In the semantics for logic databases, however, there is a significant difference: Atoms occurring negated in a rule body are preferentially made false by the semantics, while atoms occurring in a rule head are preferentially made true. This can be explained by the following standard example: Let a fact saying that Tweety is a bird and a rule specifying that a bird flies, if it is not a penguin, be given.

$$
\begin{array}{lcl}
bird(tweety) & & \\
flies(X) & \leftarrow & bird(X),\ NOT\ penguin(X)
\end{array}
$$

Then the fact that Tweety flies, i.e. $flies(tweety)$, should be derived. In contrast, the fact that Tweety is a pengiun, i.e. $penguin(tweety)$, is not an expected conclusion. The main reason for the anomalies of the negation semantics is the *closed world assumption* principle in the field of databases. Explicit specifications are meant to describe the facts that hold, while it is inconvenient to specify what does not hold, too. The latter should be implicitly derived as the complement from the facts that hold or – more operationally speaking – from the facts that can be derived. This simple idea, however, leads to severe problems and opened a whole research branch in the field of logic programming. [AB94] discusses many approaches to handle negation model-theoretically as well as operationally. One semantics which is often chosen for programs containing negation is the *well-founded model semantics* [vG89, vGRS91]. It is applicable for arbitrary programs and subsumes most standard semantics previously defined for restricted program classes.

Aggregation constructs as known from relational databases, e.g. for computing sum, average, maximum, etc., can be integrated into a language for logic databases, too. In addition to the special notation, which does not resemble classical logics anymore, their semantics causes similar problems as the semantics of negation, because an aggregation function operates on a collection of facts instead of a single fact. [vG92] presents a formal approach for aggregation, which extends the ideas of the well-founded model semantics [vG89, vGRS91].

It should be noted that the semantics of logic databases supports a form of complex data types, if (uninterpreted) function symbols are allowed in the rule-based language. In this case, the combination and separation of items can be described declaratively as shown in the following example, where *emptylist* and *cons* serve as constructors for lists (cf. lists in functional or logic programming languages).

$$
\begin{array}{lcl}
member(X, cons(X, R)) & & \\
member(X, cons(Y, R)) & \leftarrow & member(X, R) \\
first(X, cons(X, R)) & & \\
last(X, cons(X, emptylist)) & & \\
last(X, cons(Y, R)) & \leftarrow & last(X, R)
\end{array}
$$

Facts like $member(a, cons(b, cons(a, emptylist)))$ and $first(c, cons(c, emptylist))$ should be derived from this specification. The various model-theoretic semantics, which are generally based on the Herbrand pre-interpretation, can deal with such uninterpreted function symbols, although the termination of the evaluation methods becomes undecidable. Another objective is the integration of more specific interpretations for some constants, function symbols, and predicate symbols. For instance, arithmetic and string expressions should be handled. Foundations and formal semantics for this integration are investigated in the field of constraint logic programming [BC93].

Now we want to model our introduction example as a logic database. Note that we do not consider the dynamic operations, yet.

Example 2.3 [Storage (Cont.)] The static parts of our introduction example (see Examples 1.1 and 2.1) can be expressed as the following logic database:

The extensional database consists of a 3-ary relation $store(Item, Price, Amount)$ and a binary relation $journal(Item, Amount)$.

The actual EDB instance may look as follows. We choose the usual table notation to represent $store$, the representation for $journal$ is omitted.

$store$	$Item$	$Price$	$Amount$
	box	5	2
	$barrel$	20	13
	$bucket$	8	5

The intensional database consists of a unary relation $low(Item)$ which is defined by the following logical rule:

$$low(I) \quad \leftarrow \quad store(I, P, A),\ A < 10$$

Intuitively, the rule reads "low holds for item I, if there exists a stock value A less than 10 for I in the relation $store$". From the EDB instance above, the facts $low(box)$ and $low(bucket)$ can be derived, while $low(barrel)$ does not hold. The variable I is implicitly universally quantified. P and A are universally quantified, too, but as they appear locally in the rule body, they have an existential semantics in this scope (recall the logical semantics of an implication). □

Logical 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 database language SQL [DD97] discussed in Section 2.1. The syntax, which is adopted from predicate logic, is compact and easy to read. Further, the logical semantics and the evaluation of queries are well understood and formalized. The appropriate handling of dynamic behaviour in this context, however, is still a research issue. The most important work in this field will be discussed in the following section.

2.4 Logic-Based Approaches for Updates and Dynamics

In this section we cite a collection of concepts for rule-based update specification and other logical concepts dealing with dynamics. A brief, informal description and classification is provided for each

approach. The merits of the various approaches have been respected during the development of the ULTRA concept. The reader may obtain further information about this field in a recent survey article by Bonner and Kifer [BK98].

The approaches found in the literature can roughly be classified into the following groups. Of course, hybrid approaches are not excluded.

- **Updates in the rule body**

  In this paradigm, the syntax of the rule bodies is extended in a way such that they can accommodate basic update operations, e.g. insertions and deletions, and the rules are interpreted in a top-down fashion. Consequently, the rule heads can be compared with procedural declarations, the bodies with combinations of basic operations and procedure calls. The main advantage of this paradigm is the inherent modularization property: complex update operations can be built hierarchically and recursively in analogy to the (static) IDB (see Figure 6). The naming of operations by the rule heads facilitates the reuse of predefined operations. Further, update queries can be considered as transaction invocations. This generalizes the usual principle that a retrieval task is invoked by an IDB query. These advantages were decisive for the development of the ULTRA concept. Besides ULTRA, also [BK94, BK96, Che97, MBM97, MW88a, MW88b] are founded on this rule paradigm. The top-down interpretation, which can be considered as a form of abduction (cf. Section 7.2), imposes one important problem: it is difficult to define a formal semantics for set-oriented updates (bulk updates). Multiple solutions for an update query should rather be handled by non-determinism. Approaches like [MBM97, NK88], which try to handle them as implicit bulk updates, can be judged as failed: the semantics are partly incomprehensible, show several anomalies, and impose impractical constraints on the programmer. In the ULTRA approach we provide a bulk quantifier with a special semantics. As the quantifier has to be explicitly used and results are interpreted as non-deterministic otherwise, we circumvent the problems of the approaches above.

- **Updates in the rule head**

  In this paradigm, basic update operations can be used as rule heads, and the rules are considered bottom-up as condition/action specifications. This means that a basic update in a rule head is triggered, when the condition of the corresponding rule body is satisfied. Set-oriented updates are naturally justified by this rule interpretation. Their results, however, are dependent on the rule processing strategy. 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]. Further, a priori it is not clear which state a rule body should refer to. Some approaches like [LHL95, Zan93] define a clean semantics by incorporating state identifiers into the language. The programming with absolute or relative state identifiers, however, causes similar problems as the programming with labels or line numbers in imperative languages. Anyway, the bottom-up rule paradigm is not suitable for modular programming of complex operations, and its syntax does not reflect a notion of transaction spheres. Thus, we think that this rule paradigm is rather applicable in other fields like active databases, reactive systems, and continuous processes.

- **Other specification approaches**

  The concept of IDB rules can be combined with procedural concepts. This is done e.g. in [CM93], but analogous problems hold as for the extensions of SQL (cf. Section 2.1). The

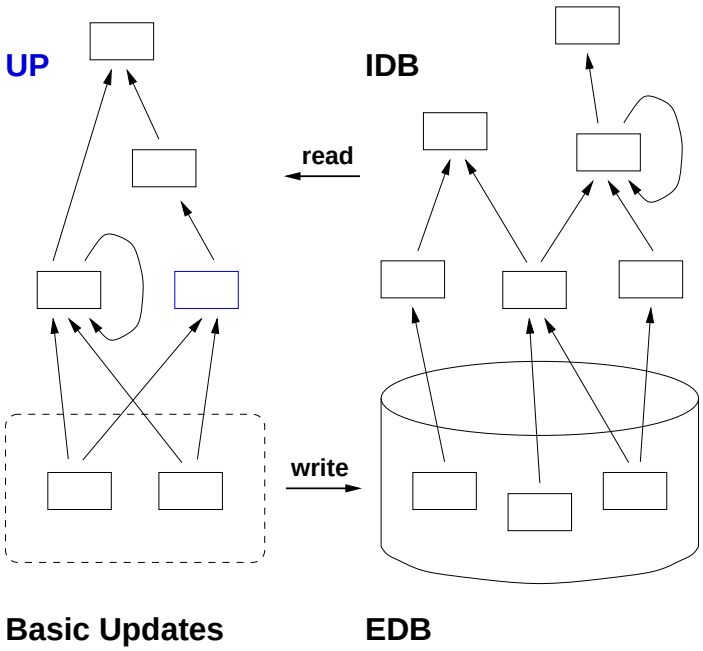

Figure 6: Extension of logic databases by updates

integration of the different paradigms at the semantical level is insufficient, such that the approaches only have an operational semantics.

- **Implicit updates**

  In several cases, updates do not need to be defined explicitly but can be generated automatically by reasoning about given constraints. One prominent example is the concept of implicit *view updates*. As shown e.g. by Kakas, Mancarella [KM90] and Bry [Bry90], requests to change an IDB relation can be translated into sets of changes on the EDB by abductive reasoning about the IDB rules. Unfortunately, a view update may lead to non-deterministic results on the database some of which do not have a sensible semantics from the user's point of view. This is roughly comparable with the problem of negation discussed in Section 2.3: recall the Tweety example where $flies(tweety)$ is an intended conclusion while $penguin(tweety)$ is not. In Section 7.2 we will explain in more detail what problems arise with implicit updates and why we focus on explicit update specifications in the ULTRA approach. It should be mentioned that more complex approaches for implicit updates also exist. In the DaCapo approach [FSMZ95], for instance, the objective is to derive sequences of actions from requirements written as formulas in a temporal logic [Eme90]. The approaches that try to generate updates automatically from abstract requirements can be considered as highly declarative from a pure logical point of view, while other approaches, e.g. the rule-based ones, are often called procedural or operational. Nevertheless, we have decided to rely on one of the rule-based paradigms, since dynamic behaviour obviously contains procedural elements and the pure, declarative approaches have their specific shortcomings. Note that the objective of the ULTRA project is not the development of a new declarative programming paradigm but the development of a compact language that is capable to describe database-oriented operations and has a well-defined semantics.

- **Reasoning about updates and actions**

  Other approaches do not deal with the *specification* of operations, but with the *reasoning* about given operations. While approaches like [Kow92, Rei95] mainly reason about basic actions or sequences of basic actions, other approaches like [LRL+97, SWM93] can reason about composite actions or programs. We will spend some thoughts on these approaches below, since we are interested in operational semantics that perform transactions hypothetically without changing the physical state. In this setting, the reasoning features become an issue.

In the following we use our introductory example to illustrate the two rule paradigms for the specification of complex operations.

Example 2.4 [Storage (Cont.)] Recall Examples 1.1 and 2.3. Our current objective is to specify the operation that reorders all transport items being low on stock. Let $INS\ r(\ldots)$ and $DEL\ r(\ldots)$ denote basic update atoms for insertions and deletions w.r.t. an EDB relation r (r can be either *store* or *journal*). Further, let $send\_mail\_order(\ldots)$ be an external basic operation that issues an order via a given e-commerce system.

Using the ULTRA syntax, a top-down-oriented specification of the reordering operation, would look as follows:

$$
\begin{array}{lcl}
order\_low & \leftarrow & \#\ I\ [\ low(I)\ \mapsto\ order(I, 20)\ ] \\
order(I, A) & \leftarrow & store(I, P, A0),\ A1 = A0 + A, \\
 & & DEL\ store(I, P, A0),\ INS\ store(I, P, A1), \\
 & & INS\ journal(I, A),\ send\_mail\_order(I, A)
\end{array}
$$

The program implements an operation *order_low* on top of a sub-operation *order*. The bulk quantifier # can be interpreted as a "for all" construct. The complete reordering is started by submitting the update query $\leftarrow order\_low$. However, the operation *order* can be used individually or in a completely different context, too. In Appendix A, the interested reader can find the complete storage example modeled in the ULTRA language.

In a bottom-up-oriented environment, one would implement the reordering operation as follows:

$$
\begin{array}{lcl}
INS\ order(I, 20) & \leftarrow & order\_low,\ low(I) \\
DEL\ order\_low & \leftarrow & order\_low \\
DEL\ store(I, P, A0) & \leftarrow & order(I, A),\ store(I, P, A0) \\
INS\ store(I, P, A1) & \leftarrow & order(I, A),\ store(I, P, A0),\ A1 = A0 + A \\
INS\ journal(I, A) & \leftarrow & order(I, A) \\
send\_mail\_order(I, A) & \leftarrow & order(I, A) \\
DEL\ order(I, A) & \leftarrow & order(I, A)
\end{array}
$$

In this setting, *order* and *order_low* are EDB predicates. When the fact *order_low* is made true (by the user) and the program is evaluated bottom-up, its rules will trigger the basic operations that accomplish the reordering. In contrast to the rules of the ULTRA example, the condition/action rules are implicitly set-oriented. Thus all items with a low stock are reordered. Although the program is conceived modularly, it lacks locality properties. The tasks necessary to perform the ordering of a single item are distributed over various rules, and it is hard to grasp what the operation actually does. Moreover, as the program does not explicitly refer to certain states, the

results depend on the rule processing strategy. The operation *order_low* is performed correctly, only if the rules are evaluated simultaneously and the triggered updates are collected in every iteration step. Otherwise, arbitrary incorrect results can occur. □

Now we will give an outline of some prominent specification and reasoning concepts. We discuss the contributions and their relevance within the ULTRA context.

One top-down-oriented logic programming language that enables the specification of complex operations is the well-known language Prolog [MW88a]. Database-oriented updates like insertions and deletions as well as other external operations can be placed into the rule bodies. Their semantics, however, is defined at the operational level, and an abstract logical semantics does not exist. Note that even the semantics of operations without side effects is tied to a top-down left-to-right rule processing strategy. Further, update operations do not run as transactions: since side effects are not backtrackable and isolation spheres cannot be defined, none of the usual transaction properties are guaranteed. Let us explain the main problems using the following two example operations p and q:

$$\begin{array}{lll} p & :- & r(a),\ \mathsf{assert}(r(a)),\ r(b) \\ q & :- & \mathsf{assert}(r(a)),\ r(a),\ r(b) \end{array}$$

In this simple Prolog program, the atoms $r(a)$ (not those within the assert statement) refer to *different* states, the conjunction of the subgoals is *not* commutative. Thus, the operation p can fail, even if q is successful. If q is called and $r(b)$ does not hold, q will lead to a failure, but the assertion of $r(a)$ will not be undone. This violates the ACID properties w.r.t. q. Due to these problems, Prolog has turned out not to be a suitable starting point for a transaction specification language. Naish, Thom, and Ramamohanarao [NTR87] also discuss the problems that arise from non-declarative update constructs as provided in Prolog. They propose a clean solution based on *deferred updates*. The central ideas have been extended and refined by other approaches, e.g. [MBM97], and also by the ULTRA concept.

The language U-Datalog of Montesi et al. [MBM97] is also based on the "updates in the rule body" paradigm and integrates the concept of deferred updates as proposed in [NTR87]. U-Datalog makes the attempt to perform bulk updates by aggregation of success paths in the resolution tree of a query. The semantics can be considered as declarative. However, due to a rigorous aggregation, one always gets a bulk update effect and cannot specify update alternatives as in other approaches. Moreover, the combination of the update requests does not distinguish between the logical conjunction and disjunction, as shown in the following example:

$$\begin{array}{lll} p & \leftarrow & +r(a) \\ p & \leftarrow & +r(b) \\ q & \leftarrow & +r(a),\ +r(b) \end{array}$$

Although the operation p can be considered as a disjunction and q as a conjunction of insertions, both operations imply the same side effect – namely the insertion of $r(a)$ and $r(b)$ – under the semantics of U-Datalog. At the operational level, update queries asked against U-Datalog programs are evaluated in two phases: In the *marking phase*, the query is resolved and basic updates occurring in the bodies are collected for every branch of the resolution tree. In the *update phase*, the updates of *all* successful branches are merged and performed on the EDB. Consistency checks must be

performed in order to guarantee that no insertion and deletion of the same tuple is specified simultaneously. The operational model can easily guarantee atomicity and durability properties. The handling of concurrent transactions, however, has not been discussed, yet. A severe restriction of U-Datalog is the impossibility to specify sequential operations. Of course, sequential operations can be performed as top-level transactions, however, a sequential composition of operations is neither integrated into the rule formalism nor into the declarative semantics.

The language DLP proposed by Manchanda and Warren [MW88b] is based on dynamic logic [KT90], although the programs that are used within the modality operators are atomic. Instead, complex operations are specified by update rules for which a model-theoretic semantics is presented. Like in U-Datalog [MBM97], the basic operations are restricted to insertions and deletions. However, the updates are considered as immediate, and the main contribution is the integration of a sequential composition with the rule formalism. Essentially, the ideas of DLP are subsumed by the ULTRA approach and (Concurrent) Transaction Logic [BK94, BK96]. Operations written in a DLP-like update language can be performed as sequential or nested transactions [Mos85]. Cronau [Cro90] has collected some methods for an adequate transaction processing.

Chen defines an update calculus and a corresponding update algebra [Che95]. The latter is an extension of the relational algebra and deals with deferred updates, concurrent/sequential composition of these increments, and consistency constraints. 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, 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 some earlier publications [WF96, WF97, WFF98b] as well as in Sections 3.2 and 4.3 of this thesis, where it is considered as an instance of the generic ULTRA framework. 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.

Transaction Logic proposed by Bonner and Kifer [BK94] is one of the concepts which have significantly inspired our own work. It forms a modal logic [Eme90] for the representation of changes in which the sequential composition of operations is handled explicitly and arbitrary basic operations can be integrated. The interpretation of formulas is defined w.r.t. state paths. This generalizes the semantics of DLP [MW88b] that is based on pairs of states. The general logical concepts have been restricted to form a rule-based language. This language has a model-theoretic and a proof-theoretic semantics, where multiple answers for the same update query are interpreted as non-deterministic solutions with specific state changes. Unfortunately, the question of how to perform complex operations as transactions is poorly addressed. 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. 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 sequential fragment of the ULTRA instance presented in Sections 3.3 and 4.4 have the same modeling power. Similarities and differences between ULTRA and (Concurrent) Transaction Logic are discussed more formally in Section 7.3.

One prominent example for a bottom-up-oriented environment with a clear logical semantics is the language Statelog of Ludäscher, Hamann, and Lausen [LHL95]. Relative state identifiers are affixed to the literals occurring in the rules, and the semantics is based on temporal logic programming [AM89]. Statelog programs can be transformed into locally stratified programs [Prz88], and the usual minimal model semantics leads to the semantics of the original program. The emphasis in the Statelog approach has been laid on the investigation of termination and determinism properties. Transactional execution is not considered – as opposed to the HiPAC project [DBC96], which also deals with condition/action rules, but on the other hand does not provide an overall semantics. 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 the bottom-up, data/event-driven Statelog evaluation environment. A similar effect can be obtained in the ULTRA context by using a suitable magic set transformation [BR91]. Zaniolo [Zan93] defines a concept similar to Statelog and investigates corresponding properties.

So far, we have presented related work dealing with the logic-based *specification* of updates and transactions rather than with *reasoning* about actions. The ULTRA concept presented in the following sections is designed as a generic framework for specification languages, too. Consequently, the frame problem (see [Rei95] for a discussion) 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 operational semantics that are based on deferred updates and hypothetical reasoning. In this case, the concepts referred to subsequently can serve as a foundation. Reiter [Rei95] describes possible actions and their 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 ULTRA 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 rule formalism. 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.

2.5 Transaction Concepts

At the execution level, we want to consider complex operations as transactions. Thus, it is necessary to discuss the main work about transaction concepts and transaction processing. We give a brief

overview and expose those parts that appear to be relevant for the ULTRA concept.

Transactions are well-known in database theory since a long time now. Traditionally, database systems implement the so-called *ACID properties* which guarantee atomicity, consistency, isolation, and durability of transactions, meaning that transactional update operations must be performed either completely or not at all, the state of the database must be kept consistent, different operations invoked concurrently must not interfere with each other, and changes of completed operations must be persistent. The ACID properties imply that a transaction behaves as an atomic operation – even in presence of concurrency and arbitrary failures. The main research problem in the field of transactions is the operational implementation of the highly abstract properties. For concurrent transactions the *serializability* property has been defined. It states that the results of transactions that are processed in parallel or in an interleaved fashion must be equivalent to the results of any serial execution of the transactions one after the other. Another important issue is the *recoverability* property. It prohibits unsolvable conflicts between the atomicity of one (aborted) transaction and the durability of another (committed) transaction. Further, to guarantee atomicity and durability, changes must be written into (persistent) *logs*. In case of failures, these logs can be considered forwards to redo actions or backwards to undo actions. When they are used backwards, they must contain enough information to restore the previous states. Additional information is also necessary, when non-deterministic operations have to be redone and should lead to a particular state. An extensive treatment of transaction concepts and implementation details can be found e.g. in the textbooks of Gray and Reuter [GR93] or Bernstein et al. [BHG87, BN97].

Most synchronization protocols for performing transactions are based on *object locking*. Before a data object can be accessed, a lock of an appropriate class (e.g. shared lock, exclusive lock) has to be acquired. Conflicts between operations are represented by conflicts on locks such that they can be handled by the transaction scheduler. The well-known *strict two-phase locking protocol* [BHG87] guarantees serializability and recoverability. Locking protocols can be implemented easily, but under some conditions they may produce deadlocks.

The ULTRA concept as it will be instantiated for logic databases in Sections 3.2, 4.3, and 5.4 perfectly fits with *optimistic scheduling protocols* [BHG87], for instance [Här84, KR81, Tho98]. These methods are called optimistic, because checks for conflicts between concurrent transactions are only performed at commit-time, i.e. every transaction is allowed to run to its end. This is usually done in three phases: During execution of a transaction, it is only allowed to read data from the database; the changes are made to a private workspace. When the transaction commits, it enters the validation phase in which its updates are checked for conflicts with other transactions. Depending on the outcome of the validation, the changes are materialized in an atomic write phase, or the whole transaction is aborted. This also shows the main drawback of optimistic protocols: conflicts are detected very late, so a lot of work may be lost. In [HD91] the authors propose a technique called ODL *(optimistic method with dummy locks)*, which merges ideas of locking into optimistic scheduling such that part of the conflicts can be detected earlier. This may save a lot of unnecessary work. Moreover, ODL avoids deadlocks.

Although ULTRA database transactions as described in this paper can be implemented with the techniques of "traditional" transaction processing by using an optimistic protocol (see above), an alternative operational semantics for ULTRA incorporates *nested transactions*, cf. [FWF00, WFF98a]. Nested transactions are transactions that are made up of subtransactions which in turn may be built from other subtransactions, and so on, thus forming transaction trees with basic operations as their leaves. This fits well with the modular specification of complex operations using the "updates in the rule body" paradigm. Nested transactions guarantee the ACID properties for top-level

transactions, while subtransactions may lose some of them; especially durability is usually missing for subtransactions.

Nested transactions are traditionally due to Moss [Mos85], who has written the first exhaustive treatment of this concept. The group around Lynch and Weihl [LMWF94] formalizes nested transactions with the help of I/O automata. In this approach, all components of the system, i.e. transactions, data objects, and the schedulers itself, are modeled as automata. In [AFL+88, FLMW90] some protocols for nested transactions are proved to be correct. Another way to formally capture nested transactions, which stays closer to the classical model of flat transactions, is given in [BBG89]. There, a nested transaction system is modeled as a forest of computations with a given ordering of actions. In contrast to this model which takes the semantics of the actions into account, [HAD97] develops a simpler model for nested transactions in multi-databases which leaves the semantics aside.

Also Schek and Weikum and their research groups have done a lot of work about nested transactions, e.g. [DSW94, Wei91, WS92]. Considerable part of their work concentrates on applying the theory of nested transactions to composite systems, i.e. multi-databases or federated databases. Recently, a new approach [ABFS97] has been proposed, which is based on weak and strong order dependencies and enables a higher degree of parallelism within and between nested transactions.

3 The Update Language ULTRA

3.1 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]. The language can be refined for specific applications as shown in the subsequent sections.

Definition 3.1 [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 3.2, 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 3.2 [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 3.3 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 3.4 [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 references 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 3.5 [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 3.5 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 consistency and 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 (see Section 6.3 for details). 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 3.6 [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 3.5. □

Update goals form rule bodies or top-level update queries (see Examples 3.17 and 3.20).

Definition 3.7 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. □

Remark 3.8 [Renaming of Variables] Quantified variables can be renamed consistently within the scope of the binding quantifier. The renaming must obey the usual constraints known from first-order predicate logic [Llo87]. □

Definition 3.9 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 3.10 [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 *update queries*. An update rule with an empty body is called an *update fact*. The empty body corresponds to the update literal NOP. □

Remark 3.11 Since all variables in a rule are explicitly or implicitly quantified, it is possible to rename them according to Remark 3.8. This corresponds to the renaming of parameters and local variables in other programming languages. □

Definition 3.12 [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 may also 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 4.

3.2 Instantiating the Framework: ULTRA for Logic Databases

In this section we are going to define a more specific ULTRA language that can be used for specifying update operations in logic databases (cf. Section 2.3). The same database language has been proposed as a stand-alone concept in [WFF98b]. In this thesis, however, it is formulated as an instance of the framework presented in Section 3.1. Essentially, we restrict the basic update operations to insertions and deletions and integrate the notion of deductive rules, which serve as view definitions.

As already mentioned in Section 2.3, 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 3.1 is partitioned as follows.

Definition 3.13 [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 3.14 [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 4.3 for details).

Definition 3.15 [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).

Definition 3.16 [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 $\mathcal{B}_{BU}$. Further, $INS\ r$ and $DEL\ r$ can be considered as predicate symbols in $Pred_{BU}$. □

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 database language, we will use another running example. Although the introductory example (see Example 1.1) has been suitable to show the basic problems, it looks rather trivial and does not really point out the power of the results presented in this thesis. As an extended example, we use a simplified version of a personal calendar. Since the discussion of the example is distributed over various sections, the full program is listed in Appendix B.

Example 3.17 [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 throughout the example 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

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

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, “Presentation”) into the *description* relation, respectively. Together, this amounts 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

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

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 $\mathsf{newid}(ID)$ 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 $\mathsf{newid}(ID)$ 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 thesis.

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.

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

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).

$$do\_move(ID, D, S) \leftarrow [\ duration\_of(ID, L),\ do\_deallocate(ID)\ ] : do\_allocate(D, S, L, ID)$$

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 3.18 [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 3.19 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.

3.3 Instantiating the Framework: ULTRA for External Operations

While in Section 3.2 we have shown how a logic database language supporting updates can be derived from the ULTRA framework, we now focus on a rather different application domain: we will show how ULTRA can be used for the specification of complex operations in arbitrary external environments. The main problem lies in the definition of the specific semantics (see Section 4.4). At the syntactical level, no refinements of the ULTRA framework are necessary. So, we can proceed with an example: a driver for a non-intelligent robot. The setting has been adopted from [LRL+97, Rei95].

Example 3.20 [Robot World] Consider a robot working over a (theoretically infinite) grid of discrete positions. The robot can move stepwise in each direction, or it can try to pick up or put down a block at its current position. Corresponding to these basic operations, let the basic update predicates *xstep*, *ystep*, *pickup*, and *putdown* be given. Let *xstep* and *ystep* be unary with the direction as its only parameter (with values -1 and 1), 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 is not aware of what it is really doing, but rather just performs the predefined movements. However, let us assume that the robot is equipped with a sensor that checks whether its hand actually holds a block or is empty. Syntactically, the state of the sensor is modeled by a nullary predicate *empty* (a DB predicate in the sense of Section 3.1). Let the robot have two other sensors to check the coordinates of the current position (X, Y) on the grid. These sensors are modeled by the unary predicates *xpos* and *ypos*.

Next, we want to define an operation $move(X, Y)$ to move the robot to a certain position (X, Y). The operation is composed from *independent* move operations in x- and y-direction, named *xmove* and *ymove*, which are implemented recursively using the basic operations *xstep* and *ystep*. The definition of *xmove* looks at follows:

$$
\begin{array}{lcl}
xmove(X) & \leftarrow & xpos(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X < X0 \ : \ xstep(-1) \ : \ xmove(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X > X0 \ : \ xstep(1) \ : \ xmove(X)
\end{array}
$$

The implementation of *xmove* is straight-forward: steps towards the desired position have to be performed until the target is reached. We use the sequential conjunction ":" in the rule bodies, as the sub-operations should be performed sequentially and the x-position of the robot has to be checked in every intermediate state. The operation *ymove* is defined analogously. Now we can specify the complex *move* operation by the following rule:

$$move(X, Y) \ \leftarrow \ xmove(X), \ ymove(Y)$$

Because the movements in the two orthogonal directions do not interfere with each other, it is possible to compose the subgoals by concurrent conjunction ",". This will allow an operational semantics to perform the two movements of the robot in an interleaved fashion. In contrast, the sequential conjunction would imply that the robot first moves multiple steps in x-direction and then multiple steps in y-direction.

Finally, we are going to specify a more complex composite operation to pick up a block at some position (X, Y). As a precondition we require that the robot is empty, as a postcondition that it is not empty, i.e. that it has actually 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 perform the operations *pickup*, *putdown*, and *move* and check the sensor in the robot's hand by querying the *empty* predicate.

$$pickup\_at\_position(X, Y) \ \leftarrow \ [\ empty, \ move(X, Y)\ ] \ : \ pickup \ : \ NOT \ empty$$

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

4 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, several implications arise for the operational semantics: 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 or a selection of computed transitions must be physically executed with an undo option. The latter point of view also applies for the robot example (cf. Example 3.20), if the robot world is considered as a black box. In this case, the robot 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 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 beyond 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, 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 [Rei95] 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 of the operational model, and the latter can and should be designed and optimized w.r.t. a more specific instance of the ULTRA framework.

To be able to detect conflicts between immediate retrieval and deferred updates, the semantics allows to assign a logging transition to every DB atom. It must be guaranteed by semantical properties that a validation of the corresponding (immediate) read access is possible when the materialization of a logging transition takes place. Note that this validation is called a certification in optimistic synchronization protocols [BHG87]. The read-isolation problem will be investigated in Section 5, where we will also develop an optimistic protocol for the execution of ULTRA transactions.

4.1 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 4.1 [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 4.2). 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. Essentially, it provides a set of states and describes possible state changes. The basic notion of a transition system, which can be found e.g. in [WN95], is refined to meet the special requirements for the ULTRA semantics. In particular, the representation of composite transitions must be supported.

Definition 4.2 [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 that maps every multi-set over $\mathcal{T}$ onto a transition in $\mathcal{T}$,
- $\oplus$ 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) $\Delta_1 \in \mathcal{T}_{Cons} \ \wedge \ \Delta_2 \in \mathcal{T}_{Cons} \iff \Delta_1 \sqcup \Delta_2 \in \mathcal{T}_{Cons}$
   (b) $\Delta_1 \in \mathcal{T}_{Cons} \ \wedge \ \Delta_2 \in \mathcal{T}_{Cons} \implies \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons}$
   (c) $\Delta_2 \in \mathcal{T}_{Cons} \iff \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons}$

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, where $\uplus$ denotes the union of multi-sets. Further

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

   holds for the empty multi-set.

□

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 4.3 For the database language of Section 3.2, 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 4.3 for more details.

When using the ULTRA framework for external operations as demonstrated in Section 3.3, the states will be defined as the states of the external system, and consistent transitions will be defined as partially ordered multi-sets of basic actions. The execution $\oplus_E$ models the state change that results from performing the external actions respecting the given order dependencies. See Section 4.4 for more details. □

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 transitions. 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 4.2 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 6). 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 $\sqcup$. To the 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 4.4 [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 4.5 [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 4.6 [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 4.7 [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 by side effects.

Example 4.8 In the database-oriented ULTRA instance, the function I_{DB} will map each EDB instance onto the well-founded model [vG89, vGRS91] of its extension by the given IDB program. This way, I_{DB} provides the missing semantics for the IDB predicates. *Log* will map an atom over a DB predicate $q \in Pred_{DB}$ onto the set $\{?r_1, \ldots, ?r_n\}$, where $r_1, \ldots, r_n \in Pred_{DB}$ are the EDB predicates q depends on. The update transition assignment *Upd* will be defined as a simple adaptation: for example, it will map an insertion atom $INS\ r(\vec{t})$ onto the singleton set $\{+r(\vec{t})\}$.

In the ULTRA instance for external operations, I_{DB} will yield observable truth values for every external state, while *Log* and *Upd* will be adaptations that simply map atoms to singleton sets of actions. See Sections 4.3 and 4.4 for more details. □

4.2 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.

$$s_0 \quad \stackrel{\Delta}{\longrightarrow} \quad s$$

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.

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

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* [GSZ95, Sha89] 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 4.9. In Section 4.6 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 tacitly use the Herbrand pre-interpretation [Llo87], where every domain element can be represented by a ground term of $\mathcal{U}$.

Definition 4.9 [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 term tuples $(\vec{t}) \in \mathcal{U}^n$,
   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:
   $(\Delta_C, \Delta) \in I(\varphi \to \psi) \ :\Longleftrightarrow$
   $(\Delta_C, \Delta) \in I(\varphi) \Longrightarrow (\Delta_C, \Delta) \in I(\psi)$

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

□

Definition 4.9 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* (cf. Definition 3.2) 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 applied when rewriting update formulas. These properties are discussed formally in Section 6. The most important one is the associativity of ",", ":", and $\vee$ (see Proposition 6.1). Due to this 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 $\varphi_1 \vee \ldots \vee \varphi_n$.

4.3 Specific Semantics for Logic Databases

Now we develop the semantics for the database language presented in Section 3.2. 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. Except for 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 4.10 [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 4.11 [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 will check the absence of read/write conflicts with concurrent transactions. Obviously, the granularity of the read tags is rather coarse. Thus, conflicts may be noticed at the syntactical level which are – semantically – no proper conflicts. The determination of a minimal set of relevant EDB data for any derived information, however, is an undecidable problem [Elk90] and outside the scope of this thesis. Note that even many standard database systems use read locks on whole base relations, if serializability is to be ensured. In the following, read tags are treated like update requests. Note, however, that read access is an immediate operation as opposed to update operations which are deferred. Only the certification of a read access is also deferred.

Definition 4.12 [Update Request Sets, 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 4.13 [States] The set $\mathcal{S}$ of *(database) states* is defined as the set of all EDB instances over the Herbrand base $\mathcal{B}$. □

Definition 4.14 [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 4.12. ($\mathcal{T}_{Cons}$ equals the set denoted by $\mathcal{D}$ in earlier publications [WF97, WFF98b].)

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 4.15 [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 4.16 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 3.17 results in the new state:

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

□

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 (cf. Section 2.3). The well-founded model of a database $\Phi = P_{IDB} \cup DB$, where P_{IDB} is a (fixed) IDB program and $DB \in \mathcal{S}$ is an EDB instance, is denoted by $WFM(\Phi)$.

Definition 4.17 [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, constraints, and extra-logical constructs (computed functions or predicates also called built-ins). This is not considered in the pure semantics. However, these extensions can be seen as part of the state semantics provided by I_{DB}. Consequently, they are non-critical for the ULTRA concept.

Definition 4.18 [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

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

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 4.19 [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 4.20 [Write-Compatibility] Recall Definition 4.4. If two or more consistent update request sets are conforming with each other, we call them also *write-compatible* (cf. [WFF98b]). In Corollary 4.25 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 4.21 Consistent updates request sets are for example:

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

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. Definitions 4.4 and 4.20).

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

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

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

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

□

Proposition 4.23 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 $+r(\vec{t})$:

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

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 $\Delta_1 \sqcup \Delta_2$. □

After having defined the specific concepts, we must verify that they satisfy the required algebraic properties of Definition 4.2. This will be done in the following theorem.

Theorem 4.24 [Algebraic Properties] The algebraic properties of Definition 4.2 hold for the transition system defined for the extended database language.

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

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 $+r(\vec{t})$:

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

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.

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

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 easily be shown. □

Corollary 4.25 [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 4.23 and the algebraic properties proved in Theorem 4.24. □

In Theorem 4.24 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 4.2 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 4.9.

Example 4.26 [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 3.17 be given as the initial state. By Definition 4.9, 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 4.9 is thus not applicable.

Finally, we would like to illustrate the semantics of the bulk quantifier. Have a look at the update formula

$$\psi :\equiv \#\ D, S\ [\ entry(D, S, 7)\ \mapsto\ [\ DEL\ entry(D, S, 7),\ INS\ entry(D, S, 0)\ ]\ ]$$

which is an instance of the rule body defining the update predicate *do_deallocate* (see Example 3.17 and Appendix B). We assume that the EDB instance DB_0 of Example 3.17, where the assertions $I_{DB}(DB_0) \models entry(mon, 12, 7)$ and $I_{DB}(DB_0) \models entry(mon, 13, 7)$ hold and $I_{DB}(DB_0)$ assigns the truth value "false" to other ground instances of the EDB atom $entry(D, S, 7)$, is given as the initial state. Then according to Definition 4.9, case (Bulk),

$$T_{entry(D,S,7),\emptyset} = \{(mon, 12), (mon, 13)\}$$

holds. $T_{entry(D,S,7),\emptyset}$ determines the relevant ground instances of the update subgoal

$$DEL\ entry(D, S, 7),\ INS\ entry(D, S, 0)$$

occurring in ψ, and we must find interpretations for these instances. It is easy to derive the following assertions regardless of any specific interpretation I_{UP} of definable update atoms:

$$\begin{array}{l}(\emptyset, \{-entry(mon, 12, 7), +entry(mon, 12, 0)\}) \\ \quad \in\ \ I(DEL\ entry(mon, 12, 7),\ INS\ entry(mon, 12, 0)) \\ (\emptyset, \{-entry(mon, 13, 7), +entry(mon, 13, 0)\}) \\ \quad \in\ \ I(DEL\ entry(mon, 13, 7),\ INS\ entry(mon, 13, 0))\end{array}$$

Using case (Bulk) of Definition 4.9, we can now deduce

$$(\emptyset, \Delta) \in I(\psi)$$

for the consistent update request set

$$\begin{array}{rcl}\Delta & = & \{?entry, -entry(mon, 12, 7), -entry(mon, 13, 7), \\ & & \ +entry(mon, 12, 0), +entry(mon, 13, 0)\}.\end{array}$$

Informally speaking, Δ is a representation of the bulk update specified by ψ (w.r.t. the initial state DB_0): it contains the update requests of the single updates (see above) and also a read tag $?entry$ that corresponds to the read access necessary to construct the set $T_{entry(D,S,7),\emptyset}$. □

4.4 Specific Semantics for External Operations

In this section we will develop the semantical parts of the ULTRA instance that is adequate for arbitrary external operations (cf. Section 3.3). Like in Section 4.3, we have to define a transition system together with the mappings I_{DB}, Log, and Upd. Essentially, we generalize the structural notion of update request sets to partially ordered multi-sets of actions. Consequently, order constraints between actions and multiple occurrences of actions can be represented. This becomes necessary, when basic operations beyond insertions and deletions are considered.

4.4.1 States and Actions

In the following, we consider a given set $\mathcal{S}$ of states and a given set Σ of actions. We assume that the states and actions are related by an execution function

$$do : \Sigma \times \mathcal{S} \to \mathcal{S}$$

which models the behaviour of the external system.

Let further the DB interpretation

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

be given as the projection of the states onto the observable properties represented by the DB atoms. Let

$$Log^{act} : \mathcal{B} \to \Sigma$$

be a given mapping from the ground DB atoms to the actions, and let

$$Upd^{act} : \mathcal{B}_{BU} \to \Sigma$$

be a given mapping from the ground basic update atoms to the actions. Note that an action assigned by Log^{act} does not need to be a proper action. It may be a simple mark about a retrieval operation that is necessary to check the truth value of the logged DB atom. In this case, we call such an action a *read tag* and assume that its execution does not change the external state.

Definition 4.27 [Compatibility] Two actions $a_1, a_2 \in \Sigma$ are called *compatible* with each other, if

$$do(a_2, do(a_1, s)) = do(a_1, do(a_2, s))$$

holds for every state $s \in \mathcal{S}$. □

Definition 4.28 [Independence] A ground DB atom $A \in \mathcal{B}$ is called *independent* of an action $a \in \Sigma$, if for every state $s \in \mathcal{S}$ and for every finite sequence $a_1, \ldots, a_n$ of actions $a_i \in \Sigma$ the following equivalences hold:

$$I_{DB}(do(a_n, do(\ldots, do(a_1, do(a, s))..))) \models A \iff I_{DB}(do(a_n, do(\ldots, do(a_1, s)..))) \models A$$
$$I_{DB}(do(a_n, do(\ldots, do(a_1, do(a, s))..))) \models \neg A \iff I_{DB}(do(a_n, do(\ldots, do(a_1, s)..))) \models \neg A$$

□

Remark 4.29 [Conditions for Special Instances] In specialized instances of ULTRA, the actions of Σ might be identified with DB atoms (or DB predicates) and basic update atoms. In this case, Σ would be equal to $\mathcal{B} \cup \mathcal{B}_{BU}$ (or $Pred_{DB} \cup \mathcal{B}_{BU}$), Log^{act} would be the identity mapping (or the mapping that maps a DB atom onto its predicate symbol), and Upd^{act} would be the identity mapping. Moreover, it is possible to prefix the actions that are returned by Log^{act} with a special symbol, e.g. "?", to emphasize that they are merely read tags instead of proper actions. As state retrieval usually has no side effect, the actions assigned by Log^{act} should not change the state, i.e. for an action $a \in Log^{act}(\mathcal{B})$ and an arbitrary state $s \in \mathcal{S}$, $do(a, s) = s$ should hold. □

Remark 4.30 The ULTRA instance defined in this section is capable to deal with complex operations in arbitrary systems. Hence, it can be refined to be used even in the database context. For this purpose, the preliminaries required above, i.e. $\mathcal{S}$, Σ, do, I_{DB}, Log^{act}, and Upd^{act}, have to be defined for database specific operations. It is possible to use definitions that resemble the definitions of Section 4.3. For instance, Upd^{act} will map a basic update atom $INS\ r(\vec{t})$ onto the update request $+r(\vec{t})$, and $do(+r(\vec{t}), DB)$ will be equal to the database state $DB \cup \{r(\vec{t})\}$ for each database state DB. The resulting semantics of update formulas, however, will be slightly different from that one defined in Section 4.3: As we will see below, the sequential composition of transitions generates order dependencies and does not eliminate invalidated update requests. Consequently, the transitions may contain a lot of obsolete information, which is not relevant for the characterization of subsequent database states but has a negative impact on hypothetical reasoning techniques needed for the two-phase execution strategy of Section 8.1. Besides the illustration purposes, this is another reason for presenting the database-oriented ULTRA instance with a dedicated semantics based on update request sets. □

Example 4.31 [Robot World (Cont.)] In our robot example, the states in $\mathcal{S}$ will be the states of the robot and its environment, i.e. the blocks world.

We can define the set Σ of actions by

$$\Sigma := \mathcal{B}_{BU} \cup \{?q \mid q \in Pred_{DB}\}$$

and the mappings Log^{act} and Upd^{act} according to Remark 4.29. Consequently, we will obtain actions like $xstep(-1)$, $xstep(1)$, $pickup$, $?xpos$, $?empty$, etc. The actions prefixed by "?" are considered as read tags without side effects. They are relevant only for optimistic transaction processing strategies (cf. Section 5).

If the robot world is currently in state $s \in \mathcal{S}$, then $do(xstep(1), s)$ denotes the state that results from executing the operation $xstep(1)$, i.e. by moving the robot by one step in x-direction. Similarly, $do(pickup, s)$ denotes the state resulting from a $pickup$ operation. 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 hand of the robot.

Since the execution of a read tag is supposed not to change the state, the read tags are compatible with all other actions. But we can also find compatibilities between proper actions: the movement actions $xstep(-1)$, $xstep(1)$, $ystep(-1)$, and $ystep(1)$ are pair-wise compatible with each other. In contrast, the movement actions are not compatible with $pickup$ and $putdown$ (provided that the blocks world is not degenerated). See Table 2 for a matrix representation of the complete compatibility relation.

compatibility	$xstep(\ldots)$	$ystep(\ldots)$	$pickup$	$putdown$	$?\ldots$
$xstep(\ldots)$	+	+	−	−	+
$ystep(\ldots)$	+	+	−	−	+
$pickup$	−	−	+	−	+
$putdown$	−	−	−	+	+
$?\ldots$	+	+	+	+	+

Table 2: Compatibility relation in the robot example

In our example, the only observable items are the position indicators $xpos$ and $ypos$ as well as the $empty$ sensor of the robot. Assume, for instance, that the robot is empty in state s and a block is lying on the floor at the current position, then the semantics I_{DB} will state that $I_{DB}(s) \models empty$ and $I_{DB}(do(pickup, s)) \models \neg empty$ holds. Similarly, if the robot is at x-position 1 in state s, then $I_{DB}(s) \models xpos(1)$ and $I_{DB}(do(xstep(1), s) \models xpos(2)$ will hold.

Finally, let us discuss some independence properties between actions and ground DB atoms. Every DB atom is trivially independent of every read tag. Considering proper actions, a DB atom $xpos(\ldots)$ is independent of an action $ystep(\ldots)$, and $ypos(\ldots)$ is independent of $xstep(\ldots)$ analogously, as the movements in x- and y-direction behave orthogonally to each other. A move in y-direction, for instance, will never have an effect on the x-position of the robot. One could be tempted to think that $empty$ is independent of $xstep(1)$ and other movements, too, as the truth value of $empty$ is not directly affected by these actions. However, this property is not sufficient to guarantee the independence: the movement can have an *indirect* effect on $empty$ after the next $pickup$ action (provided that the blocks world is not degenerated), this violates the independence condition. A matrix representation of the complete independence relation can be found in Table 3. □

independence	$empty$	$xpos(\ldots)$	$ypos(\ldots)$
$xstep(\ldots)$	−	−	+
$ystep(\ldots)$	−	+	−
$pickup$	−	+	+
$putdown$	−	+	+
$?\ldots$	+	+	+

Table 3: Independence relation in the robot example

Based on the preliminaries defined above we will build a new instance of the ULTRA framework during the following sections.

4.4.2 Partially Ordered Multi-Sets of Actions

In this section we present the foundations of partially ordered multi-sets. The formal definitions are adapted from [Pra86]. Essentially, the classical notion of sets is generalized in two directions: elements can occur multiple times, and order dependencies between elements can be represented.

The main area of application for partially ordered multi-sets is the formal description of arbitrary concurrent and sequential processes. In this context, the elements are called actions, and the ordering relation, which does not need to be linear, specifies execution constraints: ordered actions are considered as sequential, while unordered actions are considered as concurrent. Partially ordered multi-sets can serve as a viable representation structure for the deferred transitions in a universal ULTRA instance, especially since the composition functions $\sqcup$ and $\oplus$ are easy to define with a commonly accepted semantics (see [Pra86] for more information about concurrent and sequential composition of processes).

Definition 4.32 [Labelled Partial Order] Let Σ be a set of actions. A *labelled partial order* over Σ is a triple $(V, \leq, \mu)$ where V is a set of *events*, $\leq$ is a partial order on V, and $\mu : V \to \Sigma$ is a *labelling function*. □

Definition 4.33 [Congruence] Two labelled partial orders $(V, \leq, \mu)$ and $(V', \leq', \mu')$ are *congruent*, iff there exists a bijective mapping $f : V \to V'$, such that $e_1 \leq e_2 \iff f(e_1) \leq' f(e_2)$ holds for arbitrary events $e_1, e_2 \in V$ and $\mu = \mu' \circ f$. □

Remark 4.34 Let a fixed set Σ of actions be given. Then the congruence is a well-defined equivalence relation on the set of labelled partial orders over Σ. f can be interpreted as a renaming function. □

Definition 4.35 [Partially Ordered Multi-Sets] Let Σ be a set of actions. The set of *partially ordered multi-sets (pomsets)* over Σ, denoted by $\Sigma^{\ddagger}$, is defined as the set of all labelled partial orders over Σ modulo congruence, i.e. the set containing one representative of each equivalence class w.r.t. congruence. □

In the following, we keep a set Σ of actions fixed and consider the pomsets in $\Sigma^{\ddagger}$. Every element of $\Sigma^{\ddagger}$ is denoted by a representative $[V, \leq, \mu]$. For the sake of brevity we denote the empty multi-set $[\emptyset, \emptyset, \emptyset]$ by $\emptyset$ and a singleton $[\{e\}, \{(e, e)\}, \{e \mapsto a\}]$ (with $a \in \Sigma$) by $\{a\}$.

Note that a partially ordered multi-set can be represented and visualized by a graph. The vertices V are marked with actions according to the function μ, the edges correspond to a relation on V that induces the ordering $\leq$ by reflexive-transitive closure. Some examples of pomsets are shown in Figure 7.

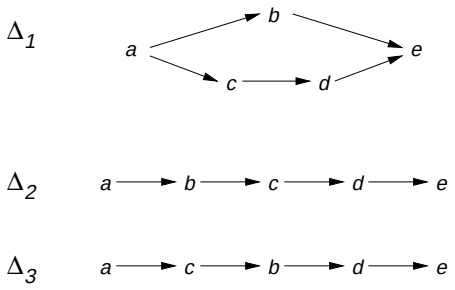

Figure 7: Partially ordered multi-sets

Next, we define the concurrent and the sequential composition of two partially ordered multi-sets.

Definition 4.36 [Concurrent Composition] The *concurrent composition* $\sqcup : \Sigma^\ddagger \times \Sigma^\ddagger \to \Sigma^\ddagger$ is defined by

$$\Delta_1 \sqcup \Delta_2 \;:=\; [\; V_1 \cup V_2 \;,\;\; \leq_1 \cup \leq_2 \;\;,\; \mu_1 \cup \mu_2 \;]$$

for two partially ordered multi-sets $\Delta_1 := [V_1, \leq_1, \mu_1]$ and $\Delta_2 := [V_2, \leq_2, \mu_2]$ over Σ, where V_1 and V_2 are chosen disjoint (w.l.o.g.). □

Definition 4.37 [Sequential Composition] The *sequential composition* $\oplus : \Sigma^\ddagger \times \Sigma^\ddagger \to \Sigma^\ddagger$ is defined by

$$\Delta_1 \oplus \Delta_2 \;:=\; [\; V_1 \cup V_2 \;,\;\; \leq_1 \cup (V_1 \times V_2) \cup \leq_2 \;\;,\; \mu_1 \cup \mu_2 \;]$$

for two partially ordered multi-sets $\Delta_1 := [V_1, \leq_1, \mu_1]$ and $\Delta_2 := [V_2, \leq_2, \mu_2]$ over Σ, where V_1 and V_2 are chosen disjoint (w.l.o.g.). □

The composition functions $\sqcup$ and $\oplus$ are well-defined. Since V_1 and V_2 are disjoint, it is easy to verify that $\leq_1 \cup \leq_2$ and $\leq_1 \cup (V_1 \times V_2) \cup \leq_2$ are partial orders on $V_1 \cup V_2$ and that $\mu_1 \cup \mu_2$ is the graph of a function $\mu : V_1 \cup V_2 \to \Sigma$. Thus, the composed objects are partially ordered multi-sets over Σ. Using standard techniques, it is possible to show the independence of the chosen representatives V_1 and V_2.

Lemma 4.38 [Monoid Properties] $(\Sigma^\ddagger, \sqcup, \emptyset)$ forms a commutative monoid, and $(\Sigma^\ddagger, \oplus, \emptyset)$ forms a monoid.

Proof: The associativity of $\sqcup$ and $\oplus$ is easy to see, when the representatives are chosen disjoint (w.l.o.g.): apply the associativity of the set union. The commutativity of $\sqcup$ follows directly from the commutativity of the set union. The empty multi-set $\emptyset$ (i.e. $[\emptyset, \emptyset, \emptyset]$) behaves neutral, because $\emptyset$ is the neutral element of the set union. Consequently, the assertions hold. □

Example 4.39 [Composition of Pomsets] Have a look at Figure 7. The pomset Δ_1 can be constructed as follows:

$$\Delta_1 := \{a\} \oplus (\{b\} \sqcup (\{c\} \oplus \{d\})) \oplus \{e\}$$

In contrast, Δ_2 and Δ_3 are constructed using the sequential composition $\oplus$ only. □

It is straight-forward to define a concurrent composition for multi-sets of pomsets.

Definition 4.40 [Concurrent Composition] Let M be a multi-set of partially ordered multi-sets over Σ. Choose (w.l.o.g.) a representation of M of the form $\{[V_i, \leq_i, \mu_i] \mid i \in I\}$, where the event sets V_i are disjoint. The *concurrent composition* $\bigsqcup$ for the multi-set M is defined by:

$$\bigsqcup M \;:=\; \left[\; \bigcup_{i \in I} V_i \;,\; \bigcup_{i \in I} \leq_i \;\;,\; \bigcup_{i \in I} \mu_i \;\right]$$

□

The concurrent composition $\bigsqcup$ for multi-sets of pomsets is well-defined. This can be shown in a similar fashion as for the $\sqcup$ function: the result is indeed a pomset and does not depend on the chosen representatives.

The following lemma states that the concurrent composition for multi-sets of pomsets fits with the concurrent composition of two pomsets.

Lemma 4.41 Let M be a multi-set of pomsets over Σ, and let $\Delta \in \Sigma^\ddagger$ be a pomset over Σ. Then the equality

$$\bigsqcup(\{\Delta\} \uplus M) = \Delta \sqcup \bigsqcup M$$

holds, where $\uplus$ denotes the union of multi-sets.

Proof: Choose (w.l.o.g.) a representation $\{[V_i, \leq_i, \mu_i] \mid i \in I\}$ of M as in Definition 4.40 such that the event sets V_i are disjoint. Further choose (w.l.o.g.) a representative $[V, \leq, \mu]$ of Δ such that V is disjoint from all V_i. We can reason as follows:

$$\begin{array}{ll} & \bigsqcup(\{\Delta\} \uplus M) \\ = & [\ V \cup \bigcup_{i\in I} V_i\ ,\ \ \leq \cup \bigcup_{i\in I} \leq_i\ \ ,\ \mu \cup \bigcup_{i\in I} \mu_i\ ] \\ = & [V, \leq, \mu]\ \sqcup\ [\ \bigcup_{i\in I} V_i\ ,\ \bigcup_{i\in I} \leq_i\ \ ,\ \bigcup_{i\in I} \mu_i\ ] \\ = & \Delta \sqcup \bigsqcup M \end{array}$$

□

In the following, we will define some special classes of partially ordered multi-sets. The classifications are relevant for the subsequent definition of the execution function $\oplus_E$.

Definition 4.42 [Finite Pomsets] A partially ordered multi-set $[V, \leq, \mu] \in \Sigma^\ddagger$ is *finite*, iff V is finite. □

Definition 4.43 [Linear Pomsets] A partially ordered multi-set $[V, \leq, \mu] \in \Sigma^\ddagger$ is *linear*, iff $\leq$ is a linear ordering relation on V, i.e. iff for arbitrary events $e, e' \in V$, $e \leq e'$ or $e' \leq e$ holds.

Let $\Delta, \Delta' \in \Sigma^\ddagger$ be pomsets. Δ' is called a *linearization* of Δ, if Δ' is a linear pomset and if Δ and Δ' are representable by $[V, \leq, \mu]$ and $[V, \leq', \mu]$, respectively, such that $\leq \subseteq \leq'$. Note that both $\leq$ and $\leq'$ are ordering relations on V. □

The properties of finiteness and linearity are well-defined: the independence of the chosen representative is easy to verify. Further, the linearization condition does not depend on a particular V: if the condition holds for representations in terms of some set V, it is clearly possible to find representations that satisfy the condition w.r.t. another set V' of the same cardinality as V.

Lemma 4.44 Let $\Delta_1, \Delta_2 \in \Sigma^\ddagger$ be arbitrary pomsets. Then the following equivalences hold:

$$\begin{array}{lll} \Delta_1 \text{ finite } \wedge\ \Delta_2 \text{ finite } & \Longleftrightarrow & \Delta_1 \sqcup \Delta_2 \text{ finite} \\ \Delta_1 \text{ finite } \wedge\ \Delta_2 \text{ finite } & \Longleftrightarrow & \Delta_1 \oplus \Delta_2 \text{ finite} \end{array}$$

Proof: The assertions follow directly from the properties of the set union. □

Lemma 4.45 Every partially ordered multi-set $\Delta \in \Sigma^\ddagger$ has at least one linearization $\Delta' \in \Sigma^\ddagger$.

A linear pomset $\Delta \in \Sigma^\ddagger$ has Δ as its unique linearization.

Proof: Let Δ be representable by $[V, \leq, \mu]$. Then there exists at least one linear ordering relation $\leq'$ that extends $\leq$. Define $\Delta' := [V, \leq', \mu]$.

If Δ is linear, it is indeed a linearization of itself. The uniqueness follows from the fact that there exists no ordering relation $\leq'$ on V different from $\leq$ such that $\leq \subseteq \leq'$ holds. □

Lemma 4.46 Let $\Delta_1, \Delta_1', \Delta_2, \Delta_2' \in \Sigma^\ddagger$ be pomsets, where Δ_1' is a linearization of Δ_1 and Δ_2' is a linearization of Δ_2. Then $\Delta_1' \oplus \Delta_2'$ is a linearization of both $\Delta_1 \sqcup \Delta_2$ and $\Delta_1 \oplus \Delta_2$.

Proof: Choose (w.l.o.g.) representatives $[V_1, \leq_1, \mu_1]$, $[V_1, \leq_1', \mu_1]$, $[V_2, \leq_2, \mu_2]$, and $[V_2, \leq_2', \mu_2]$ (for Δ_1 through Δ_2'), such that V_1 and V_2 are disjoint and the inclusions $\leq_1 \subseteq \leq_1'$ and $\leq_2 \subseteq \leq_2'$ hold. Recall that $\leq_1'$ and $\leq_2'$ are linear orders. Define:

$$\begin{array}{lcl} \leq_\sqcup & := & \leq_1 \cup \leq_2 \\ \leq_\oplus & := & \leq_1 \cup (V_1 \times V_2) \cup \leq_2 \\ \leq_\oplus' & := & \leq_1' \cup (V_1 \times V_2) \cup \leq_2' \end{array}$$

$\leq_\sqcup$, $\leq_\oplus$, and $\leq_\oplus'$ are the ordering relations (over $V_1 \cup V_2$) of $\Delta_1 \sqcup \Delta_2$, $\Delta_1 \oplus \Delta_2$, and $\Delta_1' \oplus \Delta_2'$, respectively.

It is straight-forward to show that $\leq_\oplus'$ is a linear ordering relation: two arbitrary events of $V_1 \cup V_2$ that are both contained in the set V_1 are ordered by $\leq_1'$ and thus by $\leq_\oplus'$, the same holds w.r.t. V_2 and $\leq_2'$, two events belonging to different sets are related by $V_1 \times V_2$ and thus ordered by $\leq_\oplus'$. Consequently, $\Delta_1' \oplus \Delta_2'$ is a linear pomset.

Further, the inclusions

$$\leq_\sqcup \subseteq \leq_\oplus \subseteq \leq_\oplus'$$

can be derived easily. This completes the proof of the main assertions. □

Lemma 4.47 Let $\Delta, \Delta' \in \Sigma^\ddagger$ be non-empty pomsets, where Δ' is a linearization of Δ. Choose (w.l.o.g.) representatives $[V, \leq, \mu]$ and $[V, \leq', \mu]$ for Δ and Δ', respectively, such that $\leq \subseteq \leq'$ holds. Let $e \in V$ be an arbitrary event, and define $V' := V \setminus \{e\}$. Then the pomset $[V', \leq'_{|V'}, \mu|_{V'}]$ is a linearization of the pomset $[V', \leq_{|V'}, \mu|_{V'}]$. (For an ordering relation $\leq$, we denote the ordering relation $\leq \cap (V' \times V')$ by $\leq_{|V'}$. This resembles the common notation $|$ for restricted functions, which is also used for μ.)

Proof: Since Δ and Δ' are pomsets over Σ, it is easy to verify that $[V', \leq_{|V'}, \mu|_{V'}]$ as well as $[V', \leq'_{|V'}, \mu|_{V'}]$ are pomsets over Σ, too. The linearization property follows directly from the linearity of the order $\leq'$ and the inclusion $\leq \subseteq \leq'$. □

Remark 4.48 [List Representation] Finite linear pomsets $[V, \leq, \mu] \in \Sigma^\ddagger$ are isomorphic to lists. Assume that $V = \{e_1, \ldots, e_n\}$ with $e_1 \leq \ldots \leq e_n$. Then $[V, \leq, \mu]$ can be denoted by the list $[\mu(e_1), \ldots, \mu(e_n)]$. The empty pomset $\emptyset$ can be denoted by the empty list $[\,]$ and a singleton $\{a\}$ can be denoted by $[a]$. Further, the sequential composition $\oplus$ corresponds to the list concatenation $\circ$. □

Example 4.49 [Finite Linear Pomsets] The pomsets shown in Figure 7 on page 54 are finite. In addition, Δ_2 and Δ_3 are linearizations of Δ_1: in both cases, the partial order of Δ_1 is extended to a linear order.

Being finite linear pomsets, Δ_2 and Δ_3 can also be represented by lists as follows:

$$\begin{array}{lcl} \Delta_2 & = & [a, b, c, d, e] \\ \Delta_3 & = & [a, c, b, d, e] \end{array}$$

□

4.4.3 Execution of Finite Pomsets

In this section we define what the execution of finite pomsets means. We first introduce a notion of consistency.

Definition 4.50 [Consistency] A partially ordered multi-set $[V, \leq, \mu] \in \Sigma^{\ddagger}$ is called *consistent*, if for arbitrary events $e, e' \in V$ the following holds:

$$e \not\leq e' \ \wedge \ e' \not\leq e \implies \mu(e) \text{ is compatible with } \mu(e')$$

□

The definition of consistency is well-defined. Note that a linear pomset is always consistent, as every pair of events is ordered by the linear ordering relation.

Lemma 4.51 Let $\Delta_1, \Delta_2 \in \Sigma^{\ddagger}$ be arbitrary pomsets. Then the following conditions hold:

$$\begin{array}{lcl} \Delta_1 \text{ consistent} \ \wedge \ \Delta_2 \text{ consistent} & \Longleftarrow & \Delta_1 \sqcup \Delta_2 \text{ consistent} \\ \Delta_1 \text{ consistent} \ \wedge \ \Delta_2 \text{ consistent} & \Longleftrightarrow & \Delta_1 \oplus \Delta_2 \text{ consistent} \end{array}$$

Proof: Choose (w.l.o.g.) representatives $[V_1, \leq_1, \mu_1]$, $[V_2, \leq_2, \mu_2]$, $[V_1 \cup V_2, \leq_{\sqcup}, \mu]$, $[V_1 \cup V_2, \leq_{\oplus}, \mu]$ (with $V_1 \cap V_2 = \emptyset$) for Δ_1, Δ_2, $\Delta_1 \sqcup \Delta_2$, and $\Delta_1 \oplus \Delta_2$, respectively, according to Definitions 4.36 and 4.37.

First, assume that $\Delta_1 \sqcup \Delta_2$ is consistent. Let $e, e' \in V_1$ be events that are unordered by $\leq_1$. Since e and e' cannot be related by $\leq_2$, they are unordered by $\leq_{\sqcup}$, too. Consequently, $\mu(e)$ and $\mu(e')$ must be compatible. As the labellings μ and μ_1 coincide on V_1, $\mu_1(e)$ and $\mu_1(e')$ are compatible with each other. So, the consistency condition is satisfied for Δ_1. The consistency of Δ_2 can be shown analogously.

Next, assume that $\Delta_1 \oplus \Delta_2$ is consistent. Let $e, e' \in V_1$ be events that are unordered by $\leq_1$. Since e and e' cannot be related by $V_1 \times V_2$ or $\leq_2$, they are unordered by $\leq_{\oplus}$, too. As in the case above $\mu_1(e)$ and $\mu_1(e')$ must be compatible. So, the consistency condition is satisfied for Δ_1. The consistency of Δ_2 can be shown analogously.

Finally, assume that both Δ_1 and Δ_2 are consistent. Let $e, e' \in V_1 \cup V_2$ be events that are unordered by $\leq_{\oplus}$. As two events from different sets V_i are related by $V_1 \times V_2$ and thus ordered by $\leq_{\oplus}$, both events e and e' must be either contained in V_1 or V_2. Recall that the inclusions $\leq_1 \subseteq \leq_{\oplus}$ and $\leq_2 \subseteq \leq_{\oplus}$ hold. Like in the cases above we can apply the consistency condition w.r.t. V_1 or V_2 to show that $\mu(e)$ and $\mu(e')$ must be compatible. This completes the proof of the consistency of $\Delta_1 \oplus \Delta_2$. □

Example 4.52 [Consistency] Figure 8 shows some pomsets taken from the robot domain. Being finite linear pomsets, Δ_1 and Δ_2 are also consistent pomsets. Δ_3, the concurrent composition of Δ_1 and Δ_2, is consistent, too, because the actions contained in Δ_1 are mutually compatible with the actions contained in Δ_2 (cf. Example 4.31). Δ_4 is an example of a pomset that is not consistent: although $xstep(1)$ is not compatible with *pickup*, the corresponding events are not ordered in Δ_4, which contradicts the consistency condition. □

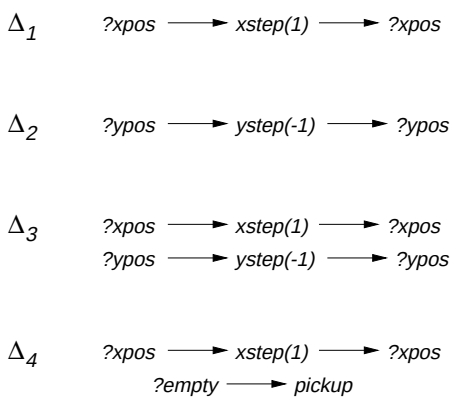

Figure 8: Pomsets in the robot domain

Next, we define the execution of finite linear pomsets. Referring to this natural definition, the execution of finite consistent pomsets will be defined subsequently.

Definition 4.53 [Execution of Linear Pomsets] The *execution* $\oplus_E^{lin}$ of a finite linear pomset $\Delta \in \Sigma^\ddagger$ w.r.t. a state $s \in \mathcal{S}$ is defined recursively by:

$$s \oplus_E^{lin} \Delta := \begin{cases} do(a,s) \oplus_E^{lin} \Delta', & \text{if } \Delta = \{a\} \oplus \Delta' \text{ for an action } a \in \Sigma \\ & \text{and a finite linear pomset } \Delta' \\ s, & \text{if } \Delta = \emptyset \end{cases}$$

□

The execution function $\oplus_E^{lin}$ is well-defined w.r.t. recursion and case distinction. This becomes obvious when the formal definition is rewritten into a list notation according to Remark 4.48.

Lemma 4.54 Let $\Delta \in \Sigma^\ddagger$ be a finite linear pomset containing at least two actions. Choose (w.l.o.g.) a representation $[V, \leq, \mu]$ for Δ such that $V = \{e_1, \ldots, e_n\}$ $(n \geq 2)$ and $e_1 \leq \ldots \leq e_n$. Let $i \in \{1, \ldots, n-1\}$ be an arbitrary but fixed index. Define the linear ordering relation $\leq'$ such that $e_1 \leq' \ldots \leq' e_{i-1} \leq' e_{i+1} \leq' e_i \leq' e_{i+2} \leq' \ldots \leq' e_n$ holds, and define the linear pomset $\Delta' := [V, \leq', \mu]$.

If $\mu(e_i)$ and $\mu(e_{i+1})$ are compatible with each other, then

$$s \oplus_E^{lin} \Delta = s \oplus_E^{lin} \Delta'$$

holds for every state $s \in \mathcal{S}$.

Proof: The assertion follows easily from Definitions 4.27 and 4.53. □

Example 4.55 Have a look at the pomsets Δ_2 and Δ_3 of Figure 7 on page 54. Δ_3 can be derived from Δ_2 by exchanging the second and the third event. Let us assume that the corresponding actions b and c are compatible with each other. Then $s \oplus_E^{lin} \Delta_2 = s \oplus_E^{lin} \Delta_3$ holds for every state $s \in \mathcal{S}$. In other words, the execution of Δ_2 always leads to the same final state as the execution of Δ_3. □

Lemma 4.56 Let $\Delta \in \Sigma^\ddagger$ be a finite consistent pomset, and let $\Delta_1, \Delta_2 \in \Sigma^\ddagger$ be linearizations of Δ. Then

$$s \oplus_E^{lin} \Delta_1 = s \oplus_E^{lin} \Delta_2$$

holds for every state $s \in \mathcal{S}$.

Proof: We prove the assertion by induction on the size of the pomset Δ, which can be uniquely defined as the cardinality of the event set V of a chosen representative $[V, \leq, \mu]$.

Base case: $V = \emptyset$

In this case, $\Delta = \emptyset$ and thus also $\Delta_1 = \emptyset$ and $\Delta_2 = \emptyset$. The assertion holds trivially.

Induction step: $V \neq \emptyset$

Choose (w.l.o.g.) representatives $[V, \leq, \mu]$, $[V, \leq_1, \mu]$, and $[V, \leq_2, \mu]$ for Δ, Δ_1, and Δ_2, respectively, such that the inclusions $\leq \subseteq \leq_1$ and $\leq \subseteq \leq_2$ hold. By the induction hypothesis, the assertion holds for pomsets having an event set with a cardinality less than that of V.

Let $e \in V$ be the least event w.r.t. the ordering relation $\leq_1$. e exists, as V is finite and $\leq_1$ is linear. e does not need to be the least event w.r.t. $\leq_2$, but in the first step of the proof we will show that $\leq_2$ (and thus Δ_2) can be modified such that e becomes the least event without changing the execution results determined by Δ_2. In the second step we will eliminate the event e and apply the induction hypothesis to the restricted pomsets. Note that these pomsets can be regarded as continuations after the execution of $\mu(e)$.

Let us construct the linear ordering relation $\leq_3$ from the given ordering $\leq_2$ by replacing each order dependency between e and a corresponding other event e' in a way such that $e \leq_3 e'$ holds. Note that this preserves the order of the events contained in $V \setminus \{e\}$, while e becomes the least event. Define the new pomset $\Delta_3 := [V, \leq_3, \mu]$.

We are going to show that the execution of Δ_2 always leads to the same state as the execution of Δ_3. Let $e_1, \ldots, e_m \in V \setminus \{e\}$ be the events for which $e_i \leq_2 e$ holds. If such events do not exist, then e is also the least element of $\leq_2$, and Δ_2 equals Δ_3. Otherwise, action $\mu(e)$ must be compatible with each action $\mu(e_i)$ ($i \in \{1, \ldots, m\}$): since $e \leq_1 e_i$ as well as $e_i \leq_2 e$ holds, e and e_i cannot be ordered by $\leq$, and the desired compatibility follows from the consistency condition w.r.t. Δ. Applying Lemma 4.54 inductively, one can exploit the compatibilities between $\mu(e)$ and $\mu(e_i)$ ($i \in \{1, \ldots, m\}$) and prove that

$$s \oplus_E^{lin} \Delta_2 = s \oplus_E^{lin} \Delta_3$$

holds for arbitrary states $s \in \mathcal{S}$. Note that in each step, the order of e and a neighbour e_i is exchanged. This way, $\leq_3$ is derived from $\leq_2$.

Let $V' := V \setminus \{e\}$, and define the following pomsets:

$$\begin{array}{lcl} \Delta' & := & [V', \leq_{|V'}, \mu|_{V'}] \\ \Delta'_1 & := & [V', \leq_{1|V'}, \mu|_{V'}] \\ \Delta'_2 & := & [V', \leq_{2|V'}, \mu|_{V'}] \end{array}$$

By Lemma 4.47, this is well-defined, and Δ'_1 and Δ'_2 are both linearizations of Δ'. Furthermore, Δ' is consistent, and we can apply the induction hypothesis. It is easy to see that the equalities $\{\mu(e)\} \oplus \Delta'_1 = \Delta_1$ and $\{\mu(e)\} \oplus \Delta'_2 = \Delta_3$ hold.

Now we are ready to prove the main assertion. Choose an arbitrary state $s \in \mathcal{S}$. We can reason as follows:

$$\begin{array}{ll} & s \oplus_E^{lin} \Delta_1 \\ = & do(\mu(e), s) \oplus_E^{lin} \Delta'_1 \\ = & do(\mu(e), s) \oplus_E^{lin} \Delta'_2 \\ (IH) & \\ = & s \oplus_E^{lin} \Delta_3 \\ = & s \oplus_E^{lin} \Delta_2 \\ (see\ above) & \end{array}$$

□

Definition 4.57 [Execution of Consistent Pomsets] The *execution* $\oplus_E$ of a consistent finite pomset $\Delta \in \Sigma^\ddagger$ w.r.t. a state $s \in \mathcal{S}$ is defined by

$$s \oplus_E \Delta := s \oplus_E^{lin} \Delta'$$

where Δ' is an arbitrary linearization of Δ. □

The definition of the execution $\oplus_E$ is well-defined due to Lemmata 4.45 and 4.56. Note that $\oplus_E^{lin}$ and $\oplus_E$ coincide for finite linear pomsets.

Lemma 4.58 Let $s \in \mathcal{S}$ be a state and $\Delta_1, \Delta_2 \in \Sigma^\ddagger$ be finite consistent pomsets. Then

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

holds.

Proof: Let $\Delta'_1 \in \Sigma^\ddagger$ and $\Delta'_2 \in \Sigma^\ddagger$ be some linearizations of Δ_1 and Δ_2, respectively. By Lemma 4.46, $\Delta'_1 \oplus \Delta'_2$ is a linearization of $\Delta_1 \oplus \Delta_2$. Note that $\Delta'_1 \oplus \Delta'_2$ is a finite linear pomset. Using induction, it is easy to show that

$$(s \oplus_E^{lin} \Delta'_1) \oplus_E^{lin} \Delta'_2 = s \oplus_E^{lin} (\Delta'_1 \oplus \Delta'_2)$$

holds. This becomes more obvious when considering the list representation of finite linear pomsets (see Remark 4.48).

Now we can reason as follows:

$$\begin{array}{lcl} (s \oplus_E \Delta_1) \oplus_E \Delta_2 & = & (s \oplus_E^{lin} \Delta'_1) \oplus_E^{lin} \Delta'_2 \\ = \quad s \oplus_E^{lin} (\Delta'_1 \oplus \Delta'_2) & = & s \oplus_E (\Delta_1 \oplus \Delta_2) \end{array}$$

□

Let us finally discuss the execution semantics in more detail. A pomset can be considered as the representation of a process, where the actions are performed respecting the order dependencies. While ordered actions must be performed sequentially, unordered actions can be performed without any synchronization. When atomic actions are considered, this concurrency corresponds to an execution in any order. Consequently, the executions of a pomset can be modeled by the (sequential) executions of its linearizations. The deferred update semantics of the ULTRA approach requires that it is possible to reason about a state that will finally be reached from a given state by executing a consistent transition. This imposes two constraints: the execution must terminate, and it must lead to a uniquely defined state. Lemma 4.56 shows that finite consistent pomsets as defined above satisfy the desired properties, such that we can define the execution function $\oplus_E$ for them. Note that although the execution of a finite consistent pomset may be non-deterministic at the operational level, it is deterministic w.r.t. the ULTRA semantics, where only the resulting final state is relevant (cf. also Example 4.62 below). The execution semantics of ULTRA should also be contrasted with the execution semantics of independent transactions [BHG87, BN97, GR93]. Commonly, one accepts concurrent executions of multiple transactions as long as they lead to final states that would also be reached if the transactions were executed one after the other. However, the transactions do not characterize a unique final state. The state that is actually reached depends on the operational settings.

4.4.4 The ULTRA Instance based on Pomsets

Now we are ready to obtain a new instance of the ULTRA framework. Let $\mathcal{S}$, Σ, *do*, I_{DB}, Log^{act}, and Upd^{act} be given as in Section 4.4.1. It should be recalled that an ULTRA instance is defined by a transition system and the mappings I_{DB}, *Log*, and *Upd*. To construct the transition system, we simply refer to partially ordered multi-sets over Σ together with their composition and execution semantics. *Log* and *Upd* are straight-forward to define.

Definition 4.59 [Transitions] The set $\mathcal{T}$ of *transitions* is defined as the set $\Sigma^{\ddagger}$ of all partially ordered multi-sets over Σ.

The subset $\mathcal{T}_{Cons} \subseteq \mathcal{T}$ is defined as the set of finite consistent pomsets over Σ.

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

Note that the consistency notion in the ULTRA instance is more restrictive than the general consistency notion of pomsets. In ULTRA, we consider infinite pomsets as inconsistent, too, because the execution function $\oplus_E$ is not defined for them. Intuitively, the execution of an infinite pomset will not terminate, such that no final state to reason about will be reached. So, we exclude the infinite consistent pomsets from $\mathcal{T}_{Cons}$.

Definition 4.60 [Transition Assignments] We define the mapping *Log* by

$$Log(q(\vec{t})) := \{Log^{act}(q(\vec{t}))\}$$

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

$$Upd(u(\vec{t})) := \{Upd^{act}(u(\vec{t}))\}$$

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

The transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ for the new ULTRA instance can already be considered as completely specified, because the missing parameters $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$ have been defined in Sections 4.4.2 and 4.4.3. Nevertheless, we must verify the algebraic properties required in Definition 4.2. This will be done in the following theorem.

Theorem 4.61 [Algebraic Properties] The algebraic properties of Definition 4.2 hold for the transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ defined for arbitrary external operations.

Proof: The assertion follows from the definitions of $\oplus_E$, $\sqcup$, $\bigsqcup$, and $\oplus$ in this section. Note that most of the properties have already been proved above.

Properties 1 and 2, which state that $(\mathcal{T}, \sqcup, \Delta_\varepsilon)$ and $(\mathcal{T}, \oplus, \Delta_\varepsilon)$ form monoids, have been shown in Lemma 4.38.

Property 3 concerning consistency aspects follows directly from Lemmata 4.44 and 4.51.

Property 4 (a) has been shown in Lemma 4.58, while property 4 (b) holds by Definition 4.53.

Property 5 (a) has been shown in Lemma 4.41. The proof of property 5 (b) is simple:

$$\bigsqcup \emptyset = [\ \bigcup \emptyset, \bigcup \emptyset, \bigcup \emptyset\ ] = [\emptyset, \emptyset, \emptyset] = \Delta_\varepsilon$$

□

In Theorem 4.61 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 4.2 and get a particular semantics for the ULTRA language. Let us illustrate this semantics using the robot example.

Example 4.62 [Robot World (Cont.)] Recall Examples 3.20 and 4.31 and consider the update formula

$$\varphi :\equiv [\ xpos(1)\ :\ xstep(1)\ :\ xpos(2)\ ]\ ,\ [\ ypos(1)\ :\ ystep(-1)\ :\ ypos(0)\ ]$$

which specifies a robot movement from position $(1,1)$ to position $(2,0)$. Note that φ has been chosen according to the definition of the *move* operation (cf. Example 3.20 and Appendix C), where the actual position of the robot is to be checked regularly. In this example, all variables that should be bound during an evaluation have been replaced by some suitable values.

Let us assume that the robot is indeed at position $(1,1)$ in state s_0. Then $I_{DB}(s_0) \models xpos(1)$ and $I_{DB}(s_0) \models ypos(1)$ holds, further $I_{DB}(do(xstep(1), s_0)) \models xpos(2)$ and $I_{DB}(do(ystep(-1), s_0)) \models ypos(0)$.

We can now interpret the formula φ using Definition 4.9. The assertions derived below hold regardless of any specific interpretation I_{UP} of definable update atoms. (To keep the example short, we use the list notation for finite linear pomsets with more than one element.)

By cases (DB) and (BU), the assertions

$$\begin{array}{lcl} (\emptyset, \{?xpos\}) & \in & I(xpos(1)) \\ (\{?xpos\}, \{xstep(1)\}) & \in & I(xstep(1)) \\ ([?xpos, xstep(1)], \{?xpos\}) & \in & I(xpos(2)) \end{array}$$

hold. Applying case (SCj) twice we can deduce:

$$(\emptyset, [?xpos, xstep(1), ?xpos]) \in I(xpos(1) : xstep(1) : xpos(2))$$

Recall that $[?xpos, xstep(1), ?xpos]$ is equal to Δ_1 in Figure 8 on page 59. It is possible to show that Δ_2 is an analogous solution for the subgoal $ypos(1) : ystep(-1) : ypos(0)$. Consequently, we know:

$$\begin{array}{lcl} (\emptyset, \Delta_1) & \in & I(xpos(1) : xstep(1) : xpos(2)) \\ (\emptyset, \Delta_2) & \in & I(ypos(1) : ystep(-1) : ypos(0)) \end{array}$$

Next, recall from Example 4.52 that the concurrent composition of Δ_1 and Δ_2 is the (consistent) pomset Δ_3 shown in Figure 8. Thus, applying case (CCj) we can finally conclude:

$$(\emptyset, \Delta_3) \in I(\varphi)$$

Intuitively, this result states that whenever the robot world is in state s_0 (represented by $\emptyset$), the formula φ has a solution with a side effect expressed by the pomset Δ_3 of Figure 8.

It should be noted that the operation specified by the formula φ is a deterministic operation. In particular, the solution Δ_3 is uniquely defined for the current state represented by $\emptyset$. Consequently, there is no need for a choice when the materialization of a solution should take place. However, the materialization itself can be performed non-deterministically, i.e. the robot can move on different paths from position $(1, 1)$ to position $(2, 0)$. The system component responsible for the materialization can autonomously and independently of the ULTRA semantics decide to move the robot via $(1, 0)$ or $(2, 1)$. □

4.5 Other Instantiations of the Framework

In Sections 4.3 and 4.4 we have demonstrated how the semantical parts of the ULTRA framework can be instantiated. In this section we briefly discuss other possibilities to create instances. We do not develop full-fledged solutions but present some ideas of what else can be done with the generic ULTRA concept.

4.5.1 Cost Calculation for Complex Operations

A quite different instance of the ULTRA framework that serves rather for cost calculation than for update specification can be derived from well-known monoid properties in the domain of the natural numbers (extended by infinity). We define the states $\mathcal{S}$ and the consistent transitions $\mathcal{T}_{Cons}$ as the natural numbers and add ∞ to the set $\mathcal{T}$ as a non-consistent transition (note that ∞ is needed only for formal reasons). The interesting part lies in the composition of the transitions, where we use the functions sum and maximum.

Definition 4.63 We define $\mathcal{S}$, $\mathcal{T}$, $\mathcal{T}_{Cons}$, and Δ_ε as follows:

$$\begin{array}{lcl} \mathcal{S} & := & \mathbb{N} \\ \mathcal{T} & := & \mathbb{N} \cup \{\infty\} \\ \mathcal{T}_{Cons} & := & \mathbb{N} \\ \Delta_\varepsilon & := & 0 \end{array}$$

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

$$s \oplus_E \Delta := s + \Delta$$

The concurrent composition of two transitions $\Delta_1, \Delta_2 \in \mathcal{T}$ is defined either by

$$\Delta_1 \sqcup \Delta_2 := \begin{cases} \Delta_1 + \Delta_2, & \text{if } \Delta_1, \Delta_2 \in \mathbb{N} \\ \infty, & \text{otherwise} \end{cases}$$

or (alternatively) by

$$\Delta_1 \sqcup \Delta_2 := \begin{cases} max\{\Delta_1, \Delta_2\}, & \text{if } \Delta_1, \Delta_2 \in \mathbb{N} \\ \infty, & \text{otherwise.} \end{cases}$$

In analogy, we define the concurrent composition $\bigsqcup$ for a multi-set T of transitions by $\sum$ respectively max (suitably extended to $\mathbb{N} \cup \{\infty\}$).

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

$$\Delta_1 \oplus \Delta_2 := \begin{cases} \Delta_1 + \Delta_2, & \text{if } \Delta_1, \Delta_2 \in \mathbb{N} \\ \infty, & \text{otherwise} \end{cases}$$

□

Proposition 4.64 The transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ (in both versions of Definition 4.63) satisfies the algebraic properties of Definition 4.2.

Proof: The proof follows easily from the properties of the functions + and max in the domain of natural numbers. □

In Proposition 4.64 we have shown that Definition 4.63 yields valid structures for building instances of the ULTRA framework. These instances are not suitable to capture update semantics, but they can be used to compute costs for complex update operations. In this setting, the transitions can be considered as cost values. If *Log* and *Upd* assign cost values for retrieval operations and basic update operations, respectively, the interpretation $I(\varphi)$ of a ground update formula φ contains pairs the second component of which expresses possible execution costs for φ (see Example 4.65 below). We assume that sequential operations always accumulate costs, while for concurrent operations we provide one model (sum semantics) that accumulates costs – applicable for resource calculations – and one model (maximum semantics) that chooses the "critical" value – applicable for time calculations in a parallel execution environment. However, other cost models can be defined as well. They only must lead to legal transition systems. Also the Cartesian product of cost models is definable with the technique shown in Section 4.5.2.

In the cost-oriented ULTRA instance, the states can be considered as accumulated costs, too. If the Herbrand universe $\mathcal{U}$ contains representatives for natural numbers and $Pred_{DB}$ provides a DB predicate *costs*, we can define a state interpretation I_{DB} such that

$$I_{DB}(s) \models costs(t) \iff t \text{ represents the value } s$$

holds for each state $s \in \mathcal{S}$. In this case, it is possible to reason about accumulated costs in a (hypothetical) current state, since the value of *costs* can be asked within the update formulas.

We want to explain a cost calculation using an update goal of the robot example.

Example 4.65 Consider the update formula

$$\varphi :\equiv [\ xstep(1)\ :\ xstep(1)\ ]\ ,\ ystep(-1)$$

specifying a robot movement in Example 3.20. Let Upd assign the transition $\Delta := 1$ to every ground basic update atom.

By Definition 4.9, cases (BU), (CCj), and (SCj), depending on the cost model above either

$$(0,3) \in I(\varphi) \qquad \text{(sum semantics for } \sqcup\text{)}$$

or

$$(0,2) \in I(\varphi) \qquad \text{(maximum semantics for } \sqcup\text{)}$$

holds. Informally speaking, the execution of φ induces resource-oriented costs of value 3 and time-oriented costs of value 2 (provided that a parallel execution of the movements in x- and y-direction can actually take place). □

The cost model as described above has two major disadvantages: First, despite the cost value, it does not incorporate a notion of system states. Thus, it cannot handle intermediate states that would be reached by performing basic operations. This problem, however, can be solved by combining the cost model with another ULTRA instance, e.g. the one of Section 4.4. See Section 4.5.2 for more details on the composition of multiple instances. Secondly, the cost model above does not respect costs that arise from searching a solution in an operational semantics. Costs for retrieval and (performed and undone) basic operations on failing branches are not modeled. These costs appear "backtracked", too. Only in an operational semantics that is purely based on deferred updates and where the hypothetical reasoning does not induce extra costs, the operational costs (for evaluation and materialization) coincide with the semantically specified costs. In this case, the cost values can serve as a means to find optimal choices for non-determinism. For instance, the materialization of solutions with minimal cost values could be favoured. Note that the combination with another ULTRA instance is again necessary to treat the update semantics.

4.5.2 Combination of Instances

This section is devoted to the composition of already defined instances of the ULTRA framework. Two or more transition systems can be combined in analogy to the Cartesian product of sets. This is shown in the following proposition. Essentially, the local transition systems are combined to a global transition system, where the components remain independent of each other. The resulting ULTRA semantics describes global changes in terms of local changes but cannot impose synchronization constraints between multiple local components. If such a synchronization is necessary, the straightforward composition is not sufficient, and one will have to put more effort into the development of a global transition system.

Proposition 4.66 For $i \in \{1, \ldots, n\}$, let each $(\mathcal{S}^{(i)}, \mathcal{T}^{(i)}, \mathcal{T}^{(i)}_{Cons}, \oplus^{(i)}_E, \Delta^{(i)}_\varepsilon, \sqcup^{(i)}, \bigsqcup^{(i)}, \oplus^{(i)})$ be a transition system according to Definition 4.2.

Define $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{T}_{Cons}$ as Cartesian products, i.e.

$$\begin{array}{lcl} \mathcal{S} & := & \mathcal{S}^{(1)} \times \ldots \times \mathcal{S}^{(n)}, \\ \mathcal{T} & := & \mathcal{T}^{(1)} \times \ldots \times \mathcal{T}^{(n)}, \\ \mathcal{T}_{Cons} & := & \mathcal{T}_{Cons}^{(1)} \times \ldots \times \mathcal{T}_{Cons}^{(n)}, \end{array}$$

define $\oplus_E$, Δ_ε, $\sqcup$, $\bigsqcup$, and $\oplus$ component-wise, i.e.

$$\begin{array}{lcl} (s^{(1)}, \ldots, s^{(n)}) \oplus_E (\Delta^{(1)}, \ldots, \Delta^{(n)}) & := & (s^{(1)} \oplus_E^{(1)} \Delta^{(1)}, \ldots, s^{(n)} \oplus_E^{(n)} \Delta^{(n)}), \\ \Delta_\varepsilon & := & (\Delta_\varepsilon^{(1)}, \ldots, \Delta_\varepsilon^{(n)}), \\ (\Delta_1^{(1)}, \ldots, \Delta_1^{(n)}) \sqcup (\Delta_2^{(1)}, \ldots, \Delta_2^{(n)}) & := & (\Delta_1^{(1)} \sqcup^{(1)} \Delta_2^{(1)}, \ldots, \Delta_1^{(n)} \sqcup^{(n)} \Delta_2^{(n)}), \\ \bigsqcup \{(\Delta_j^{(1)}, \ldots, \Delta_j^{(n)}) \mid j \in J\} & := & (\bigsqcup^{(1)} \{\Delta_j^{(1)} \mid j \in J\}, \ldots, \bigsqcup^{(n)} \{\Delta_j^{(n)} \mid j \in J\}), \\ (\Delta_1^{(1)}, \ldots, \Delta_1^{(n)}) \oplus (\Delta_2^{(1)}, \ldots, \Delta_2^{(n)}) & := & (\Delta_1^{(1)} \oplus^{(1)} \Delta_2^{(1)}, \ldots, \Delta_1^{(n)} \oplus^{(n)} \Delta_2^{(n)}). \end{array}$$

Then $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ is also a well-defined transition system.

Proof: The assertion can be shown easily: since the algebraic properties hold for the components and the functions $\oplus_E$, Δ_ε, $\sqcup$, $\bigsqcup$, and $\oplus$ are defined by delegation to the components, the algebraic properties carry over to the composite system. The formal proofs are straight-forward. □

An ULTRA instance must further provide the state interpretation I_{DB} and the transition assignments *Log* and *Upd*. Note that the ULTRA semantics does not impose any restrictions. *Log* and *Upd* can be defined from given $Log^{(i)}$ and $Upd^{(i)}$ on component basis similar to Δ_ε. The definition of I_{DB}, however, is not that easy, since every state needs a consistent interpretation. For instance, if one component says $I_{DB}^{(i)}(s^{(i)}) \models A$ and another component says $I_{DB}^{(j)}(s^{(j)}) \models \neg A$, then it is questionable whether A should be true or false in the global state s. Cautious definitions on component basis could assign the truth value "unknown" to such atoms. A more practical approach would partition the Herbrand base $\mathcal{B}$ (probably at the predicate-level) and use a dedicated component of the state for each class of atoms.

Example 4.67 For simplicity, let us integrate the two cost models of Section 4.5.1 into one ULTRA instance. Recall the update formula

$$\varphi :\equiv [\ xstep(1)\ :\ xstep(1)\ ]\ ,\ ystep(-1)$$

of Example 4.65. We assume that *Upd* is defined component-wise using the definition of *Upd* in the example above and thus assigns the value $(1, 1)$ to each basic update atom.

In the combined model, where the first component uses the sum semantics for $\sqcup$ and the second component uses the maximum semantics for $\sqcup$, the statement

$$((0, 0), (3, 2)) \in I(\varphi)$$

holds.

If we want to refer to costs within the update formulas, it is sensible to have two different DB predicates *resource_costs* and *time_costs*, whose interpretation is defined w.r.t. the first and the second component of the state, respectively. □

The component-wise definition of *Log* and *Upd* together with the partition-wise definition of I_{DB} is a viable method especially in the case that multiple systems that have been defined for disjoint sets of symbols must be integrated using a common Herbrand base $\mathcal{B}$ and a common basic update base $\mathcal{B}_{BU}$: The given partition of $\mathcal{B}$ can be used to define I_{DB} consistently. The logging and update transition assignments for the local systems must be extended to the full sets of atoms by assigning neutral transitions, then they can be composed on component basis. The resulting semantics of update formulas expresses an independent, unsynchronized composition of the local systems.

4.5.3 Transitions and their Consistency

The set $\mathcal{T}_{Cons}$ of consistent transitions (a subset of $\mathcal{T}$) is an important component of each transition system. Under the conditions of the following proposition, it is possible to derive a new instance of the ULTRA framework from a given instance by strengthening the consistency property.

Proposition 4.68 Let an instance of the ULTRA framework be given, i.e. a transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ together with the mappings I_{DB}, *Log*, and *Upd*. Let Φ be a property on the set $\mathcal{T}$ of transitions, i.e. Φ may be either true or false for a transition $\Delta \in \mathcal{T}$. Define

$$\mathcal{T}'_{Cons} := \{\Delta \in \mathcal{T}_{Cons} \mid \Phi(\Delta)\} = \{\Delta \in \mathcal{T} \mid \Delta \text{ consistent } \wedge\ \Phi(\Delta)\}.$$

Further, let Φ satisfy the following algebraic properties:

1. The following properties holds for the neutral transition and the transition assignments:

   (a) $\Phi(\Delta_\varepsilon)$
   (b) $\Phi(\Delta)$ holds for all $\Delta \in \mathit{Log}(\mathcal{B})$.
   (c) $\Phi(\Delta)$ holds for all $\Delta \in \mathit{Upd}(\mathcal{B}_{BU})$.

2. For arbitrary $\Delta_1, \Delta_2 \in \mathcal{T}$, the following holds:

   (a) $\Phi(\Delta_1) \wedge \Phi(\Delta_2) \Longleftarrow \Phi(\Delta_1 \sqcup \Delta_2)$
   (b) $\Phi(\Delta_1) \wedge \Phi(\Delta_2) \Longrightarrow \Phi(\Delta_1 \oplus \Delta_2)$
   (c) $\Phi(\Delta_2) \Longleftarrow \Phi(\Delta_1 \oplus \Delta_2)$

   In other words, property 3 of Definition 4.2 holds for Φ analogously.

Then $(\mathcal{S}, \mathcal{T}, \mathcal{T}'_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ together with I_{DB}, *Log*, and *Upd* forms an ULTRA instance. (Note that $\oplus_E$, *Log*, and *Upd* are implicitly restricted to $\mathcal{T}'_{Cons}$.)

Proof: The assertion can be proved easily: the properties of Definition 4.2 are respected and the restrictions of the mappings $\oplus_E$, *Log*, and *Upd* are well-defined. □

Corollary 4.69 Let an ULTRA instance and a property Φ be given as in an Proposition 4.68. Define

$$\mathcal{T}''_{Cons} := \{\Delta \in \mathcal{T} \mid \Phi(\Delta)\}.$$

If Φ satisfies the conditions of Proposition 4.68 and further entails the given consistency property, i.e.

$$\mathcal{T}''_{Cons} \subseteq \mathcal{T}_{Cons}$$

holds, then $(\mathcal{S}, \mathcal{T}, \mathcal{T}''_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \bigsqcup, \oplus)$ together with I_{DB}, Log, and Upd forms an ULTRA instance. This instance does not need to refer to the given consistency property anymore.

Proof: The assertion follows directly from Proposition 4.68. Note that $\mathcal{T}''_{Cons}$ equals $\mathcal{T}'_{Cons}$ due to the additional condition. □

In the following example, we give an outline of how to apply Corollary 4.69. It is much easier to derive a new ULTRA instance from a given one than to develop it from scratch.

Example 4.70 Recall the ULTRA instance for external operations that has been presented in Sections 3.3 and 4.4. In particular, recall Definition 4.50, where the consistency of transitions has been defined using the notion of compatibility. In a real life setting, this *semantical* notion of compatibility may not be tractable due to computational constraints. This will inhibit an effective operational semantics. It would thus be desirable to express the semantics of formulas using a notion of *syntactical compatibility*, which is explicitly provided as an additional parameter, namely a compatibility matrix. We require that a given compatibility matrix is correct w.r.t. the state semantics, i.e. that syntactical compatibility entails semantical compatibility.

If we now redefine that a partially ordered multi-set $[V, \leq, \mu] \in \Sigma^\ddagger$ is consistent, iff for arbitrary events $e, e' \in V$,

$$e \not\leq e' \ \wedge \ e' \not\leq e \implies \mu(e) \text{ is syntactically compatible with } \mu(e')$$

holds, then we obtain a more restrictive ULTRA instance. The new consistency property (which corresponds to the parameter Φ) satisfies the conditions of Proposition 4.68 and entails the old consistency property, such that Corollary 4.69 can be applied.

The replacement of semantical properties by syntactical properties is very common in the field of database transactions. For instance, there exist two notions of serializability [BHG87]: view serializability and conflict serializability. While the former property is defined in terms of observable state changes, i.e. at the semantical level, the latter property is defined using an explicit notion of conflicts between the basic operations. Unfortunately, the chosen conflict model may specify conflicts between operations that are actually compatible with each other. In this case, there exist execution schedules that are view serializable but not conflict serializable. The significantly greater efficiency of the methods to check and maintain conflict serializability, however, outweighs this minor drawback. □

4.6 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} (w.r.t. s_0). The results are shown for the ULTRA framework and can thus be applied to every particular instance.

As stated in Section 4.1, 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 4.71 [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. □

Lemma 4.72 An interpretation I_{UP} is a model of an update program P_{UP}, iff for every ground instance $U \to p(\vec{t})$ of a rule $r \in P_{UP}$

$$I_{UP}(U) \subseteq I_{UP}(p(\vec{t}))$$

holds.

Proof: The assertion follows directly from cases (Impl) and (Univ) of Definition 4.9. □

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}. In this sense, Definition 4.71 requires 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 4.2). According to Lemma 4.72 this holds, iff $I_{UP}(U) \subseteq I_{UP}(p(\vec{t}))$ holds for every ground instance $U \to p(\vec{t})$ 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.

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

Definition 4.73 $\mathcal{I}$ 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, we define $I_\bot, I_\top : \mathcal{B}_{DU} \to 2^{\mathcal{T}_{Cons} \times \mathcal{T}_{Cons}}$ by

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

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

Definition 4.74 [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 4.75 [Lattice of Interpretations] The set $\mathcal{I}$ of all interpretations together with its ordering relation $\leq$ (cf. Definition 4.74) forms a complete lattice.

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

Furthermore, for arbitrary non-empty sets $\mathcal{J} \subseteq \mathcal{I}$ of interpretations, the equalities

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

and

$$lub(\mathcal{J})(p(\vec{t})) = \bigcup_{I \in \mathcal{J}} I(p(\vec{t}))$$

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

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

Remark 4.76 [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 Lemma 4.72. □

Definition 4.77 [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 4.78 Let $I_1, I_2 \in \mathcal{I}$ be interpretations over $\mathcal{B}_{DU}$ with $I_1 \leq I_2$. Then

$$I_1(U) \subseteq I_2(U)$$

holds for every ground update goal U.

Proof: We prove the assertion by structural induction. In each case shown below, we choose arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ and show the implication

$$(\Delta_C, \Delta) \in I_1(U) \implies (\Delta_C, \Delta) \in I_2(U).$$

Base cases:

Essentially, we apply Definition 4.9.

1. DB literal

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

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(q(\vec{t})) \\ \Longrightarrow & I_{DB}(s_0 \oplus_E \Delta_C) \models q(\vec{t}) \text{ and } \Delta = Log(q(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(q(\vec{t})) \end{array}$$

2. NOP literal

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(NOP) \\ \Longrightarrow & \Delta = \Delta_\varepsilon \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(NOP) \end{array}$$

3. Basic update atom

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(u(\vec{t})) \\ \Longrightarrow & \Delta = Upd(u(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(u(\vec{t})) \end{array}$$

4. Definable update atom

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(p(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(p(\vec{t})) \end{array}$$

   The assertion follows directly from the precondition $I_1 \leq I_2$.

Induction step:

By the induction hypothesis, $I_1(\varphi) \subseteq I_2(\varphi)$ holds for any direct proper subgoal φ of the composite goals analyzed in the following. Using Definition 4.9 we can reason as follows.

1. Concurrent conjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\varphi, \psi) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I_1(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I_1(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I_2(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I_2(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\varphi, \psi) \end{array}$$

2. Sequential conjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\varphi : \psi) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I_1(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I_1(\psi) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I_2(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I_2(\psi) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\varphi : \psi) \end{array}$$

3. Disjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\varphi \vee \psi) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_1(\varphi) \text{ or } (\Delta_C, \Delta) \in I_1(\psi) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\varphi) \text{ or } (\Delta_C, \Delta) \in I_2(\psi) \\ {\scriptstyle (IH)} & \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\varphi \vee \psi) \end{array}$$

4. Existential quantification

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

   $$I_1(\varphi[\vec{X} / \vec{t}]) \subseteq I_2(\varphi[\vec{X} / \vec{t}])$$

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

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\exists \vec{X}\, \varphi) \\ \Longrightarrow & \text{there exists a ground term tuple } (\vec{t}) \in \mathcal{U}^n \text{ such that :} \\ & (\Delta_C, \Delta) \in I_1(\varphi[\vec{X} / \vec{t}]) \\ \Longrightarrow & \text{there exists a ground term tuple } (\vec{t}) \in \mathcal{U}^n \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta) \in I_2(\varphi[\vec{X} / \vec{t}]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\exists \vec{X}\, \varphi) \end{array}$$

5. Bulk quantification

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

   $$I_1(\varphi[\vec{X} / \vec{t}]) \subseteq I_2(\varphi[\vec{X} / \vec{t}])$$

   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:

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\#\, \vec{X}\, [A \mapsto \varphi]) \\ \Longrightarrow & \Delta = Log(A[\vec{X} / \vec{all}]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\#\, \vec{X}\, [A \mapsto \varphi]) \end{array}$$

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

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I_1(\#\, \vec{X}\, [A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{t}) \in T_{A,\Delta_C} :\ (\Delta_C, f(\vec{t})) \in I_1(\varphi[\vec{X} / \vec{t}]) \\ & \text{and } \Delta = Log(A[\vec{X} / \vec{all}]) \ \oplus \ \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C}} f(\vec{t}) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & \forall (\vec{t}) \in T_{A,\Delta_C} :\ (\Delta_C, f(\vec{t})) \in I_2(\varphi[\vec{X} / \vec{t}]) \\ & \text{and } \Delta = Log(A[\vec{X} / \vec{all}]) \ \oplus \ \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C}} f(\vec{t}) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_2(\#\, \vec{X}\, [A \mapsto \varphi]) \end{array}$$

□

Lemma 4.79 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: Due to the construction of the interpretation I by intersection, $I \leq I'$ holds for all interpretations $I' \in \mathcal{J}$. Now let U be an arbitrary ground update goal. By Lemma 4.78,

$$I(U) \subseteq I'(U)$$

holds for all $I' \in \mathcal{J}$, and the desired inclusion follows directly. □

Lemma 4.80 [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: By Lemma 4.72, an interpretation I is a model of P_{UP}, iff for every ground instance $U \rightarrow p(\vec{t})$ of a rule $r \in P_{UP}$ the set inclusion $I(U) \subseteq I(p(\vec{t}))$ holds.

Now let $U \rightarrow p(\vec{t})$ 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 4.79 we can conclude:

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

□

Theorem 4.81 [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 4.76. 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 4.80, 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 program in our calendar example. Recall Theorem 4.24, which allow us to use all results proved for the generic ULTRA language also for the database language.

Example 4.82 [Personal Calendar (Cont.)] Consider the update rules

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

of our calendar example (see Appendix B) and recall Example 4.26, in particular the definition:

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

Because

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

holds for arbitrary interpretations $I \in \mathcal{I}$ (see Example 4.26), it follows that

$$(\emptyset, \{?entry, -entry(mon,10,0), +entry(mon,10,23)\}) \\ \in M(do\_allocate(mon,10,1,23))$$

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

If an assertion holds for some models, it does not need to hold for the minimal model. Let us have a look at the following negative example. Although there are many models M of P_{UP} for which

$$(\{-entry(mon,10,0)\}, \{?entry, -entry(mon,10,0), +entry(mon,10,23)\}) \\ \in M(do\_allocate(mon,10,1,23))$$

holds, it does not hold for the minimal model M_{UP}, because it is not justified by any rule body (see Example 4.26). A corresponding state change for the goal $do\_allocate(mon,10,1,23)$ is not specified by the programmer and thus excluded by the minimality condition. □

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

Definition 4.83 [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 $\mathcal{B}_{DU}$ by specifying its value w.r.t. arbitrary definable update atoms as follows: let $I \in \mathcal{I}$ be an arbitrary interpretation, then define

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

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 4.84 [Monotonicity] The immediate consequence operator $T_{P_{UP}}$ of Definition 4.83 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. By Lemma 4.78, $I_1(U) \subseteq I_2(U)$ holds for arbitrary ground update 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 4.83. □

Lemma 4.85 Let P_{UP} be an update program and $I \in \mathcal{I}$ be an arbitrary 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 Lemma 4.72 and Definition 4.83. □

Definition 4.86 The *ordinal powers* $T_{P_{UP}} \uparrow \gamma$ of the immediate consequence operator of an update program P_{UP} are defined as usual (cf. [Llo87]):

$$\begin{array}{lcll} T_{P_{UP}} \uparrow 0 & := & I_\perp & \\ T_{P_{UP}} \uparrow \alpha + 1 & := & T_{P_{UP}}(T_{P_{UP}} \uparrow \alpha) & \text{for a successor ordinal } \alpha + 1 \\ T_{P_{UP}} \uparrow \beta & := & lub\ \{\ T_{P_{UP}} \uparrow \gamma \mid \gamma < \beta\ \} & \text{for a limit ordinal } \beta \end{array}$$

□

Theorem 4.87 [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: Recall from Lemma 4.75 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 4.85. 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 $T_{P_{UP}}$ as proved in Lemma 4.84 is essential. □

Using our robot example, we want to sketch how the minimal model of an update program can be computed. The results proved for the generic ULTRA language are applicable to the programming language for external operations due to Theorem 4.61.

Example 4.88 [Robot World (Cont.)] Consider the update rules

$$
\begin{array}{lcl}
xmove(X) & \leftarrow & xpos(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X < X0 \ : \ xstep(-1) \ : \ xmove(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X > X0 \ : \ xstep(1) \ : \ xmove(X) \\
ymove(Y) & \leftarrow & ypos(Y) \\
ymove(Y) & \leftarrow & ypos(Y0) \ : \ Y < Y0 \ : \ ystep(-1) \ : \ ymove(Y) \\
ymove(Y) & \leftarrow & ypos(Y0) \ : \ Y > Y0 \ : \ ystep(1) \ : \ ymove(Y)
\end{array}
$$

of our robot example (see Example 3.20 and Appendix C). Further recall how the interpretation of update formulas has been derived in Example 4.62. Again, we assume that the robot is at position $(1,1)$ in the initial state s_0. Using the immediate consequence operator, one can easily derive the following assertions for the minimal model M_{UP}:

$$
\begin{array}{ll}
(\emptyset, \{?xpos\}) & \in \ M_{UP}(xmove(1)) \\
([?xpos, xstep(-1)], \{?xpos\}) & \in \ M_{UP}(xmove(0)) \\
(\emptyset, [?xpos, xstep(-1), ?xpos]) & \in \ M_{UP}(xmove(0)) \\
([?xpos, xstep(1)], \{?xpos\}) & \in \ M_{UP}(xmove(2)) \\
(\emptyset, [?xpos, xstep(1), ?xpos]) & \in \ M_{UP}(xmove(2)) \\
\text{etc.} &
\end{array}
$$

Analogous assertions can be derived for $ymove$.

Next, consider the rules

$$
\begin{array}{lcl}
move(X,Y) & \leftarrow & xmove(X), \ ymove(Y) \\
pickup\_at\_position(X,Y) & \leftarrow & [\ empty, \ move(X,Y)\ ] \ : \ pickup \ : \ NOT \ empty
\end{array}
$$

which specify a composite movement and a complex pickup operation, respectively. Further, have a look at the consistent pomsets shown in Figure 9. Provided that in state s_0 the robot is empty and there is a block lying on the floor at position $(2,0)$, it is straight-forward to show the following assertions:

$$
\begin{array}{ll}
(\emptyset, \Delta_1) & \in \ M_{UP}(move(2,0)) \\
(\emptyset, \Delta_2) & \in \ M_{UP}(pickup\_at\_position(2,0))
\end{array}
$$

□

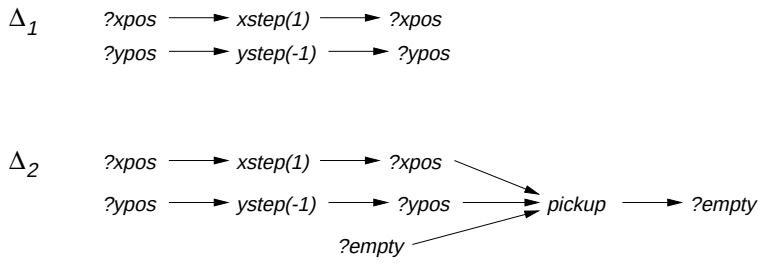

Figure 9: Pomsets for $move(2,0)$ and $pickup\_at\_position(2,0)$

In Theorem 4.87 we have shown that the minimal model of an update program can be characterized as a least fixpoint. Unfortunately, the iterated application of the immediate consequence operator at most to the first limit ordinal ω may be insufficient to compute the minimal model, i.e. the closure ordinal may be strictly greater than ω. This will be illustrated by the following example taken from the database domain (see Sections 3.2 and 4.3).

Example 4.89 Let us consider a logic database featuring a unary EDB predicate r as well as a unary IDB predicate q, which is defined by the following IDB rules:

$$\begin{array}{lcl} q(0) & & \\ q(s(X)) & \leftarrow & q(X) \end{array}$$

Independently of a chosen database state, I_{DB} will assign the truth value "true" to infinitely many ground instances of the atom $q(X)$. These instances result from replacing X by the terms s^i, where $s^0 :\equiv 0$ and $s^{i+1} :\equiv s(s^i)$ for $i \in \mathbb{N}$.

Next, let us define the following update program P_{UP}:

$$\begin{array}{lcl} p(0) & \leftarrow & INS\ r(0) \\ p(s(X)) & \leftarrow & INS\ r(s(X)),\ p(X) \\ t & \leftarrow & \#\ X\ [\ q(X)\ \mapsto\ p(X)\ ] \end{array}$$

P_{UP} has a unique minimal model M_{UP}, and the assertions

$$(\emptyset, \{+r(s^0), \ldots, +r(s^i)\}) \in M_{UP}(p(s^i))$$

can be inductively derived for all $i \in \mathbb{N}$. Now have a look at the bulk quantification, whose semantics requires the accumulation of infinitely many transitions. The choice function f (see case (Bulk) of Definition 4.9) can be defined to map each term s^i ($i \in \mathbb{N}$) to the update request set $\{+r(s^0), \ldots, +r(s^i)\}$. Consequently,

$$(\emptyset, \{?q\} \cup \{+r(s^i) \mid i \in \mathbb{N}\}) \in M_{UP}(\#X[q(X) \mapsto p(X)])$$

and thus

$$(\emptyset, \{?q\} \cup \{+r(s^i) \mid i \in \mathbb{N}\}) \in M_{UP}(t)$$

holds.

It should be observed that the minimal model cannot be derived by an iterated application of the immediate consequence operator $T_{P_{UP}}$ to the first limit ordinal ω. The interpretation of the bulk quantification remains empty until ω is reached and the interpretations of all relevant instances $p(s^i)$ of the update subgoal $p(X)$ have been computed. The computation of the correct interpretation of the atom t requires a subsequent application of $T_{P_{UP}}$. In other words, the closure ordinal $\alpha_{P_{UP}}$ is equal to $\omega + 1$.

Note that the closure ordinal $\omega + 1$ does not arise from a need to compute the truth values w.r.t. the IDB predicate q. Although this might be a significant problem for an operational semantics, the model-theoretic semantics abstracts from the computation of the state observations, i.e. I_{DB}

can be considered as implicitly materialized. The immediate consequence operator $T_{P_{UP}}$, which is an operator defined by the update program P_{UP} only, causes problems, because the set of single updates relevant for the bulk quantification is infinite and the single updates depend recursively on each other. □

A fixpoint semantics can be called impure at the computational level, when finite iterations of the immediate consequence operator or infinite iterations to the first limit ordinal are not sufficient to compute the fixpoint. Thus, we aim at finding conditions that render the immediate consequence operator $T_{P_{UP}}$ of an update program P_{UP} continuous. As shown by Kleene (see [Llo87] for details), the closure ordinal of a continuous operator is finite or equal to ω. The conditions we present in this thesis concern the bulk quantifications $\#\,\vec{X}\,[A \mapsto \varphi]$ that occur in a program: if either the set T_{A,Δ_C} defined in case (Bulk) of Definition 4.9 is always finite, or φ is a basic update atom (compare this to updates in the SQL language [DD97]), it is possible to show the continuity of the immediate consequence operator. Before we can present the main theorem, we need two lemmata.

Lemma 4.90 Let $\mathcal{J} \subseteq \mathcal{I}$ be a directed set of interpretations. Let $U_1, \ldots, U_n$ be ground update goals and $I_1, \ldots, I_n \in \mathcal{J}$ be corresponding interpretations. Then there exists an interpretation $I \in \mathcal{J}$ such that

$$I_i(U_i) \subseteq I(U_i)$$

holds for every $i \in \{1, \ldots, n\}$.

Proof: Define $I := lub\ \{I_1, \ldots, I_n\}$. Since $\mathcal{J}$ is directed, $I \in \mathcal{J}$ will hold.

Next, choose an arbitrary index $i \in \{1, \ldots, n\}$. Because I is defined as an upper bound, $I_i \leq I$ holds, and by Lemma 4.78, $I_i(U_i) \subseteq I(U_i)$ follows. □

Lemma 4.91 Let $\mathcal{J} \subseteq \mathcal{I}$ be a directed set of interpretations.

Let U be an arbitrary ground update goal, such that one of the following conditions holds for every bulk quantification $\#\,\vec{X}\,[A \mapsto \varphi]$ contained in U:

1. The set T_{A,Δ_C} as defined in case (Bulk) of Definition 4.9 is finite for every consistent transition $\Delta_C \in \mathcal{T}_{Cons}$.
2. The subgoal φ is a basic update atom, i.e. an atom $u(\vec{t})$ with $u \in Pred_{BU}$.

Then

$$(\Delta_C, \Delta) \in lub(\mathcal{J})(U) \implies \text{there exists an interpretation } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(U)$$

holds for arbitrary consistent transitions $\Delta_C, \Delta \in \mathcal{T}_{Cons}$.

Proof:

We prove the assertion by structural induction. In each case shown below, we choose arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ and show the desired implication.

Note that $\mathcal{J}$ is not empty, since $lub(\emptyset) \in \mathcal{J}$ holds. We will refer to $I_0 := lub(\emptyset)$ in some base cases, where we must show the existence of any interpretation in $\mathcal{J}$.

Base cases:

Essentially, we apply Definition 4.9.

1. DB literal

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

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(q(\vec{t})) \\ \Longrightarrow & I_{DB}(s_0 \oplus_E \Delta_C) \models q(\vec{t}) \text{ and } \Delta = Log(q(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_0(q(\vec{t})) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(q(\vec{t})) \end{array}$$

2. NOP literal

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(NOP) \\ \Longrightarrow & \Delta = \Delta_\varepsilon \\ \Longrightarrow & (\Delta_C, \Delta) \in I_0(NOP) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(NOP) \end{array}$$

3. Basic update atom

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(u(\vec{t})) \\ \Longrightarrow & \Delta = Upd(u(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_0(u(\vec{t})) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(u(\vec{t})) \end{array}$$

4. Definable update atom

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(p(\vec{t})) \\ \underset{(4.75)}{\Longrightarrow} & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(p(\vec{t})) \end{array}$$

   The assertion follows directly from the property

   $$lub(\mathcal{J})(p(\vec{t})) = \bigcup_{I \in \mathcal{J}} I(p(\vec{t}))$$

   presented in Lemma 4.75.

Induction step:

By the induction hypothesis, the assertion holds for any direct proper subgoal φ of the composite goals analyzed in the following. Note that Lemma 4.90 and the preconditions w.r.t. bulk quantifications will be essential for the induction step. Using Definition 4.9 we can reason as follows.

1. Concurrent conjunction

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in lub(\mathcal{J})(\varphi, \psi) \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in lub(\mathcal{J})(\varphi) \text{ and } (\Delta_C, \Delta_2) \in lub(\mathcal{J})(\psi) \\
 & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ and } I_1, I_2 \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I_1(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I_2(\psi) \\
 & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ and } I \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (4.90)} & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\varphi, \psi)
\end{array}
$$

2. Sequential conjunction

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in lub(\mathcal{J})(\varphi : \psi) \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in lub(\mathcal{J})(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in lub(\mathcal{J})(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ and } I_1, I_2 \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I_1(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I_2(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ and } I \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (4.90)} & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\varphi : \psi)
\end{array}
$$

3. Disjunction

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in lub(\mathcal{J})(\varphi \vee \psi) \\
\Longrightarrow & (\Delta_C, \Delta) \in lub(\mathcal{J})(\varphi) \text{ or } (\Delta_C, \Delta) \in lub(\mathcal{J})(\psi) \\
\Longrightarrow & \text{there exist } I_1, I_2 \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (IH)} & (\Delta_C, \Delta) \in I_1(\varphi) \text{ or } (\Delta_C, \Delta) \in I_2(\psi) \\
\Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (4.90)} & (\Delta_C, \Delta) \in I(\varphi) \text{ or } (\Delta_C, \Delta) \in I(\psi) \\
\Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\varphi \vee \psi)
\end{array}
$$

4. Existential quantification

   Consider a quantification $\exists\, \vec{X}\, \varphi$, where $\vec{X} = X_1, \ldots, X_n$. By the induction hypothesis, the assertion holds for all instances $\varphi[\, \vec{X} \,/\, \vec{t}\,]$ of the update subgoal φ.

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in lub(\mathcal{J})(\exists\, \vec{X}\, \varphi) \\
\Longrightarrow & \text{there exists a ground term tuple } (\,\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
 & (\Delta_C, \Delta) \in lub(\mathcal{J})(\varphi[\, \vec{X} \,/\, \vec{t}\,]) \\
\Longrightarrow & \text{there exists } (\,\vec{t}\,) \in \mathcal{U}^n \text{ and } I \in \mathcal{J} \text{ such that :} \\
{\scriptstyle (IH)} & (\Delta_C, \Delta) \in I(\varphi[\, \vec{X} \,/\, \vec{t}\,]) \\
\Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\exists\, \vec{X}\, \varphi)
\end{array}
$$

5. Bulk quantification

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

$$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longrightarrow & \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_0(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\#\,\vec{X}\,[A \mapsto \varphi]) \end{array}$$

Next, let us consider the case $T_{A,\Delta_C} \neq \emptyset$. To show the assertion, we have to use at least one of the preconditions required for the bulk quantification.

If T_{A,Δ_C} is finite (according to condition 1), we get:

$$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T_{A,\Delta_C} : \ (\Delta_C, f(\,\vec{t}\,)) \in lub(\mathcal{J})(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus \ \bigsqcup_{(\,\vec{t}\,) \in T_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longrightarrow & \text{there exist interpretations } I_{(\,\vec{t}\,)} \in \mathcal{J} \text{ for the tuples } (\,\vec{t}\,) \in T_{A,\Delta_C} \\ (IH) & \text{and a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T_{A,\Delta_C} : \ (\Delta_C, f(\,\vec{t}\,)) \in I_{(\,\vec{t}\,)}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus \ \bigsqcup_{(\,\vec{t}\,) \in T_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longrightarrow & \text{there exist an interpretation } I \in \mathcal{J} \\ (4.90) & \text{and a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T_{A,\Delta_C} : \ (\Delta_C, f(\,\vec{t}\,)) \in I(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus \ \bigsqcup_{(\,\vec{t}\,) \in T_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\#\,\vec{X}\,[A \mapsto \varphi]) \end{array}$$

The finiteness of T_{A,Δ_C} is essential, because Lemma 4.90 can deal only with finite sequences of goals and corresponding interpretations.

If φ is a basic update atom (according to condition 2), its interpretation does not depend on a particular interpretation of definable update atoms (cf. case (BU) of Definition 4.9). Thus, we can use $I_0 \in \mathcal{J}$ instead of $lub(\mathcal{J})$ and reason as follows:

$$\begin{array}{ll} & (\Delta_C, \Delta) \in lub(\mathcal{J})(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T_{A,\Delta_C} : \ (\Delta_C, f(\,\vec{t}\,)) \in lub(\mathcal{J})(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus \ \bigsqcup_{(\,\vec{t}\,) \in T_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longrightarrow & \text{there exists a function } f : T_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ (see\ above) & \forall(\,\vec{t}\,) \in T_{A,\Delta_C} : \ (\Delta_C, f(\,\vec{t}\,)) \in I_0(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus \ \bigsqcup_{(\,\vec{t}\,) \in T_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longrightarrow & (\Delta_C, \Delta) \in I_0(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longrightarrow & \text{there exists } I \in \mathcal{J} \text{ such that : } (\Delta_C, \Delta) \in I(\#\,\vec{X}\,[A \mapsto \varphi]) \end{array}$$

□

Now we are able to show the continuity of the immediate consequence operator of an update program. Recall that we must require some conditions for the bulk quantifications that occur in the program.

Theorem 4.92 [Continuity] Let P_{UP} be an update program, where every bulk quantification $\#\,\vec{X}\,[A \mapsto \varphi]$ occurring in a rule body satisfies one of the following conditions:

1. The set T_{A,Δ_C} as defined in case (Bulk) of Definition 4.9 is finite for every consistent transition $\Delta_C \in \mathcal{T}_{Cons}$.
2. The subgoal φ is a basic update atom, i.e. an atom $u(\vec{t})$ with $u \in Pred_{BU}$.

Then the immediate consequence operator $T_{P_{UP}}$ is continuous.

Proof: According to [Llo87], we have to show that

$$T_{P_{UP}}(lub(\mathcal{J})) = lub(T_{P_{UP}}(\mathcal{J}))$$

holds for every directed set $\mathcal{J} \subseteq \mathcal{I}$ of interpretations.

Now let $\mathcal{J} \subseteq \mathcal{I}$ be directed. To prove the desired equality, let us choose an arbitrary ground definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$ and two arbitrary consistent transitions $\Delta_C, \Delta \in \mathcal{T}_{Cons}$. We can show the following equivalences:

$$\begin{array}{ll}
 & (\Delta_C, \Delta) \in T_{P_{UP}}(lub(\mathcal{J}))(p(\vec{t})) \\
\iff & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
\scriptstyle{(4.83)} & \text{such that } (\Delta_C, \Delta) \in lub(\mathcal{J})(U) \\
\iff & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP} \\
\scriptstyle{(see\ below)} & \text{and an interpretation } I \in \mathcal{J}, \text{ such that } (\Delta_C, \Delta) \in I(U) \\
\iff & \text{there exists an interpretation } I \in \mathcal{J}, \text{ such that } (\Delta_C, \Delta) \in T_{P_{UP}}(I)(p(\vec{t})) \\
\scriptstyle{(4.83)} & \\
\iff & (\Delta_C, \Delta) \in \bigcup_{I \in \mathcal{J}} T_{P_{UP}}(I)(p(\vec{t})) \\
\iff & (\Delta_C, \Delta) \in lub(T_{P_{UP}}(\mathcal{J}))(p(\vec{t})) \\
\scriptstyle{(4.75)} &
\end{array}$$

The second equivalence is the most important one and must be explained in more detail: For the direction '$\Rightarrow$' we can apply Lemma 4.91. Recall that the preconditions required for the bulk quantifications are essential. The converse '$\Leftarrow$' follows directly from Lemma 4.78, because $I \leq lub(\mathcal{J})$ holds for each interpretation $I \in \mathcal{J}$. □

Corollary 4.93 [Continuity] Let an instance of the ULTRA framework be given. Each of the following conditions is sufficient to guarantee that every update program has a continuous immediate consequence operator.

1. The Herbrand universe $\mathcal{U}$ is finite. (This implies also the absence of function symbols.)
2. For every state $s \in \mathcal{S}$, $I_{DB}(s)$ assigns the truth value "true" to finitely many ground DB atoms $A \in \mathcal{B}$.

3. The use of the bulk quantifier is generally forbidden.

Proof: Each condition above implies that all bulk quantifications in a program satisfy condition 1 of Theorem 4.92. □

Let us summarize the results about the semantics of update programs. For every program, we have defined a unique minimal model that can be characterized as the least fixpoint of a straightforward immediate consequence operator. Finally, we have presented some conditions (w.r.t. the bulk quantification) that guarantee the continuity of the immediate consequence operator, such that the minimal model can be computed by iteration at most to the first limit ordinal ω. Consequently, the fixpoint semantics may serve as the starting point for an operational semantics of the ULTRA language.

Definition 4.94 Let P_{UP} be an update program, and let $s_0 \in \mathcal{S}$ be a state. Then

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

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

Note that $M_{UP}[P_{UP}, s_0] \in \mathcal{I}$ is an interpretation of definable update atoms. The interpretation $M_{UP}[P_{UP}, s_0](\varphi)$ of a ground update formula φ should be implicitly defined w.r.t. s_0, too. □

5 Transactions and Serializability

In this section we will consider operations defined by an update program in the ULTRA language as transactions. After defining their effects according to the model-theoretic semantics developed in Section 4, we show how to support the isolation of independent transactions. The solutions are developed with an operational model based on deferred materialization and hypothetical reasoning in mind, as this strategy integrates smoothly with the logical semantics. We present sufficient preconditions for a transaction manager and derive an optimistic transaction processing method (cf. Section 2.5). This method allows the concurrent execution of multiple transactions while maintaining the ACID properties [BHG87, BN97, GR93].

The fundamental results presented in this section hold for the generic ULTRA concept. We demonstrate how to apply them directly to the database-oriented instance developed in Sections 3.2 and 4.3 as well as to the instance for external operations developed in Sections 3.3 and 4.4. It should be noted that the results are not restricted to top-level transactions. Thus, they may be further exploited for operational semantics built on the nested transaction model [BBG89, Mos85, WS92]. In particular, hybrid approaches that interleave multiple evaluation phases and multiple materialization phases within one top-level transaction can benefit from our results.

5.1 Transactions in ULTRA

A new top-level transaction is invoked by submitting a top-level update query to the ULTRA system. The interpretation of the query goal yields possible transitions, which can transform the initial state s_0 to a desired future state.

Definition 5.1 [Top-Level Update Queries] A *top-level update query* $\leftarrow U$ is an update rule without a head, where the body U does not contain any free variables. □

Definition 5.2 [Possible Transitions] Let P_{UP} be an update program and let $s_0 \in \mathcal{S}$ be a state. Let Q be a top-level update query of the form $\leftarrow U$.

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 w.r.t. the initial state s_0. □

To commit a transaction invoked by a top-level query, one possible transition of the query 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. In this case, it must not cause a (persistent) state change. A transaction will logically fail, if there exists no possible transition.

Example 5.3 [Personal Calendar (Cont.)] Our calendar tool features an update operation $do\_insert\_on\_day(D, L, T)$ 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

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

is issued in database state DB_0 of Example 3.17. 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 5.4 [Robot World (Cont.)] Recall Example 4.88, in particular the chosen initial state s_0, and assume that the update query $\leftarrow pickup\_at\_position(2, 0)$ is submitted. We have already shown that for the transition Δ_2 depicted in Figure 9 on page 77,

$$(\emptyset, \Delta_2) \in M_{UP}[P_{UP}, s_0](pickup\_at\_position(2, 0))$$

holds. Thus Δ_2 is a possible transition for the update query above.

If the robot world is in a state where the robot arm is not empty or there is no block lying at position $(2, 0)$, 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 non-deterministic update semantics defined in this thesis, we do not allow free variables in top-level update queries. Intuitively speaking, the complex operations can only be invoked with all parameters instantiated. 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 easily be integrated into the overall approach provided that they produce logging transitions according to the queried information. It should be emphasized that retrieval queries have no relation to the minimal model of an update program. They are added for convenience, as they fit into the transaction model, which we will present subsequently.

Definition 5.5 [Retrieval Queries] A *retrieval query* Q is of the form $\leftarrow q(\vec{t})$, where $q(\vec{t})$ is a DB atom containing the variables $\vec{X}$.

We assign to Q the “possible transition” $\Delta := Log(q(\vec{t})[\vec{X} / \vec{all}])$. Note that is also the possible transition of the bulk quantification $\#\vec{X}\,[q(\vec{t}) \mapsto NOP]$. □

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 below for details). These checks can be seen as part of the materialization.

Example 5.6 [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. □

5.2 Read-Isolation

Now we define the concept of read-isolation of one transition from another. Under certain conditions the read-isolation property can be used to guarantee isolation and serializability of independent transactions. This will be shown formally in Section 5.3.

Definition 5.7 [Read-Isolation] Let a binary relation R on the set $\mathcal{T}$ of transitions be given, where $(\Delta_1, \Delta_2) \in R$ expresses that a transition $\Delta_1 \in \mathcal{T}$ is *read-isolated* from another transition $\Delta_2 \in \mathcal{T}$. R does not need to be symmetric.

R is called a *read-isolation relation*, if the following properties hold:

1. Let $\Delta_0 \in \mathcal{T}_{Cons}$ be a consistent transition, and let $\Delta_1, \Delta_2 \in \mathcal{T}$ be arbitrary transitions. Then the following holds:

   (a) $\Delta_1 \sqcup \Delta_2$ read-isolated from Δ_0
   $\Longrightarrow$ Δ_1 read-isolated from Δ_0 and Δ_2 read-isolated from Δ_0

   (b) $\Delta_1 \oplus \Delta_2$ read-isolated from Δ_0
   $\Longrightarrow$ Δ_1 read-isolated from Δ_0 and Δ_2 read-isolated from Δ_0

2. Let $\Delta_0 \in \mathcal{T}_{Cons}$ be a consistent transition and A be a ground DB atom or a non-ground DB atom containing the variables $\vec{X}$. Let $A' := A[\,\vec{X} \,/\, \vec{all}\,]$ be the ground instance of A, where all variables have been replaced by the special constant *all*, and let $A'' := A[\,\vec{X} \,/\, \vec{t}\,]$ be an arbitrary ground instance of A. Further, let $Log(A')$ be read-isolated from Δ_0. Then for all states $s \in \mathcal{S}$ and all consistent transitions $\Delta \in \mathcal{T}_{Cons}$ the following holds:

   (a) $I_{DB}((s \oplus_E \Delta_0) \oplus_E \Delta) \models A'' \quad \Longleftrightarrow \quad I_{DB}(s \oplus_E \Delta) \models A''$
   (b) $I_{DB}((s \oplus_E \Delta_0) \oplus_E \Delta) \models \neg A'' \quad \Longleftrightarrow \quad I_{DB}(s \oplus_E \Delta) \models \neg A''$

□

The algebraic properties required in Definition 5.7 state that the read-isolation property carries over to subparts of a decomposable transition and that the read-isolation of a logging transition disallows observable changes in the truth value of the logged DB atom. Property 2 also defines a particular property for the special constant *all*: the semantical conditions should be maintained, when the constant *all* is replaced by an arbitrary term of $\mathcal{U}$.

Example 5.8 In the ULTRA database instance of Sections 3.2 and 4.3, one update request set Δ will be called read-isolated from another update request set Δ_0, if Δ does not contain any read tags on EDB relations, for which Δ_0 specifies insertions or deletions. This corresponds to the absence of read/write conflicts in classical database transactions [BHG87]. □

5.3 Isolation of Transactions

After having defined some preliminaries, we can investigate the isolation problem for concurrent transactions in the ULTRA framework. The formal results will be derived w.r.t. an arbitrary but fixed initial state $s_0 \in \mathcal{S}$, and we assume that two or more independent transactions are invoked in this state. Further, we require that the transaction processing is divided into an *evaluation phase*, where the transactions simultaneously operate on the fixed state s_0 without making changes visible to each other, and a subsequent *materialization phase*, where each transaction may execute one possible transition. We are going to show that checking the read-isolation property of the possible transitions is a viable method to determine the isolation of the underlying transactions. In the following we often refer to top-level transactions invoked by an update query. The results, however, are more general and can be exploited for subtransactions of a top-level transaction as well, provided that the subtransactions are executed in two phases. Note that execution in two phases does not always require a *logical* concept for hypothetical reasoning: for instance, changes could also be made in a private workspace, and external actions could be executed in a (private) simulation model instead of the real system.

Definition 5.9 Let $I \in \mathcal{I}$ be an interpretation of update formulas and $\Delta_0 \in \mathcal{T}_{Cons}$ be an arbitrary but fixed consistent transition. Let φ be a ground update formula. We say that I, Δ_0, and φ have the property (*), iff the following condition holds:

$$\text{For arbitrary } \Delta_C, \Delta'_C, \Delta \in \mathcal{T}_{Cons} \text{ such that } \Delta \text{ is read-isolated from } \Delta_0, \\ (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi) \Longleftrightarrow (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi) \text{ holds.} \qquad (*)$$

□

Our objective is to show that the minimal model $M_{UP}[P_{UP}, s_0]$ of a given update program P_{UP} w.r.t. the physical state s_0 satisfies (*) for arbitrary update request sets Δ_0 and arbitrary ground update formulas φ. From this assertion we can derive serializability properties and even an optimistic transaction processing method that guarantees full isolation of concurrent transactions.

Lemma 5.10 Let $I \in \mathcal{I}$ be an interpretation of update formulas and $\Delta_0 \in \mathcal{T}_{Cons}$ be an arbitrary but fixed consistent transition.

If (*) holds for every ground definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$, then (*) holds for every ground update formula φ.

Proof: We prove the assertion by structural induction. Let (*) hold for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$. In each case shown below, we choose arbitrary $\Delta_C, \Delta'_C, \Delta \in \mathcal{T}_{Cons}$ such that Δ is read-isolated from Δ_0 and show the equivalence in Definition 5.9 using Definition 4.9.

Base cases:

1. DB literal

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

   $$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(A) \\ \iff & I_{DB}(s_0 \oplus_E (\Delta_C \oplus \Delta_0 \oplus \Delta'_C)) \models A \text{ and } \Delta = Log(A) \\ \iff & I_{DB}(((s_0 \oplus_E \Delta_C) \oplus_E \Delta_0) \oplus_E \Delta'_C) \models A \text{ and } \Delta = Log(A) \\ \underset{(4.2)}{\iff} & I_{DB}((s_0 \oplus_E \Delta_C) \oplus_E \Delta'_C) \models A \text{ and } \Delta = Log(A) \\ \underset{(5.7)}{\iff} & I_{DB}(s_0 \oplus_E (\Delta_C \oplus \Delta'_C)) \models A \text{ and } \Delta = Log(A) \\ \underset{(4.2)}{\iff} & (\Delta_C \oplus \Delta'_C, \Delta) \in I(A) \end{array}$$

   Note that property 2 required for the read-isolation relation (see Definition 5.7) is essential for this part of the proof.

2. NOP literal

   $$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(NOP) \\ \iff & \Delta = \Delta_\varepsilon \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(NOP) \end{array}$$

3. Basic update atom

   $$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(u(\vec{t})) \\ \iff & \Delta = Upd(u(\vec{t})) \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(u(\vec{t})) \end{array}$$

4. Definable update atom

   The equivalence holds by the precondition.

Induction step:

By the induction hypothesis, (*) holds for any direct proper subformula φ of the composite formulas analyzed in the following.

1. Concurrent conjunction

   $$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi, \psi) \\ \iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta_2) \in I(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ \underset{(IH)}{} & (\Delta_C \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta'_C, \Delta_2) \in I(\psi) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi, \psi) \end{array}$$

   Both Δ_1 and Δ_2 are read-isolated from Δ_0 by property 1 of Definition 5.7. Therefore, the induction hypothesis is applicable.

2. Sequential conjunction

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi : \psi) \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } ((\Delta_C \oplus \Delta_0 \oplus \Delta'_C) \oplus \Delta_1, \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
_{(4.2)} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_0 \oplus (\Delta'_C \oplus \Delta_1), \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
_{(IH)} & (\Delta_C \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus (\Delta'_C \oplus \Delta_1), \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
_{(4.2)} & (\Delta_C \oplus \Delta'_C, \Delta_1) \in I(\varphi) \text{ and } ((\Delta_C \oplus \Delta'_C) \oplus \Delta_1, \Delta_2) \in I(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi : \psi)
\end{array}
$$

   Both Δ_1 and Δ_2 are read-isolated from Δ_0 by property 1 of Definition 5.7. Therefore, the induction hypothesis is applicable.

3. Disjunction

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi \vee \psi) \\
\iff & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi) \text{ or } (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\psi) \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi) \text{ or } (\Delta_C \oplus \Delta'_C, \Delta) \in I(\psi) \\
_{(IH)} & \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi \vee \psi)
\end{array}
$$

4. Existential quantification

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

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\exists \vec{X}\, \varphi) \\
\iff & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi[\vec{X} / \vec{t}\,]) \\
\iff & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
_{(IH)} & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi[\vec{X} / \vec{t}\,]) \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\exists \vec{X}\, \varphi)
\end{array}
$$

5. Bulk quantification

   Consider a bulk quantification $\#\vec{X}\,[A \mapsto \varphi]$, where $\vec{X} = X_1, \ldots, X_n$. Assume that $(\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\#\vec{X}\,[A \mapsto \varphi])$ or $(\Delta_C \oplus \Delta'_C, \Delta) \in I(\#\vec{X}\,[A \mapsto \varphi])$ holds. Regarding case (Bulk) of Definition 4.9, it is obvious that there exists a transition $\Delta' \in \mathcal{T}$ such that $\Delta = Log(A[\vec{X} / \vec{all}\,]) \oplus \Delta'$. ($\Delta'$ expresses the resulting bulk update, which is composed with the logging transition of the DB atom A.) Then it is possible to apply the properties of the read-isolation relation (see Definition 5.7): $Log(A[\vec{X} / \vec{all}\,])$ must be read-isolated from Δ_0 by property 1 (b), and we can reason as follows, where property

2 (a) is essential:

$$\begin{array}{ll} & T_{A,\Delta_C \oplus \Delta_0 \oplus \Delta'_C} \\ = & \{(\vec{t}) \in \mathcal{U}^n \mid I_{DB}(s_0 \oplus_E (\Delta_C \oplus \Delta_0 \oplus \Delta'_C)) \models A[\vec{X} / \vec{t}]\} \\ = & \{(\vec{t}) \in \mathcal{U}^n \mid I_{DB}(((s_0 \oplus_E \Delta_C) \oplus_E \Delta_0) \oplus_E \Delta'_C) \models A[\vec{X} / \vec{t}]\} \\ \underset{(4.2)}{=} & \\ = & \{(\vec{t}) \in \mathcal{U}^n \mid I_{DB}((s_0 \oplus_E \Delta_C) \oplus_E \Delta'_C) \models A[\vec{X} / \vec{t}]\} \\ \underset{(5.7)}{=} & \\ = & \{(\vec{t}) \in \mathcal{U}^n \mid I_{DB}(s_0 \oplus_E (\Delta_C \oplus \Delta'_C)) \models A[\vec{X} / \vec{t}]\} \\ \underset{(4.2)}{=} & \\ = & T_{A,\Delta_C \oplus \Delta'_C} \end{array}$$

The case $T_{A,\Delta_C \oplus \Delta'_C} = \emptyset$ is trivial:

$$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \\ \iff & \Delta = Log(A[\vec{X} / a\vec{l}l]) \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \end{array}$$

Let us now consider the case $T_{A,\Delta_C \oplus \Delta'_C} \neq \emptyset$:

$$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \\ \iff & \text{there exists a function } f : T_{A,\Delta_C \oplus \Delta_0 \oplus \Delta'_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{t}) \in T_{A,\Delta_C \oplus \Delta_0 \oplus \Delta'_C} : \ (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, f(\vec{t})) \in I(\varphi[\vec{X} / \vec{t}]) \\ & \text{and } \Delta = Log(A[\vec{X} / a\vec{l}l]) \ \oplus \ \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C \oplus \Delta_0 \oplus \Delta'_C}} f(\vec{t}) \\ \iff & \text{there exists a function } f : T_{A,\Delta_C \oplus \Delta'_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{t}) \in T_{A,\Delta_C \oplus \Delta'_C} : \ (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, f(\vec{t})) \in I(\varphi[\vec{X} / \vec{t}]) \\ & \text{and } \Delta = Log(A[\vec{X} / a\vec{l}l]) \ \oplus \ \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C \oplus \Delta'_C}} f(\vec{t}) \\ \iff & \text{there exists a function } f : T_{A,\Delta_C \oplus \Delta'_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ \underset{(IH)}{} & \forall (\vec{t}) \in T_{A,\Delta_C \oplus \Delta'_C} : \ (\Delta_C \oplus \Delta'_C, f(\vec{t})) \in I(\varphi[\vec{X} / \vec{t}]) \\ & \text{and } \Delta = Log(A[\vec{X} / a\vec{l}l]) \ \oplus \ \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C \oplus \Delta'_C}} f(\vec{t}) \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\# \vec{X}\, [A \mapsto \varphi]) \end{array}$$

The induction hypothesis is applicable, because for all ground tuples $(\vec{t}) \in T_{A,\Delta_C \oplus \Delta'_C}$, $f(\vec{t})$ must be read-isolated from Δ_0. This can be shown using the property 5 (a) required for $\bigsqcup$ in Definition 4.2 and property 1 of the read-isolation relation (see Definition 5.7).

6. Implication

$$\begin{array}{ll} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi \to \psi) \\ \iff & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi) \Longrightarrow (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\psi) \\ \iff & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \notin I(\varphi) \text{ or } (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\psi) \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \notin I(\varphi) \text{ or } (\Delta_C \oplus \Delta'_C, \Delta) \in I(\psi) \\ \underset{(IH)}{} & \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi) \Longrightarrow (\Delta_C \oplus \Delta'_C, \Delta) \in I(\psi) \\ \iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi \to \psi) \end{array}$$

7. Universal quantification

   Consider a quantification $\forall \vec{X}\, \varphi$, where $\vec{X} = X_1, \ldots, X_n$.

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\forall \vec{X}\, \varphi) \\
\iff & \text{for all ground term tuples } (\vec{t}) \in \mathcal{U}^n, \\
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in I(\varphi[\vec{X} \,/\, \vec{t}]) \text{ holds} \\
\iff & \text{for all ground term tuples } (\vec{t}) \in \mathcal{U}^n, \\
\scriptstyle (IH) & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\varphi[\vec{X} \,/\, \vec{t}]) \text{ holds} \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in I(\forall \vec{X}\, \varphi)
\end{array}
$$

□

Theorem 5.11 [Validity of Property (*)] The unique minimal model of an update program P_{UP} satisfies the property (*) for any arbitrary but fixed consistent transition $\Delta_0 \in \mathcal{T}_{Cons}$ and for every ground update formula φ.

Proof: It is sufficient to prove the assertion for definable update atoms. The assertion for arbitrary update formulas follows by Lemma 5.10.

After choosing an arbitrary transition $\Delta_0 \in \mathcal{T}_{Cons}$, we show by transfinite induction that (*) holds w.r.t. Δ_0 and $T_{P_{UP}} \uparrow \gamma$ for arbitrary ordinals γ. That means, we verify the following condition for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$:

> For arbitrary $\Delta_C, \Delta'_C, \Delta \in \mathcal{T}_{Cons}$ such that Δ is read-isolated from Δ_0,
> $(\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \gamma\,(p(\vec{t})) \iff (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \gamma\,(p(\vec{t}))$ holds.

Base case: $\gamma = 0$

> Let $p(\vec{t}) \in \mathcal{B}_{DU}$ be a definable update atom. Because $I_\perp(p(\vec{t})) = \emptyset$, the property (*) trivially holds w.r.t. $I_\perp$ and $p(\vec{t})$.

Induction step: successor ordinal $\alpha + 1$

> By the induction hypothesis, (*) holds w.r.t. $T_{P_{UP}} \uparrow \alpha$ for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$. By Lemma 5.10, (*) even holds w.r.t. $T_{P_{UP}} \uparrow \alpha$ for arbitrary ground update formulas φ.
>
> Now choose a definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$. For arbitrary $\Delta_C, \Delta'_C, \Delta \in \mathcal{T}_{Cons}$ such that Δ is read-isolated from Δ_0, the following equivalences hold.

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha + 1\,(p(\vec{t})) \\
\iff & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}}(T_{P_{UP}} \uparrow \alpha)(p(\vec{t})) \\
\scriptstyle (4.86) & \\
\iff & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
\scriptstyle (4.83) & \text{such that } (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(U) \\
\iff & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
\scriptstyle (IH) & \text{such that } (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(U) \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}}(T_{P_{UP}} \uparrow \alpha)(p(\vec{t})) \\
\scriptstyle (4.83) & \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha + 1\,(p(\vec{t})) \\
\scriptstyle (4.86) &
\end{array}
$$

Induction step: limit ordinal β

By the induction hypothesis, (*) holds w.r.t. each $T_{P_{UP}} \uparrow \alpha \ \ (\alpha < \beta)$ for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$.

Now choose a definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$. For arbitrary $\Delta_C, \Delta'_C, \Delta \in \mathcal{T}_{Cons}$ such that Δ is read-isolated from Δ_0, the following equivalences hold.

$$
\begin{array}{ll}
 & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \beta\,(p(\vec{t})) \\
\stackrel{(4.75)}{\iff} & (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\iff & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C \oplus \Delta_0 \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\stackrel{(IH)}{\iff} & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\iff & (\Delta_C \oplus \Delta'_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\stackrel{(4.75)}{\iff} & (\Delta_C \oplus \Delta'_C, \Delta) \in T_{P_{UP}} \uparrow \beta\,(p(\vec{t}))
\end{array}
$$

In particular, the assertion holds w.r.t. $T_{P_{UP}} \uparrow \alpha_{P_{UP}}$, which is equal to the minimal model of P_{UP} due to Theorem 4.87. □

Theorem 5.11 states that the model-theoretic interpretation of an update formula φ may contain corresponding pairs (Δ_C, Δ) of transitions that only differ at the first position Δ_C where the difference lies in a composition with Δ_0 or Δ_ε (as the neutral transition). Provided that Δ is read-isolated from Δ_0, the (hypothetical) execution of Δ_0 at any time is irrelevant for the interpretation of φ. This formal result will be applied to legitimate a simultaneous evaluation of concurrent transactions. Note that the assertion of Theorem 5.11 is not restricted to the top-level, since it does not explicitly refer to top-level update queries. It merely expresses a result about the model-theoretic semantics w.r.t. different hypothetical states and can thus also be exploited for an operational semantics based on the nested transaction model [BBG89, Mos85, WS92]. In combination with the properties shown in Section 6, Theorem 5.11 may be helpful to verify operational models that feature multiple evaluation and materialization phases within one top-level transaction.

We are now able to prove the following corollary expressing some form of isolation of top-level transactions. We consider two already computed transitions Δ_1 and Δ_2 which have to be materialized in order to complete the corresponding transactions. Although it is not necessary for the proof of Corollary 5.12, the property we desire becomes more obvious when looking at Remark 5.13, which says that Δ_1 is computed w.r.t. the same initial state (and update program) as Δ_2.

Corollary 5.12 [Isolation] Let P_{UP} be an update program and $s_0 \in \mathcal{S}$ be the initial state. Let U_2 be a ground update goal.

Let $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ be transitions such that:

1. Δ_2 is read-isolated from Δ_1.
2. $(\Delta_\varepsilon, \Delta_2) \in M_{UP}[P_{UP}, s_0](U_2)$

Then the following holds:

$$(\Delta_1, \Delta_2) \in M_{UP}[P_{UP}, s_0](U_2)$$

Proof: The assertion follows from Theorem 5.11 with $\Delta_0 := \Delta_1$ and $\varphi := U_2$. Apply (*) backwards with $\Delta_C := \Delta_\varepsilon$, $\Delta'_C := \Delta_\varepsilon$ and $\Delta := \Delta_2$. □

Remark 5.13 Let U_1 be another ground update goal such that

$$(\Delta_\varepsilon, \Delta_1) \in M_{UP}[P_{UP}, s_0](U_1).$$

Under this additional condition, Corollary 5.12 states that the computed possible transition Δ_2 for goal U_2 is still valid in a (hypothetical) state where the transition of the other goal U_1 has already been executed, although the possible transitions have been computed w.r.t. the same physical state s_0. Thus, if first Δ_1 and then Δ_2 is materialized, the observable effect is equal to a serial, isolated execution of the transactions invoked by the queries $\leftarrow U_1$ and $\leftarrow U_2$, respectively. This is illustrated in Figure 10. Note that serializability is not guaranteed, if Δ_2 is materialized before Δ_1. This would require that Δ_1 is also read-isolated from Δ_2. We do not have to care about execution conflicts between the transitions, as the materialization is assumed to be performed in a strictly serial way, i.e. Δ_2 after Δ_1. □

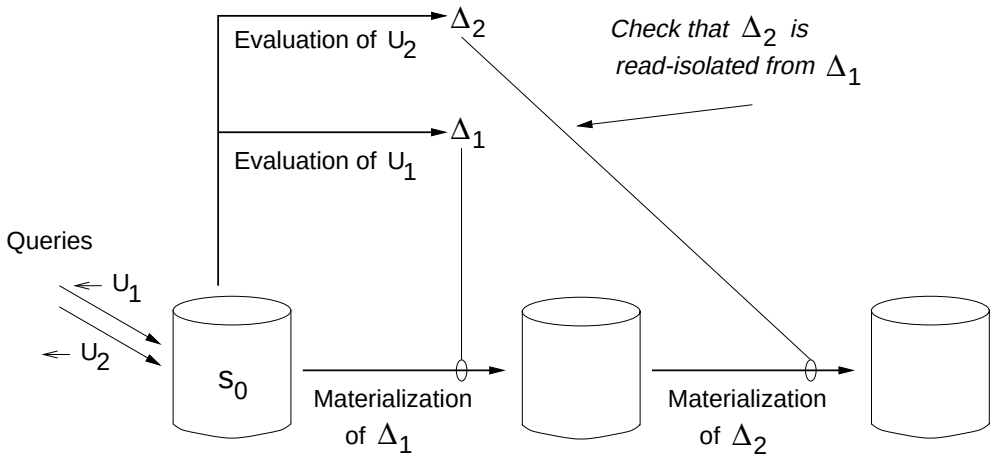

Figure 10: Serializability of two (top-level) transactions

Corollary 5.14 [Serializability] Let P_{UP}, s_0, U_1, Δ_1, U_2, and Δ_2 be given as in Corollary 5.12 and Remark 5.13. Recall that Δ_2 is assumed to be read-isolated from Δ_1. Then the following holds:

$$(\Delta_\varepsilon, \Delta_1 \oplus \Delta_2) \in M_{UP}[P_{UP}, s_0](U_1 : U_2)$$

Proof: The assertion follows directly from Corollary 5.12 using case (SCj) of Definition 4.9. □

Like in Remark 5.13 the possible transitions Δ_1 and Δ_2 can be computed in parallel, because they both refer to the same initial state s_0. Nevertheless, the sequential composition $\Delta_1 \oplus \Delta_2$ of Δ_1 and Δ_2 is a possible transition for the sequential goal $U_1 : U_2$. In other words, Corollary 5.14 reformulates serializability in terms of the underlying logical concepts, i.e. the sequential conjunction.

Until now, we have only considered a simultaneous *evaluation* of multiple transactions, but we have required that the *materialization* is performed in a strictly serial way. By the following remark, we suggest how to make the materialization phase more flexible.

Remark 5.15 [Materialization of Transactions] Have a look at property 4 (a) of Definition 4.2. In the light of Corollary 5.12, it can be interpreted as offering an alternative way to deal with the computed transitions. Instead of materializing Δ_1 and Δ_2 in a serial fashion, one can compute and materialize a final transition $\Delta := \Delta_1 \oplus \Delta_2$. The computation of Δ can be interpreted as a (sequential) merging of two local transitions into one global transition. Additional algebraic properties of $\mathcal{T}$ and the composition functions $\sqcup$ and $\oplus$ can be exploited for optimization purposes during the materialization phase. This will be exemplified in Section 5.4. □

Finally, there is the question what to do, when both transactions are logically successful, but there exist no possible transitions that satisfy the read-isolation criterion. As serializability is not guaranteed in this case, one has to abort one of the transactions and restart (recompute) it, after the other transaction has been committed. Note that the transaction restarted in the new physical state may fail logically, as the possible transitions for an update query depend on the initial state.

The results of Corollaries 5.12 and 5.14 can easily be extended to multiple transactions that are processed in parallel. This will serve as the foundation of an optimistic transaction protocol we will present below.

Remark 5.16 [Multiple Transactions] Let P_{UP} be an update program and $s_0 \in \mathcal{S}$ be the initial state. Let $\leftarrow U_1, \ldots, \leftarrow U_n$ be update queries that have possible transitions (w.r.t. s_0) $\Delta_1, \ldots, \Delta_n \in \mathcal{T}_{Cons}$, respectively. Further, let Δ_i be read-isolated from Δ_j for $i, j \in \{1, \ldots, n\}$ with $i > j$.

Then the serialization of the underlying transactions is possible using the materialization order $\Delta_1, \ldots, \Delta_n$. Moreover, $\Delta_1 \oplus \ldots \oplus \Delta_n$ is a possible transition for the sequential update query $\leftarrow U_1 : \ldots : U_n$.

Proof: Applying Theorem 5.11 repeatedly, it is possible to show by induction that the two assertions

$$\begin{array}{lcl} (\Delta_1 \oplus \ldots \oplus \Delta_{i-1}, \Delta_i) & \in & M_{UP}[P_{UP}, s_0](U_i) \\ (\Delta_\varepsilon, \Delta_1 \oplus \ldots \oplus \Delta_i) & \in & M_{UP}[P_{UP}, s_0](U_1 : \ldots : U_i) \end{array}$$

hold for every $i \in \{1, \ldots, n\}$. The main assertion follows from the case $i = n$. Recall that property 4 of Definition 4.2 states that sequential compositions of transitions harmonize with serial executions. □

Before we are going to illustrate the semantical results of this section in the context of the formerly defined ULTRA instances, we would like to summarize how the results can be used in practice. We sketch a transaction protocol that is suitable for an operational model based on deferred materializations (cf. Section 8.1). Example 5.23 in Section 5.4 will discuss the execution behaviour of some transactions on the personal calendar (see Appendix B).

Remark 5.17 [Optimistic Transaction Protocol] In our operational model, transactions invoked by top-level update queries are processed block-wise in two phases: an *evaluation phase*,

where the transactions simultaneously operate on the current initial state s_0 without making changes visible to each other, and a subsequent *materialization phase*, where each transaction may execute one possible transition. The transactions are processed as follows:

1. Accept (or restart) independent update queries $\leftarrow U_1, \ldots, \leftarrow U_n$ for evaluation, as long as the materialization of the active transactions has not begun, yet.
2. Evaluate the active transactions hypothetically w.r.t. the current initial state s_0. Synchronization between different evaluation threads is not necessary, as all queries are evaluated w.r.t. the same state and no state changes are actually performed.
3. If a transaction invoked by a query $\leftarrow U_i$ is ready to enter the materialization phase, i.e. it provides a possible transition Δ_i, check whether this transition is read-isolated from all transitions collected so far. If the read-isolations hold, collect the new transition for materialization. Otherwise, resume the evaluation of the query $\leftarrow U_i$ in order to compute another possible transition, or abort the corresponding transaction.
4. When the evaluation of the accepted transactions has been finished, materialize the collected transitions in the given order. This way, the system reaches a new physical state s_0'. Commit those transactions with a successful materialization and abort the other ones. Note that the materializations must be performed as external "write-only" transactions to ensure atomicity and durability of the materializations.

Due to Remark 5.16, the protocol ensures serializability of the independent transactions. The correctness is also guaranteed, if some of the transactions fail during the materialization phase: these transactions do not have an effect and thus can be considered as non-existent.

Failed transactions should be restarted in the new physical state, when they have failed due to isolation conflicts with other transactions or due to materialization errors. A transaction that logically fails, however, should be definitively aborted, as the success of a repeated evaluation is unlikely. Although the transaction might be successful in a new physical state, the decision about a restart is not the task of the transaction protocol.

It should be noted that the protocol described above can easily be enriched by additional constraints, e.g. an upper limit of active transactions in an evaluation phase, time-out constraints, etc. Such constraints can serve as tuning parameters to enable more serial or more concurrent and optimistic transaction processing. □

5.4 Read-Isolation in Logic Databases

Now we intend to apply the results of Section 5.3 to the database language presented in Section 3.2. Recall that the transitions in $\mathcal{T}$ are represented by update request sets containing insertions, deletions, and read tags. First we define a concrete read-isolation relation on the set of update request sets, then we show that it entails the properties required in Definition 5.7. Again, the general results of Section 5.3 lead to the specific results presented in [WFF98b].

Definition 5.18 [Read-Isolation] An update request set $\Delta \in \mathcal{T}$ is called *read-isolated* from another update request set $\Delta_0 \in \mathcal{T}$, if there exists no update request $+r(\vec{t}) \in \Delta_0$ or $-r(\vec{t}) \in \Delta_0$ such that the read tag $?r$ is contained in Δ. □

Informally speaking, Δ is called read-isolated from Δ_0, iff the basic update requests of Δ_0 cannot imply changes in EDB relations for which a read tag exists in Δ. Recall that read tags correspond to retrieval operations during the computation of Δ. Thus, read-isolation guarantees the absence of read/write conflicts.

Example 5.19 [Personal Calendar (Cont.)] Given the two transactions

$$\begin{aligned} T_1 &\equiv \leftarrow do\_insert(mon, 10, 1, \text{“Presentation”}) \\ T_2 &\equiv \leftarrow browse(mon, S, T) \end{aligned}$$

we get the following update request sets Δ_i for T_i, assuming the assigned identifier is 23:

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

Observe that Δ_1 is read-isolated from Δ_2, as the latter contains only read tags, while Δ_2 is not read-isolated from Δ_1, which actually contains update requests for the relations *entry* and *description*. □

The following semantical property holds for the defined read-isolation relation. Under the given conditions, the updates of Δ_0 do not have an observable effect w.r.t. the predicate q.

Lemma 5.20 [Independence] Let $DB \in \mathcal{S}$ be an EDB instance and P_{IDB} be an IDB program. Let q be an EDB or an IDB predicate, and let $\Delta_0 \in \mathcal{T}_{Cons}$ be a consistent update request set such that $\Delta_q := \{?r \mid r \in Def_E[P_{IDB}](q)\}$ is read-isolated from Δ_0. Let $\Delta \in \mathcal{T}_{Cons}$ be another consistent update request set.

Then the restriction of $WFM(P_{IDB} \cup ((DB \oplus_E \Delta_0) \oplus_E \Delta))$ to the predicate q equals the restriction of $WFM(P_{IDB} \cup (DB \oplus_E \Delta))$ to q.

Proof: Δ_q is read-isolated from Δ_0 by assumption and contains read tags for all EDB predicates q depends on. Thus, Δ_0 cannot contain any update request to change an EDB relation that is relevant for the interpretation of any EDB/IDB atom $q(\vec{t})$, i.e. $DB \oplus_E \Delta_0$ restricted to EDB predicates in $Def_E[P_{IDB}](q)$ equals DB restricted to $Def_E[P_{IDB}](q)$. Hence, $(DB \oplus_E \Delta_0) \oplus_E \Delta$ restricted to EDB predicates in $Def_E[P_{IDB}](q)$ equals $DB \oplus_E \Delta$ restricted to $Def_E[P_{IDB}](q)$.

Consequently, the restriction of $WFM(P_{IDB} \cup ((DB \oplus_E \Delta_0) \oplus_E \Delta))$ to the predicate q equals the restriction of $WFM(P_{IDB} \cup (DB \oplus_E \Delta))$ to q.

Note that is sufficient to consider the syntactical dependencies, because the well-founded model has a fixpoint characterization [vG89] which is essentially built on consequence operators based on the given facts and rules. □

Theorem 5.21 [Algebraic Properties] The algebraic properties of Definition 5.7 hold for the read-isolation relation defined for the extended database language (cf. Definition 5.18).

Proof: The proof relies on the definitions of $\oplus_E$, $\sqcup$, $\oplus$, and read-isolation. Also Lemma 5.20 is essential.

1. Note that in both compositions ($\sqcup$ and $\oplus$), the read tags are merged like in the usual set union. Thus, for arbitrary update request sets $\Delta_0, \Delta_1, \Delta_2 \in \mathcal{T}$ such that $\Delta_1 \sqcup \Delta_2$ is read-isolated from Δ_0 or $\Delta_1 \oplus \Delta_2$ is read-isolated from Δ_0, also both Δ_1 and Δ_2 must be read-isolated from Δ_0. Otherwise, Δ_1 or Δ_2 would contain a conflicting read tag $?r$, which would be also contained in $\Delta_1 \sqcup \Delta_2$ and $\Delta_1 \oplus \Delta_2$ and which thus would destroy the read-isolation property. Consequently, properties (a) and (b) hold.

2. Recall that by definition

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

holds for each EDB instance $DB \in \mathcal{S}$. Further

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

holds for every ground DB atom A with predicate q. The desired equivalences (a) and (b) follow directly from Lemma 5.20. □

Theorem 5.21 allows us to apply the isolation and serializability results of Section 5.3 to the specific database language. In particular, we can use the transaction processing protocol described in Remark 5.17. Before we illustrate the results using our calendar example, some additional remarks about the materialization should follow.

Remark 5.22 [Materialization of Transactions] Recall Remark 5.15, in particular the two possible transitions Δ_1 and Δ_2, that have passed the isolation check and wait for materialization.

Now assume that Δ_1 and Δ_2 are write-compatible: Corollary 4.25 states that it is possible to materialize the concurrent composition $\Delta_1 \sqcup \Delta_2$. This allows us to apply the update requests of both transactions simultaneously (e.g. in an interleaved fashion). Such a property can be used for optimization purposes (e.g. sorting of update requests, writing of contiguous blocks, etc.) during the materialization phase. □

Example 5.23 [Personal Calendar (Cont.)] Let us consider some sample transactions issued against the database instance DB_0 given in Example 3.17. The transactions could be invoked by two different users, e.g. the owner of the calendar and her secretary. For the sake of brevity, we restrict ourselves to two transactions that are processed independently. The protocol described in Remark 5.17, however, can deal with more than two independent transactions.

First, have a look at transactions T_1 and T_2 of Example 5.19. As stated there, the update request set Δ_1 is read-isolated from Δ_2, so it makes sense to evaluate both transactions in parallel w.r.t. the same database state DB_0. According to Remark 5.13, the materialization of Δ_1 after Δ_2 corresponds to the serial execution of T_1 after T_2.

If two transactions read and write the same objects, a conflict arises and must be handled, as the following example shows:

$$\begin{array}{lcl} T_3 & \equiv & \leftarrow do\_insert(mon, 10, 1, \text{“Presentation”}) \\ T_4 & \equiv & \leftarrow do\_insert\_on\_day(mon, 1, \text{“Call Mr. Martin”}) \end{array}$$

Transaction T_3 is the same as T_1 above and thus produces the same update request set. T_4 corresponds to query Q of Example 5.3. So we get the update request set

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

for transaction T_3 and three alternatives for T_4, namely:

$$\begin{aligned}\Delta_4^{(1)} \;=\; \{&?entry, -entry(mon, 10, 0), +entry(mon, 10, 28),\\ &+description(28, \text{“Call Mr. Martin”})\}\\ \Delta_4^{(2)} \;=\; \{&?entry, -entry(mon, 11, 0), +entry(mon, 11, 28),\\ &+description(28, \text{“Call Mr. Martin”})\}\\ \Delta_4^{(3)} \;=\; \{&?entry, -entry(mon, 14, 0), +entry(mon, 14, 28),\\ &+description(28, \text{“Call Mr. Martin”})\}\end{aligned}$$

None of the computed sets $\Delta_4^{(1)}$ through $\Delta_4^{(3)}$ is read-isolated from Δ_3, and Δ_3 is not read-isolated from any of the sets $\Delta_4^{(1)}$ through $\Delta_4^{(3)}$, so the two transactions cannot be serialized anymore, when they both have been evaluated in database state DB_0. Therefore, the system has to choose one transaction, while restarting the other after the materialization of the first.

Observe that if the system chooses to restart T_4 while materializing T_3, in the new database state it is no longer possible to insert the entry at 10am, as this time slot now is occupied by the “Presentation”. So, for the restarted T_4, the only possible transitions will be $\Delta_4^{(2)}$ and $\Delta_4^{(3)}$.

On the other hand, if the system decides to materialize T_4 and to restart T_3, the success of T_3 depends on the non-deterministic choice in transaction T_4. If the second or the third update request set is chosen, the restarted transaction T_3 will succeed, as the time slot at 10am remains free. But if the system chooses to insert the “Call” into the slot at 10am, the restarted T_3 will fail, because the time slot at 10am will be occupied.

It should be kept in mind, that the conflict between T_3 and T_4 is not caused by the fact that both *write* to a common relation, but that both *read* a relation that *the other one writes*. Two write operations on the same relation can of course be serializable, as the transactions

$$\begin{aligned}T_5 \;&\equiv\; \leftarrow do\_insert\_on\_day(mon, 1, \text{“Call Mr. Martin”})\\ T_6 \;&\equiv\; \leftarrow do\_insert\_priority(mon, 14, \text{“Dentist”})\end{aligned}$$

show. The predicate *do_insert_priority* (see Appendix B) enters appointments without checking whether or not the requested time slot is free. When evaluated in parallel w.r.t. state DB_0, transaction T_6 creates the update request set

$$\begin{aligned}\Delta_6 \;=\; \{&-entry(mon, 14, 0), +entry(mon, 14, 25),\\ &+description(25, \text{“Dentist”})\}\end{aligned}$$

assuming the assigned identifier is 25. For T_5, the same possible transitions as for T_4 are used in this example, i.e. $\Delta_5^{(1)} = \Delta_4^{(1)}$, $\Delta_5^{(2)} = \Delta_4^{(2)}$, and $\Delta_5^{(3)} = \Delta_4^{(3)}$. Although both transactions write to the same EDB relation and none of the sets $\Delta_5^{(1)}$ through $\Delta_5^{(3)}$ is read-isolated from Δ_6, Δ_6 is read-isolated from $\Delta_5^{(1)}$, $\Delta_5^{(2)}$, and $\Delta_5^{(3)}$. So, the two transactions can be evaluated in parallel in

the same database state DB_0, but one of the three possible transitions for T_5 must be chosen and materialized before Δ_6 is applied to the database.

Note that it is no error to have both $entry(mon, 14, 25)$ and $entry(mon, 14, 28)$ in the database, as the former is inserted by *do_insert_priority*, which does not explicitly check the slot and thus allows overlapping entries. For the sake of the example, we do not remove the old entry, as this would require a read access, which in turn destroys the read-isolation property.

We conclude our running example with a reference to Remark 5.22. As write/write conflicts are defined on a finer granularity than read/write conflicts and in the calendar example most write operations are accompanied by read operations on the same relation, write/write conflicts between independent transactions which have already been certified to be serializable are unlikely. Indeed, in the example cases investigated above the certified possible transitions are always write-compatible with each other, i.e. there are no requests to insert and delete the same tuple. Thus, due to Remark 5.22, also the materialization of the certified possible transitions can be performed simultaneously. In this case, the materialization does not correspond to a serial execution of the transactions, as the intermediate state between the transactions does not need to be reached during the materialization. However, the final state will be the same state as if the materialization was done in a serial fashion.

□

5.5 Read-Isolation of Pomsets

As opposed to the database-oriented ULTRA instance, the ULTRA instance based on partially ordered multi-sets (see Sections 3.3 and 4.4) has been defined with external operations and immediate execution strategies (cf. Section 8.2) in mind. Nevertheless, the pomset approach is very universal and could serve as the semantical basis in other domains, where a transaction processing strategy based on deferred materialization and hypothetical reasoning is adequate. Furthermore, recall that the results about isolation of transactions are not restricted to the top-level and could thus be relevant for hybrid strategies that interleave multiple local evaluation and materialization phases within a transaction. For these reasons, we are going to define a read-isolation property for partially ordered multi-sets, too, such that we are open to refer to the results of Section 5.3.

To fit with the optimistic transaction protocol of Remark 5.17, the system on which the transactions are performed must enable the hypothetical reasoning about the effects of deferred operations. Either the system behaviour has to be axiomatized, such that it can be investigated by formal reasoning methods, or a simulation model of the system has to be provided. For instance, an industrial robot might be simulated using physical models or software models, before the (possibly non-retractable) actions are finally carried out by the robot. We will adopt this point of view in the example presented subsequently.

Definition 5.24 [Read-Isolation] A pomset $[V, \leq, \mu] \in \Sigma^{\ddagger}$ is called *read-isolated* from another pomset $[V_0, \leq_0, \mu_0] \in \Sigma^{\ddagger}$, where V and V_0 are chosen disjoint (w.l.o.g.), if for arbitrary events $e \in V$ and arbitrary DB atoms A (ground or with variables $X_1, \ldots, X_n$) the following holds:

> If $\mu(e) = Log^{act}(A[\,\vec{X}\,/\,\vec{all}\,])$, then for all $e' \in V_0$ and for all ground tuples $(\,\vec{t}\,) \in \mathcal{U}^n$, $A[\,\vec{X}\,/\,\vec{t}\,]$ is independent of $\mu_0(e')$.

□

The definition of read-isolation is well-defined. The isolation property depends only on the labelling functions μ and μ_0. Informally speaking, a pomset Δ is read-isolated from another pomset Δ_0, iff Δ_0 does not contain any action a that affects the truth value of a DB atom for which a logging action exists in Δ.

Lemma 5.25 Let $\Delta_0, \Delta_1, \Delta_2 \in \Sigma^\ddagger$ be arbitrary pomsets. Then the following equivalences hold:

$$\begin{array}{ll} & \Delta_1 \text{ read-isolated from } \Delta_0 \ \wedge\ \Delta_2 \text{ read-isolated from } \Delta_0 \\ \iff & \Delta_1 \sqcup \Delta_2 \text{ read-isolated from } \Delta_0 \\ & \Delta_1 \text{ read-isolated from } \Delta_0 \ \wedge\ \Delta_2 \text{ read-isolated from } \Delta_0 \\ \iff & \Delta_1 \oplus \Delta_2 \text{ read-isolated from } \Delta_0 \end{array}$$

Proof: Choose (w.l.o.g.) representatives $[V_0, \leq_0, \mu_0]$, $[V_1, \leq_1, \mu_1]$, and $[V_2, \leq_2, \mu_2]$ for Δ_0 to Δ_2, where the V_i are disjoint. By Definitions 4.36 and 4.37, the event sets of $\Delta_1 \sqcup \Delta_2$ and $\Delta_1 \oplus \Delta_2$ are both $V_1 \cup V_2$, and the labelling functions are both $\mu_1 \cup \mu_2$. Since $V_1 \cup V_2$ contains exactly the events of V_1 and V_2, the desired properties can be shown easily. □

The following theorem shows that also the read-isolation property defined in this section fits into the generic ULTRA framework. This allows us to apply the isolation and serializability results of Section 5.3.

Theorem 5.26 [Algebraic Properties] The algebraic properties of Definition 5.7 hold for the read-isolation relation defined for pomsets.

Proof: Property 1 follows directly from Lemma 5.25. So, we just have to verify property 2.

Let $\Delta_0 \in \mathcal{T}_{Cons}$ be a consistent transition and A be a DB atom (containing the variables $\vec{X}$) with the ground instance $A' := A[\,\vec{X} \,/\, \vec{all}\,]$, such that $Log(A')$ is read-isolated from Δ_0. Let A'' be an arbitrary ground instance of A, let $s \in \mathcal{S}$ be a state, and let $\Delta \in \mathcal{T}_{Cons}$ be another consistent transition.

First, we will find representations of $(s \oplus_E \Delta_0) \oplus_E \Delta$ and $s \oplus_E \Delta$. Let $\Delta_0' \in \Sigma^\ddagger$ and $\Delta' \in \Sigma^\ddagger$ be some linearizations of Δ_0 and Δ, respectively. For the sake of brevity, we assume that Δ_0 has the list representation $[a_1, \ldots, a_m]$ and that Δ has the list representation $[b_1, \ldots, b_n]$ (cf. Remark 4.48). Applying Definition 4.53 inductively, it is possible to show that the following equalities hold:

$$\begin{array}{lcl} (s \oplus_E \Delta_0) \oplus_E \Delta & = & do(b_n, do(\ldots, do(b_1, do(a_m, do(\ldots, do(a_1, s)..)))..)) \\ s \oplus_E \Delta & = & do(b_n, do(\ldots, do(b_1, s)..)) \end{array}$$

Next, consider the read-isolation condition of Definition 5.24. As $Log(A')$ is read-isolated from Δ_0 by assumption and contains one event labelled with $Log^{act}(A[\,\vec{X} \,/\, \vec{all}\,])$, A'' must be independent of each a_i ($i \in \{1, \ldots, m\}$). Applying Definition 4.28 inductively, one can easily show the desired equivalences. □

Example 5.27 [Robot World (Cont.)] Recall Example 3.20 and consider the recursively defined operations *xmove* and *ymove*. Let us assume that the robot is at position $(1, 1)$ in the initial state s_0 and the following transactions have been invoked:

$$\begin{array}{lcl} T_1 & \equiv & \leftarrow xmove(3) \\ T_2 & \equiv & \leftarrow ymove(3) \end{array}$$

For each update query T_i, there exists exactly one possible transition Δ_i:

$$\begin{aligned}\Delta_1 &= [?xpos, xstep(1), ?xpos, xstep(1), ?xpos]\\ \Delta_2 &= [?ypos, ystep(1), ?ypos, ystep(1), ?ypos]\end{aligned}$$

In Example 4.31 we have already justified that arbitrary DB atoms of the form $ypos(\ldots)$ are independent of the actions $?xpos$ and $xstep(1)$. Further, there are no other DB atoms that have the logging transition $?ypos$. Consequently, the pomsets $\{?ypos\}$ and Δ_2 are both read-isolated from the pomset Δ_1 by Definition 5.24. It should be noted that also Δ_1 is read-isolated from Δ_2, since the x- and y-components can be exchanged.

The results of Section 5.3 can be applied within an operational model based on deferred external actions. For instance, let us consider the following environment: In state s_0, a snapshot of the robot world is taken by a camera, preprocessed, and distributed to two computer systems that allow (local) simulations on the (virtual) robot world. If T_1 is evaluated using the first and T_2 using the second simulator, then the locally computed transitions Δ_1 and Δ_2 can be materialized later – either Δ_2 after Δ_1 or vice versa –, while the effect on the real robot world is the same as if the transactions were executed strictly in serial. Unfortunately, the robot example presented in this thesis is too simple to feature more interesting transactions that satisfy the isolation conditions. □

5.6 Read-Isolation and Stronger Constraints

Whenever a valid read-isolation relation R is found for an ULTRA instance, the relation represents a sufficient condition for isolation checks on the basis of computed transitions. In this section we claim that stronger conditions can be used as well. They may lead to more restrictive protocols but do not compromise the correctness results.

Proposition 5.28 Let R be a read-isolation relation according to Definition 5.7 and let Φ be another binary relation on the set $\mathcal{T}$ of transitions, such that the inclusion $\Phi \subseteq R$ holds, i.e.

$$\Phi(\Delta, \Delta_0) \implies \Delta \text{ read-isolated from } \Delta_0$$

holds for arbitrary transitions $\Delta, \Delta_0 \in \mathcal{T}$. Then Theorem 5.11 and its consequences also hold, if the read-isolation property R is continuously replaced by the property Φ.

Proof: As the read-isolation property always occurs as a precondition, it can be replaced by the stronger property Φ without invalidating the formal results. □

Remark 5.29 Let R and Φ be as in Proposition 5.28, in particular, with $\Phi \subseteq R$. If further property 1 of Definition 5.7 holds for Φ analogously, i.e. the conditions

$$\begin{aligned}&(a)\quad \Phi(\Delta_1 \sqcup \Delta_2, \Delta_0) \implies \Phi(\Delta_1, \Delta_0) \wedge \Phi(\Delta_2, \Delta_0)\\ &(b)\quad \Phi(\Delta_1 \oplus \Delta_2, \Delta_0) \implies \Phi(\Delta_1, \Delta_0) \wedge \Phi(\Delta_2, \Delta_0)\end{aligned}$$

hold for arbitrary $\Delta_1, \Delta_2 \in \mathcal{T}$ and $\Delta_0 \in \mathcal{T}_{Cons}$, then Φ is a valid read-isolation relation.

Proof: Property 1 of Definition 5.7 holds by the precondition. Property 2 follows immediately from the inclusion $\Phi \subseteq R$. □

We would like to give an outline of how to apply Proposition 5.28 to the ULTRA instance tailored to external operations.

Example 5.30 Recall the ULTRA instance that has been presented in Sections 3.3, 4.4, and 5.5. In particular, recall Definition 5.24, where the read-isolation property has been defined using the independence property (cf. Definition 4.28). This notion of independence has been formulated at the *semantical* level and might be inadequate for an operational treatment (see [Elk90] for more information about the problem of checking independence). However, in analogy to the consistency example discussed in Section 4.5.3, it would be possible to handle independence at the *syntactical* level. For this purpose, the specific instance has to be extended by an explicit independence relation. We require that the independence assertions are correct, i.e. that syntactical independence entails semantical independence.

If we now define an alternative version of read-isolation by replacing semantical independence by syntactical independence in Definition 5.24, the results presented in Section 5.3 will also hold w.r.t. the new property (corresponding to the parameter Φ in Proposition 5.28), which will obviously entail the semantical read-isolation. In particular, we can use the new property for an optimistic transaction processing. Moreover, the syntactical notion of read-isolation forms a valid read-isolation relation: the additional properties listed in Remark 5.29 can easily be shown. We decided to define read-isolation first of all at the semantical level, because the semantical notion is more general.

Recall the remark about view serializability and conflict serializability in Example 4.70. If we define read-isolation in terms of semantical independence, we can consider isolated transactions (cf. Section 5.3) as view serializable, since the isolation criterion is based on state observations. Similarly, a syntactical independence relation leads to a notion of conflict serializability, where conflicts correspond to missing independence assertions. Note that the serializability criteria in [BHG87] also take compatibilities between basic update operations into account. These compatibilities can be neglected in our operational model, because only the evaluations run concurrently, while the materializations are performed in a serial fashion. □

6 Semantical Properties of Language Constructs and Programs

In this section we are going to identify properties that hold for ULTRA constructs or update programs written in the ULTRA language. The properties refer to the generic ULTRA framework and hold regardless of any specific instance. Most of the assertions shown below can be exploited for rewriting update programs without changing their model-theoretic semantics. The assertion shown in Section 6.4 refers to minimal models of one update program but w.r.t. different initial states. On the one hand, the properties have theoretical relevance, as they demonstrate that the ULTRA semantics is a well-defined extension of the declarative Datalog semantics [Llo87], on the other hand, the properties may be helpful in practice, when evaluation methods and optimization strategies have to be developed.

6.1 Algebraic Properties of the Connectives

The connectives of the ULTRA language have several algebraic properties. The properties can be used when rewriting update formulas. Moreover, they allow simplified representations of nested formulas.

Proposition 6.1 [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 proof is straight-forward mainly using Definition 4.9 and the algebraic properties required in Definition 4.2. 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

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\varphi, \psi) \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\iff & \text{there exist } \Delta_2, \Delta_1 \in \mathcal{T}_{Cons} \text{ such that :} \\
\overset{(4.2)}{} & (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta = \Delta_2 \sqcup \Delta_1 \\
\iff & (\Delta_C, \Delta) \in I(\psi, \varphi)
\end{array}
$$

(b) Associativity

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\ [\varphi, \psi], \chi\ ) \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_{1-2}) \in I(\varphi, \psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \text{ and } \Delta = \Delta_{1-2} \sqcup \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_1, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \\
 & \text{and } \Delta_{1-2} = \Delta_1 \sqcup \Delta_2 \text{ and } \Delta = \Delta_{1-2} \sqcup \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3}, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
\scriptstyle (4.2) & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(\psi) \text{ and } (\Delta_C, \Delta_3) \in I(\chi) \\
 & \text{and } \Delta_{2-3} = \Delta_2 \sqcup \Delta_3 \text{ and } \Delta = \Delta_1 \sqcup \Delta_{2-3} \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3} \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_{2-3}) \in I(\psi, \chi) \text{ and } \Delta = \Delta_1 \sqcup \Delta_{2-3} \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\ \varphi, [\psi, \chi]\ )
\end{array}
$$

(c) Neutral element

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\varphi, NOP) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I(NOP) \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta_2 = \Delta_\varepsilon \text{ and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\varphi) \\
\scriptstyle (4.2) &
\end{array}
$$

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

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\ [\varphi : \psi] : \chi\ ) \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_{1-2}) \in I(\varphi : \psi) \text{ and } (\Delta_C \oplus \Delta_{1-2}, \Delta_3) \in I(\chi) \text{ and } \Delta = \Delta_{1-2} \oplus \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_{1-2}, \Delta_1, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \text{ and } (\Delta_C \oplus \Delta_{1-2}, \Delta_3) \in I(\chi) \\
 & \text{and } \Delta_{1-2} = \Delta_1 \oplus \Delta_2 \text{ and } \Delta = \Delta_{1-2} \oplus \Delta_3 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3}, \Delta_2, \Delta_3 \in \mathcal{T}_{Cons} \text{ such that :} \\
\scriptstyle (4.2) & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi) \\
 & \text{and } ((\Delta_C \oplus \Delta_1) \oplus \Delta_2, \Delta_3) \in I(\chi) \\
 & \text{and } \Delta_{2-3} = \Delta_2 \oplus \Delta_3 \text{ and } \Delta = \Delta_1 \oplus \Delta_{2-3} \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_{2-3} \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_{2-3}) \in I(\psi : \chi) \text{ and } \Delta = \Delta_1 \oplus \Delta_{2-3} \\
\Longleftrightarrow & (\Delta_C, \Delta) \in I(\ \varphi : [\psi : \chi]\ )
\end{array}
$$

(b) Neutral element

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\varphi : NOP) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(NOP) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(\varphi) \text{ and } \Delta_2 = \Delta_\varepsilon \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\underset{(4.2)}{\Longleftrightarrow} & (\Delta_C, \Delta) \in I(\varphi)
\end{array}
$$

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(NOP : \varphi) \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_C, \Delta_1) \in I(NOP) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\varphi) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\Longleftrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & \Delta_1 = \Delta_\varepsilon \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\varphi) \text{ and } \Delta = \Delta_1 \oplus \Delta_2 \\
\underset{(4.2)}{\Longleftrightarrow} & (\Delta_C, \Delta) \in I(\varphi)
\end{array}
$$

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

As already mentioned in Section 4.2, it is legitimate to omit the precedence brackets in update formulas of the form $\varphi_1, \ldots, \varphi_n$, $\varphi_1 : \ldots : \varphi_n$, and $\varphi_1 \vee \ldots \vee \varphi_n$, because the associativity justifies arbitrary decompositions of these formulas. The commutativity of "," and $\vee$ allows reorderings, and the neutral update literal NOP can generally be eliminated from conjunctions.

6.2 Quantifications as Abbreviations

In this section we show that the existential quantification can be considered as an extension of the disjunction. Similarly, the bulk quantifier extends the concurrent conjunction. Under finiteness constraints, the quantifiers behave like abbreviations. This is formalized in the following propositions.

Proposition 6.2 Let the Herbrand universe $\mathcal{U}$ be finite. Let $I \in \mathcal{I}$ be an interpretation of update formulas, and let $\psi :\equiv \exists \vec{X}\, \varphi$ be an existentially quantified update formula without free variables, where $\vec{X} = X_1, \ldots, X_n$.

Let $\varphi_1, \ldots, \varphi_k$ be a finite sequence of ground instances of φ that contains each instance $\varphi[\, \vec{X} \,/\, \vec{t}\,]$ with $(\vec{t}\,) \in \mathcal{U}^n$ at least once (but possibly more than once). Define the disjunctive update formula $\psi' :\equiv \varphi_1 \vee \ldots \vee \varphi_k$.

Then $I(\psi) = I(\psi')$.

Proof: We show the assertion by comparing the elements $(\Delta_C, \Delta) \in \mathcal{T}_{Cons} \times \mathcal{T}_{Cons}$.

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in I(\exists\, \vec{X}\, \varphi) \\
\iff & \text{there exists a ground term tuple } (\,\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
\underset{(4.9)}{} & (\Delta_C, \Delta) \in I(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\
\iff & \text{there exists } i \in \{1, \ldots, k\} \text{ such that :} \\
 & (\Delta_C, \Delta) \in I(\varphi_i) \\
\iff & (\Delta_C, \Delta) \in I(\varphi_1 \vee \ldots \vee \varphi_k) \\
\underset{(see\ below)}{} &
\end{array}
$$

The last step follows inductively from case (Dj) of Definition 4.9. Recall that the commutativity and the associativity of $\vee$ have already been shown in Proposition 6.1. □

To prove a relation between the bulk quantification and the concurrent conjunction, we need the following lemma, which shows that the concurrent composition $\bigsqcup$ of multi-sets of transitions naturally extends the concurrent composition $\sqcup$ of two transitions.

Lemma 6.3 Let $\Delta_1, \ldots, \Delta_k$ be a finite, non-empty sequence of transitions $\Delta_i \in \mathcal{T}$. Define T as the multi-set built from the items $\Delta_1, \ldots, \Delta_k$. Then the equality

$$\Delta_1 \sqcup \ldots \sqcup \Delta_k = \bigsqcup T$$

holds.

Proof: The assertion follows from Definition 4.2, in particular from property 5. We give a formal proof based on induction.

Base case: $k = 1$

$$\Delta_1 = \Delta_1 \sqcup \Delta_\varepsilon = \Delta_1 \sqcup \bigsqcup \emptyset = \bigsqcup(\{\Delta_1\} \uplus \emptyset) = \bigsqcup\{\Delta_1\}$$

Induction step: $k \to k+1$

By the induction hypothesis, the assertion holds for every multi-set of k elements, in particular for the multi-set T' built from $\Delta_1, \ldots, \Delta_k$. Now we can prove the assertion for the multi-set T built from the items $\Delta_1, \ldots, \Delta_{k+1}$.

$$
\begin{array}{ll}
 & \Delta_1 \sqcup \ldots \sqcup \Delta_{k+1} = \Delta_{k+1} \sqcup (\Delta_1 \sqcup \ldots \sqcup \Delta_k) \\
\underset{(IH)}{=} & \Delta_{k+1} \sqcup \bigsqcup T' = \bigsqcup(\{\Delta_{k+1}\} \uplus T') = \bigsqcup T
\end{array}
$$

□

Proposition 6.4 Let $I \in \mathcal{I}$ be an interpretation of update formulas, and let $\psi :\equiv \#\,\vec{X}\,[A \mapsto \varphi]$ be a ground bulk quantification formula, where $\vec{X} = X_1, \ldots, X_n$. Let Δ_C be an arbitrary consistent transition such that

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

is finite and has a cardinality of k.

If $k > 0$, let $\varphi_1, \ldots, \varphi_k$ be an enumeration of the ground instances $\varphi[\vec{X} / \vec{t}\,]$ with $(\vec{t}\,) \in T_{A,\Delta_C}$. Note that each formula φ_i may occur more than once within the enumeration, if not all quantified variables occur free in φ. Define the concurrent conjunction $\psi' :\equiv \varphi_1, \ldots, \varphi_k$.

Otherwise, i.e. if $k = 0$, define $\psi' :\equiv NOP$.

Then for arbitrary consistent transitions $\Delta \in \mathcal{T}_{Cons}$ the following implications hold:

$$\begin{array}{lcl} (\Delta_C, \Delta) \in I(\psi) & \Longrightarrow & \text{there exists } \Delta' \in \mathcal{T}_{Cons} \text{ such that :} \\ & & \Delta = Log(A[\vec{X} / \vec{all}\,]) \oplus \Delta' \text{ and } (\Delta_C, \Delta') \in I(\psi') \\ (\Delta_C, Log(A[\vec{X} / \vec{all}\,]) \oplus \Delta) \in I(\psi) & \Longleftarrow & (\Delta_C, \Delta) \in I(\psi') \end{array}$$

Proof: The case $k = 0$ is trivial. By cases (Bulk) and (NOP) of Definition 4.9 the following equivalences hold for arbitrary consistent transitions $\Delta \in \mathcal{T}_{Cons}$.

$$\begin{array}{lcl} (\Delta_C, \Delta) \in I(\psi) & \Longleftrightarrow & \Delta = Log(A[\vec{X} / \vec{all}\,]) \\ (\Delta_C, \Delta) \in I(\psi') & \Longleftrightarrow & \Delta = \Delta_\varepsilon \end{array}$$

The assertions '$\Rightarrow$' and '$\Leftarrow$' follow immediately. Recall that Δ_ε is the neutral element of $\oplus$.

Next, let us assume that $k > 0$. Let $\Delta \in \mathcal{T}_{Cons}$ be arbitrarily chosen.

'$\Rightarrow$':

Assume that $(\Delta_C, \Delta) \in I(\psi)$ holds. By case (Bulk) of Definition 4.9, there exists a function $f : T_{A,\Delta_C} \to \mathcal{T}_{Cons}$ such that

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

holds for all $(\vec{t}\,) \in T_{A,\Delta_C}$ and

$$\Delta = Log(A[\vec{X} / \vec{all}\,]) \oplus \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C}} f(\vec{t}\,).$$

Define

$$\Delta' := \bigsqcup_{(\vec{t}) \in T_{A,\Delta_C}} f(\vec{t}\,).$$

By property 3 of Definition 4.2, Δ' must be consistent, since Δ is consistent. Using Lemma 6.3, one can show that

$$\Delta' = \Delta_1 \sqcup \ldots \sqcup \Delta_k$$

where $\Delta_i = f(\vec{t}\,)$, iff $\varphi[\vec{X} / \vec{t}\,]$ is enumerated at position i in $\varphi_1, \ldots, \varphi_k$. Applying case (CCj) of Definition 4.9, it is easy to show that $(\Delta_C, \Delta') \in I(\psi')$ holds. Note that the properties of the function f imply that

$$(\Delta_C, \Delta_i) \in I(\varphi_i)$$

holds for all $i \in \{1, \ldots, k\}$.

'$\Leftarrow$':

Assume that $(\Delta_C, \Delta) \in I(\psi')$. Using case (CCj) of Definition 4.9, one can easily show that there exist consistent transitions $\Delta_1, \ldots, \Delta_k \in \mathcal{T}_{Cons}$ such that

$$(\Delta_C, \Delta_i) \in I(\varphi_i)$$

for all $i \in \{1, \ldots, k\}$ and

$$\Delta = \Delta_1 \sqcup \ldots \sqcup \Delta_k.$$

Next, define $f : T_{A,\Delta_C} \to \mathcal{T}_{Cons}$ by

$$f(\vec{t}) := \Delta_i, \text{ iff } \varphi[\vec{X} / \vec{t}] \text{ is enumerated at position } i \text{ in } \varphi_1, \ldots, \varphi_k$$

for all $(\vec{t}) \in T_{A,\Delta_C}$. By case (Bulk) of Definition 4.9 and Lemma 6.3, the conclusion $(\Delta_C, Log(A[\vec{X} / \vec{all}]) \oplus \Delta) \in I(\psi)$ follows. Note that the consistency of $Log(A[\vec{X} / \vec{all}]) \oplus \Delta$ is provable using property 3 of Definition 4.2.

□

6.3 Rewriting of Update Programs

In Proposition 6.1 we have already presented some algebraic properties that hold for the concurrent conjunction, the sequential conjunction, and the disjunction. Obviously, these properties enable simple program transformations. In this section we formalize some more rewriting techniques that leave the semantics of a given update program unchanged. Finite programs can be transformed into a normal form which does not allow nested subgoals. Furthermore, the disjunction and the existential quantification can be eliminated. The results may be useful when dealing with operational semantics and program optimization.

As in Section 4 we always refer to an arbitrary but fixed initial state $s_0 \in \mathcal{S}$.

Note that the rewriting techniques presented below have been adopted from logic database languages, where they are legitimate and commonly used (see e.g. [FSS91] about program simplification in the *LOLA* system). In this light, the properties emphasize that the ULTRA approach extends the well-known concepts not only at the syntactical level, but also at the semantical level. In particular, a high amount of declarativity is preserved. Many logical rewriting properties get lost, if a declarative language is extended by impure features that are only defined at the operational level (see [Wad95a] for some examples in the context of functional languages).

6.3.1 Auxiliary Rules for Complex Goals

In this section we show that update programs can be transformed in a folding style, such that complex goals are replaced by definable update atoms which are defined by auxiliary rules.

First, we show that the minimal model of a program is kept, if the program is simply augmented by an auxiliary rule. As is to be expected, due to the new rule, both minimal models are not equal. However, they coincide on the relevant part of the definable update base.

Definition 6.5 Let $I_1, I_2 \in \mathcal{I}$ be interpretations of update formulas and $p \in Pred_{DU}$ be a definable update predicate. I_1 and I_2 are called *equal modulo* p, denoted by $I_1 =_p I_2$, if

$$I_1(q(\vec{t})) = I_2(q(\vec{t}))$$

holds for all ground definable update atoms $q(\vec{t}) \in \mathcal{B}_{DU}$ with $q \neq p$. □

In the settings of Definition 6.5, the interpretations I_1 and I_2 may differ only for definable update atoms over the predicate p. Whenever p is an auxiliary predicate, its interpretation is irrelevant, and the equality $=_p$ is considered as sufficient.

Proposition 6.6 Let $I_1, I_2 \in \mathcal{I}$ be interpretations of update formulas and $p \in Pred_{DU}$ be a definable update predicate such that $I_1 =_p I_2$ holds. Then for arbitrary ground formulas φ that do not contain the predicate p, the following holds:

$$I_1(\varphi) = I_2(\varphi)$$

Proof: As the interpretation of an update formula is defined inductively having the definable update atoms as base cases (see Definition 4.9 for details), the assertion follows directly. □

The next lemma shows that it is legitimate to add an auxiliary rule to a program without changing the other rules.

Lemma 6.7 Let P_{UP} be an update program, and let $p \in Pred_{DU}$ be a definable update predicate which does not occur in the rules of P_{UP}. Let $p(\vec{s})$ be an arbitrary definable update atom over p and χ be an arbitrary update goal. Define P'_{UP} as the extension of P_{UP} by the new rule $p(\vec{s}) \leftarrow \chi$. Then

$$M_{UP}[P_{UP}, s_0] =_p M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: We prove the assertion by transfinite induction. For a given ordinal γ we show that

$$T_{P_{UP}} \uparrow \gamma =_p T_{P'_{UP}} \uparrow \gamma$$

holds.

Base case: $\gamma = 0$

The case is trivial, since $I_\perp(q(\vec{t})) = \emptyset$ holds for all ground definable update atoms $q(\vec{t}) \in \mathcal{B}_{DU}$.

Induction step: successor ordinal $\alpha + 1$

By the induction hypothesis, the desired equality modulo p holds for the ordinal α. By Proposition 6.6, $T_{P_{UP}} \uparrow \alpha\,(\varphi)$ equals $T_{P'_{UP}} \uparrow \alpha\,(\varphi)$ for all formulas that do not contain the predicate p.

Now choose a ground definable update atom $q(\vec{t}) \in \mathcal{B}_{DU}$ with $q \neq p$. Note that $q(\vec{t})$ is not defined by the new rule $p(\vec{s}) \leftarrow \chi$ and p does not occur in any other rules. Thus, for arbitrary consistent transitions $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ the following equivalences hold.

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in T_{P_{UP}} \uparrow \alpha + 1\,(q(\vec{t})) \\
\underset{(4.86)}{\iff} & (\Delta_C, \Delta) \in T_{P_{UP}}(T_{P_{UP}} \uparrow \alpha)(q(\vec{t})) \\
\underset{(4.83)}{\iff} & \text{there exists a ground instance } U \to q(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
 & \text{such that } (\Delta_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(U) \\
\underset{(see\ above)}{\iff} & \text{there exists a ground instance } U \to q(\vec{t}) \text{ of a rule } r \in P'_{UP}, \\
 & \text{such that } (\Delta_C, \Delta) \in T_{P'_{UP}} \uparrow \alpha\,(U) \\
\underset{(4.83)}{\iff} & (\Delta_C, \Delta) \in T_{P'_{UP}}(T_{P'_{UP}} \uparrow \alpha)(q(\vec{t})) \\
\underset{(4.86)}{\iff} & (\Delta_C, \Delta) \in T_{P'_{UP}} \uparrow \alpha + 1\,(q(\vec{t}))
\end{array}
$$

Induction step: limit ordinal β

By the induction hypothesis, the desired equality modulo p holds w.r.t. all ordinals $\alpha < \beta$.

Now choose a ground definable update atom $q(\vec{t}) \in \mathcal{B}_{DU}$ with $q \neq p$. For arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ the following equivalences hold.

$$
\begin{array}{ll}
 & (\Delta_C, \Delta) \in T_{P_{UP}} \uparrow \beta\,(q(\vec{t})) \\
\underset{(4.75)}{\iff} & (\Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P_{UP}} \uparrow \alpha\,(q(\vec{t})) \\
\iff & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C, \Delta) \in T_{P_{UP}} \uparrow \alpha\,(q(\vec{t})) \\
\underset{(IH)}{\iff} & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C, \Delta) \in T_{P'_{UP}} \uparrow \alpha\,(q(\vec{t})) \\
\iff & (\Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P'_{UP}} \uparrow \alpha\,(q(\vec{t})) \\
\underset{(4.75)}{\iff} & (\Delta_C, \Delta) \in T_{P'_{UP}} \uparrow \beta\,(q(\vec{t}))
\end{array}
$$

Next, let us choose an ordinal α that is greater than both closure ordinals $\alpha_{P_{UP}}$ and $\alpha_{P'_{UP}}$ (cf. Theorem 4.87). The equality of the minimal models modulo p follows directly from the inequalities

$$T_{P_{UP}} \uparrow \alpha \leq M_{UP}[P_{UP}, s_0] = T_{P_{UP}} \uparrow \alpha_{P_{UP}} \leq T_{P_{UP}} \uparrow \alpha$$

and

$$T_{P'_{UP}} \uparrow \alpha \leq M_{UP}[P'_{UP}, s_0] = T_{P'_{UP}} \uparrow \alpha_{P'_{UP}} \leq T_{P'_{UP}} \uparrow \alpha$$

which can be derived from Theorem 4.87 using fixpoint properties (see [Llo87] for details). □

The following proposition provides a statement about the interpretation of an auxiliary predicate. Informally speaking, the auxiliary predicate serves as an abbreviation of an update formula.

Proposition 6.8 Let P_{UP} be an update program, and let $p \in Pred_{DU}$ be a definable update predicate which does not occur in the rules of P_{UP}. Let $p(\vec{s})$ be an arbitrary definable update atom over p and χ be an arbitrary update goal. Define P'_{UP} as the extension of P_{UP} by the new rule $p(\vec{s}) \leftarrow \chi$.

Then for arbitrary ground instances of χ and $p(\vec{s})$ the following holds:

$$M_{UP}[P'_{UP}, s_0](\chi[\vec{Y} / \vec{t}]) = M_{UP}[P'_{UP}, s_0](p(\vec{s})[\vec{Y} / \vec{t}])$$

Proof: Define $M := M_{UP}[P'_{UP}, s_0]$.

'$\subseteq$':

By Lemma 4.72, an interpretation I is a model of a program P, iff for every ground instance $U \rightarrow q(\vec{t})$ of a rule $r \in P$ the set inclusion $I(U) \subseteq I(q(\vec{t}))$ holds.

As M is a model of P'_{UP},

$$M(\chi[\vec{Y} / \vec{t}]) \subseteq M(p(\vec{s})[\vec{Y} / \vec{t}])$$

holds for every ground instance of the auxiliary rule.

'$\supseteq$':

Recall that the auxiliary rule is the only rule that defines the predicate p. By Theorem 4.87, M is the least fixpoint of $T_{P'_{UP}}$. So, if $(\Delta_C, \Delta) \in M(p(\vec{s})[\vec{Y} / \vec{t}])$ and thus $(\Delta_C, \Delta) \in T_{P'_{UP}}(M)(p(\vec{s})[\vec{Y} / \vec{t}])$ holds, then $(\Delta_C, \Delta) \in M(\chi[\vec{Y} / \vec{t}])$ must also hold by Definition 4.83.

□

Next, we will show that under certain conditions it is possible to rewrite subgoals occurring in a rule of an update program. Before we can present the main theorem, we need the following lemma.

Lemma 6.9 Let $I \in \mathcal{I}$ be an interpretation of update formulas. Let χ_1 and χ_2 be update goals with the same free variables $\vec{X}$ such that for all ground instances

$$I(\chi_1[\vec{X} / \vec{t}]) \subseteq I(\chi_2[\vec{X} / \vec{t}])$$

holds. Let φ_1 be an update goal containing χ_1 as a subgoal at some position, and let φ_2 be a structurally identical goal, where one occurrence of χ_1 has been replaced by χ_2. Let $\vec{Y}$ denote the free variables of φ_1, which are also the free variables of φ_2. Then for all ground instances

$$I(\varphi_1[\vec{Y} / \vec{t}]) \subseteq I(\varphi_2[\vec{Y} / \vec{t}])$$

holds.

Proof: We show the assertion by structural induction on the goals φ_1 and φ_2. Outside the replacement area, both formulas are structurally equivalent, and corresponding subgoals have the same free variables. In the following we mark subgoals of φ_1 with the index 1 and subgoals of φ_2 with the index 2.

Base cases (outside the replacement area):

These cases are trivial, since the subgoals of φ_1 and φ_2 coincide outside the replacement area.

Base case (replacement goals):

The subgoals χ_1 (of φ_1) and χ_2 (of φ_2) can be treated as a further base case. The desired inclusions hold by the precondition.

Induction step:

By the induction hypothesis,

$$I(\varphi_1[\vec{Y} \,/\, \vec{t}]) \subseteq I(\varphi_2[\vec{Y} \,/\, \vec{t}])$$

holds for any direct proper subgoal φ_1 and the corresponding subgoal φ_2 of the composite goals analyzed in the following. In each case shown below, we choose arbitrary ground instances of the composite goals and arbitrary consistent transitions $\Delta_C, \Delta \in \mathcal{T}_{Cons}$, then we apply Definition 4.9.

1. Concurrent conjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\varphi_1, \psi_1)[\vec{Y} \,/\, \vec{t}]) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I(\varphi_1[\vec{Y} \,/\, \vec{t}]) \text{ and } (\Delta_C, \Delta_2) \in I(\psi_1[\vec{Y} \,/\, \vec{t}]) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I(\varphi_2[\vec{Y} \,/\, \vec{t}]) \text{ and } (\Delta_C, \Delta_2) \in I(\psi_2[\vec{Y} \,/\, \vec{t}]) \\ & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\varphi_2, \psi_2)[\vec{Y} \,/\, \vec{t}]) \end{array}$$

2. Sequential conjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\varphi_1 : \psi_1)[\vec{Y} \,/\, \vec{t}]) \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ & (\Delta_C, \Delta_1) \in I(\varphi_1[\vec{Y} \,/\, \vec{t}]) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi_1[\vec{Y} \,/\, \vec{t}]) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \Longrightarrow & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta_1) \in I(\varphi_2[\vec{Y} \,/\, \vec{t}]) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi_2[\vec{Y} \,/\, \vec{t}]) \\ & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\varphi_2 : \psi_2)[\vec{Y} \,/\, \vec{t}]) \end{array}$$

3. Disjunction

$$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\varphi_1 \vee \psi_1)[\vec{Y} \,/\, \vec{t}]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I(\varphi_1[\vec{Y} \,/\, \vec{t}]) \text{ or } (\Delta_C, \Delta) \in I(\psi_1[\vec{Y} \,/\, \vec{t}]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I(\varphi_2[\vec{Y} \,/\, \vec{t}]) \text{ or } (\Delta_C, \Delta) \in I(\psi_2[\vec{Y} \,/\, \vec{t}]) \\ {\scriptstyle (IH)} & \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\varphi_2 \vee \psi_2)[\vec{Y} \,/\, \vec{t}]) \end{array}$$

4. Existential quantification

   Consider a quantification $\exists\, \vec{Z}\, \varphi_1$, where $\vec{Z} = Z_1, \ldots, Z_n$. Let $\vec{Y}'$ denote the free variables of the subgoal φ_1. By the induction hypothesis,

   $$I(\varphi_1[\vec{Y}' / \vec{t}]) \subseteq I(\varphi_2[\vec{Y}' / \vec{t}])$$

   holds for all ground instances. The variables $\vec{Z}$ do not occur in the sequence $\vec{Y}$ of the free variables of the existential quantification. We can reason as follows:

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\exists\, \vec{Z}\, \varphi_1)[\vec{Y} / \vec{t}]) \\ \Longrightarrow & \text{there exists a ground term tuple } (\vec{s}) \in \mathcal{U}^n \text{ such that :} \\ & (\Delta_C, \Delta) \in I(\varphi_1[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{s}]) \\ \Longrightarrow & \text{there exists a ground term tuple } (\vec{s}) \in \mathcal{U}^n \text{ such that :} \\ {\scriptstyle (IH)} & (\Delta_C, \Delta) \in I(\varphi_2[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{s}]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\exists\, \vec{Z}\, \varphi_2)[\vec{Y} / \vec{t}]) \end{array}$$

5. Bulk quantification

   Consider a bulk quantification $\#\, \vec{Z}\, [A \mapsto \varphi_1]$, where $\vec{Z} = Z_1, \ldots, Z_n$. Let $\vec{Y}'$ denote the free variables of the subgoal φ_1. By the induction hypothesis,

   $$I(\varphi_1[\vec{Y}' / \vec{t}]) \subseteq I(\varphi_2[\vec{Y}' / \vec{t}])$$

   holds for all ground instances. The variables $\vec{Z}$ do not occur in the sequence $\vec{Y}$ of the free variables of the bulk quantification.

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

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

   does not depend on the interpretation of the update subgoal.

   The case $T_{A[\vec{Y} / \vec{t}], \Delta_C} = \emptyset$ is trivial:

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\#\, \vec{Z}\, [A \mapsto \varphi_1])[\vec{Y} / \vec{t}]) \\ \Longrightarrow & \Delta = Log(A[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{all}\,]) \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\#\, \vec{Z}\, [A \mapsto \varphi_2])[\vec{Y} / \vec{t}]) \end{array}$$

   For $T_{A[\vec{Y} / \vec{t}], \Delta_C} \neq \emptyset$ we get:

   $$\begin{array}{ll} & (\Delta_C, \Delta) \in I((\#\, \vec{Z}\, [A \mapsto \varphi_1])[\vec{Y} / \vec{t}]) \\ \Longrightarrow & \text{there exists a function } f : T_{A[\vec{Y} / \vec{t}], \Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall (\vec{s}) \in T_{A[\vec{Y} / \vec{t}], \Delta_C} : \ (\Delta_C, f(\vec{s})) \in I(\varphi_1[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{s}]) \\ & \text{and } \Delta = Log(A[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{all}\,]) \ \oplus \ \bigsqcup_{(\vec{s}) \in T_{A[\vec{Y} / \vec{t}], \Delta_C}} f(\vec{s}) \\ \Longrightarrow & \text{there exists a function } f : T_{A[\vec{Y} / \vec{t}], \Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ {\scriptstyle (IH)} & \forall (\vec{s}) \in T_{A[\vec{Y} / \vec{t}], \Delta_C} : \ (\Delta_C, f(\vec{s})) \in I(\varphi_2[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{s}]) \\ & \text{and } \Delta = Log(A[\vec{Y}\,,\, \vec{Z} \,/\, \vec{t}\,,\, \vec{all}\,]) \ \oplus \ \bigsqcup_{(\vec{s}) \in T_{A[\vec{Y} / \vec{t}], \Delta_C}} f(\vec{s}) \\ \Longrightarrow & (\Delta_C, \Delta) \in I((\#\, \vec{Z}\, [A \mapsto \varphi_2])[\vec{Y} / \vec{t}]) \end{array}$$

□

Now we are able to present the main result about program rewriting. The relevant part of the minimal model of an update program is kept unchanged, if an arbitrarily chosen subgoal in any rule body is replaced by an unused definable update atom and a corresponding auxiliary rule is

added to the program. To make the theorem more general, we allow the (successive) rewriting of multiple occurrences of the selected subgoal.

Theorem 6.10 [Rewriting of Update Programs] Let P_0 be an update program, let $p \in Pred_{DU}$ be a definable update predicate which does not occur in the rules of P_0, and let χ be an update goal with the free variables $\vec{X}$. Let $P_1, \ldots, P_n$ be update programs such that P_i is a rewritten version of P_{i-1} where one occurrence of χ in a rule body has been replaced by $p(\vec{X})$ for $i \in \{1, \ldots, n\}$. Define P_i' as the extension of P_i by the new rule $p(\vec{X}) \leftarrow \chi$ for each $i \in \{0, \ldots, n\}$. Then the following holds:

$$M_{UP}[P_0, s_0] =_p M_{UP}[P_n', s_0]$$

Proof: By Lemma 6.7,

$$M_{UP}[P_0, s_0] =_p M_{UP}[P_0', s_0]$$

holds. We show that

$$M_{UP}[P_{i-1}', s_0] = M_{UP}[P_i', s_0]$$

holds for all $i \in \{1, \ldots, n\}$. The assertion then follows directly by induction. Note that P_{i-1}' and P_i' differ exactly in one rule r_0 and this is not the auxiliary rule $p(\vec{X}) \leftarrow \chi$.

'$\leq$':

We show that every model of P_i' is also a model of P_{i-1}'. Then $M_{UP}[P_i', s_0]$ is a model of P_{i-1}', and

$$M_{UP}[P_{i-1}', s_0] \leq M_{UP}[P_i', s_0]$$

follows from the construction of $M_{UP}[P_{i-1}', s_0]$ as the greatest lower bound of all models of P_{i-1}' (cf. Theorem 4.81).

By Lemma 4.72, an interpretation I is a model of a program P_{UP}, iff for every ground instance $U \rightarrow q(\vec{t})$ of a rule $r \in P_{UP}$ the set inclusion $I(U) \subseteq I(q(\vec{t}))$ holds.

So, let $I \in \mathcal{I}$ be a model of P_i', i.e. for every ground instance $U \rightarrow q(\vec{t})$ of a rule $r \in P_i'$,

$$I(U) \subseteq I(q(\vec{t}))$$

holds. The same inclusion holds for the unchanged rules in P_{i-1}'. Consequently, we just have to prove a similar inclusion for the rule r_0, which has been rewritten in P_i'.

Let us consider a ground instance $U \rightarrow q(\vec{t})$ of r_0 and the corresponding instance $U' \rightarrow q(\vec{t})$ of the rewritten rule in P_i'.

Note that in particular

$$I(\chi[\vec{X} / \vec{t}]) \subseteq I(p(\vec{t}))$$

holds for all ground instances of the auxiliary rule contained in P'_{i-1} and P'_i due to the model property of I. By Lemma 6.9,

$$I(U) \subseteq I(U')$$

and thus by the model property of I

$$I(U) \subseteq I(q(\vec{t}))$$

follows. We can conclude that I is also a model of P'_{i-1}.

'$\geq$':

First, we show that for every interpretation $I \in \mathcal{I}$ such that $I \leq T_{P'_{i-1}}(I)$,

$$I(p(\vec{t})) \subseteq I(\chi[\vec{X} / \vec{t}])$$

holds for all ground instances of the free variables $\vec{X}$ of the fixed goals. The property

$$T_{P'_i}(I) \leq T_{P'_{i-1}}(I)$$

can be derived easily. In the second step we can use transfinite induction to prove the main assertion.

So, assume that $I \in \mathcal{I}$ is an interpretation for which the precondition $I \leq T_{P'_{i-1}}(I)$ holds. Consider an arbitrary instance $\chi[\vec{X} / \vec{t}] \rightarrow p(\vec{t})$ of the auxiliary rule, which is the only rule that defines p. So, if $(\Delta_C, \Delta) \in I(p(\vec{t}))$ and thus $(\Delta_C, \Delta) \in T_{P'_{i-1}}(I)(p(\vec{t}))$ holds, then $(\Delta_C, \Delta) \in I(\chi[\vec{X} / \vec{t}])$ must also hold by Definition 4.83. This concludes the proof of the first assertion.

Next, we have to show that

$$T_{P'_i}(I)(q(\vec{t})) \subseteq T_{P'_{i-1}}(I)(q(\vec{t}))$$

holds for all ground definable update atoms $q(\vec{t}) \in \mathcal{B}_{DU}$.

Let $(\Delta_C, \Delta) \in T_{P'_i}(I)(q(\vec{t}))$. If this property is derived according to Definition 4.83 by a different rule than that one corresponding to r_0, then $(\Delta_C, \Delta) \in T_{P'_{i-1}}(I)(q(\vec{t}))$ holds, too. In the other case, there is an instance $U' \rightarrow q(\vec{t})$ of the transformed rule r_0 such that $(\Delta_C, \Delta) \in I(U')$. Let U be the corresponding instance of the rule body of the original rule r_0. We have to show that $(\Delta_C, \Delta) \in I(U)$ holds, too, in order to finish the proof of the second assertion. Recall that in the body of r_0 one occurrence of the subgoal χ has been rewritten by $p(\vec{X})$. Further, U and U' are corresponding ground instances of the rule bodies. From the assertion proved above

$$I(U') \subseteq I(U)$$

follows by Lemma 6.9. Thus $(\Delta_C, \Delta) \in I(U)$ holds.

Finally, the main assertion

$$M_{UP}[P'_{i-1}, s_0] \geq M_{UP}[P'_i, s_0]$$

will be proved by transfinite induction. For a given ordinal γ we show that the following condition holds:

$$T_{P'_i} \uparrow \gamma \leq T_{P'_{i-1}} \uparrow \gamma$$

Base case: $\gamma = 0$

The case is trivial.

Induction step: successor ordinal $\alpha + 1$

By the induction hypothesis, the desired condition holds w.r.t. $T_{P'_i} \uparrow \alpha$ and $T_{P'_{i-1}} \uparrow \alpha$. Hence, the inequalities

$$T_{P'_i}(T_{P'_i} \uparrow \alpha) \leq T_{P'_i}(T_{P'_{i-1}} \uparrow \alpha)$$

and

$$T_{P'_{i-1}} \uparrow \alpha \leq T_{P'_{i-1}}(T_{P'_{i-1}} \uparrow \alpha)$$

hold due to the monotonicity of $T_{P'_{i-1}}$ and $T_{P'_i}$ (see Lemma 4.84). By the assertion proved above with $I := T_{P'_{i-1}} \uparrow \alpha$,

$$T_{P'_i}(T_{P'_{i-1}} \uparrow \alpha) \leq T_{P'_{i-1}}(T_{P'_{i-1}} \uparrow \alpha)$$

follows from the second inequality. Now we can reason as follows:

$$\begin{array}{ll} & T_{P'_i} \uparrow \alpha + 1 \\ \underset{(4.86)}{=} & T_{P'_i}(T_{P'_i} \uparrow \alpha) \\ \underset{(see\ above)}{\leq} & T_{P'_i}(T_{P'_{i-1}} \uparrow \alpha) \\ \underset{(see\ above)}{\leq} & T_{P'_{i-1}}(T_{P'_{i-1}} \uparrow \alpha) \\ \underset{(4.86)}{=} & T_{P'_{i-1}} \uparrow \alpha + 1 \end{array}$$

Induction step: limit ordinal β

By the induction hypothesis, the desired condition holds w.r.t. all ordinals $\alpha < \beta$.

Now choose a ground definable update atom $q(\vec{t}) \in \mathcal{B}_{DU}$. For arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$, the following implications hold.

$$\begin{array}{ll} & (\Delta_C, \Delta) \in T_{P'_i} \uparrow \beta\,(q(\vec{t})) \\ \underset{(4.75)}{\Longrightarrow} & (\Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P'_i} \uparrow \alpha\,(q(\vec{t})) \\ \Longrightarrow & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C, \Delta) \in T_{P'_i} \uparrow \alpha\,(q(\vec{t})) \\ \Longrightarrow & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C, \Delta) \in T_{P'_{i-1}} \uparrow \alpha\,(q(\vec{t})) \\ \underset{(IH)}{\Longrightarrow} & (\Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T_{P'_{i-1}} \uparrow \alpha\,(q(\vec{t})) \\ \Longrightarrow & (\Delta_C, \Delta) \in T_{P'_{i-1}} \uparrow \beta\,(q(\vec{t})) \\ \underset{(4.75)}{\Longrightarrow} & \end{array}$$

Let us choose an ordinal α that is greater than both closure ordinals $\alpha_{P'_{i-1}}$ and $\alpha_{P'_i}$ (cf. Theorem 4.87). The assertion about the minimal models follows directly from the inequalities

$$T_{P'_{i-1}} \uparrow \alpha \leq M_{UP}[P'_{i-1}, s_0] = T_{P'_{i-1}} \uparrow \alpha_{P'_{i-1}} \leq T_{P'_{i-1}} \uparrow \alpha$$

and

$$T_{P'_i} \uparrow \alpha \leq M_{UP}[P'_i, s_0] = T_{P'_i} \uparrow \alpha_{P'_i} \leq T_{P'_i} \uparrow \alpha$$

which can be derived from Theorem 4.87 using fixpoint properties (see [Llo87] for details).

□

Remark 6.11 Theorem 6.10 formalizes an equality between minimal models. The program rewriting can be iterated and can also be applied in the reversed order. □

Example 6.12 [Rewriting of Update Programs] Recall the setting of Example 3.20. Let P_0 be the update program

$$\begin{array}{lcl} act(X) & \leftarrow & [\ xmove(X)\ :\ pickup\ ]\ :\ [\ xmove(0)\ :\ putdown\ ] \\ act(X) & \leftarrow & [\ xmove(X)\ :\ pickup\ ]\ :\ [\ putdown\ :\ xmove(0)\ ] \end{array}$$

and p be a new predicate that does not occur in P_0. Next, we define χ as the subgoal

$$\chi :\equiv xmove(X) : pickup$$

which occurs in both rules of P_0. Note that the variable X has to be respected in the subsequent rewriting process, as it occurs free in χ.

Now we rewrite the occurrence of χ in the first rule by $p(X)$, do the same with the second rule, and add the definition $p(X) \leftarrow \chi$. This way, we have constructed a program P'_2 that reads as follows:

$$\begin{array}{lcl} act(X) & \leftarrow & p(X)\ :\ [\ xmove(0)\ :\ putdown\ ] \\ act(X) & \leftarrow & p(X)\ :\ [\ putdown\ :\ xmove(0)\ ] \\ p(X) & \leftarrow & xmove(X)\ :\ pickup \end{array}$$

By Theorem 6.10, P_0 and P'_2 have the same minimal model (modulo p). □

6.3.2 Normal Forms of Update Programs

Theorem 6.10 can be applied successively in order to transform a finite set of update rules into a normal form where nested rule bodies are not allowed.

Definition 6.13 An update rule is called *normalized*, if its rule body is either an update literal or a complex goal whose direct subgoals are update literals.

An update program P_{UP} is called *normalized*, if all rules $r \in P_{UP}$ are normalized.

An update program is called *finite*, if it is a finite set of update rules. □

Theorem 6.14 [Existence of Normal Forms] Let P_{UP} be a finite update program, and let the set of definable update predicates $Pred_{DU}$ be infinite. Define $Pred_{DU}|_{P_{UP}}$ as the (finite) subset of update predicates that occur in P_{UP} and $\mathcal{B}_{DU}|_{P_{UP}}$ as the restriction of the definable update base $\mathcal{B}_{DU}$ to the predicates of $Pred_{DU}|_{P_{UP}}$.

Then there exists a normalized and finite update program P'_{UP} such that

$$M_{UP}[P_{UP}, s_0](p(\vec{t})) = M_{UP}[P'_{UP}, s_0](p(\vec{t}))$$

holds for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}|_{P_{UP}}$.

Proof: It is possible to construct a sequence $P_0, \ldots, P_n$ of finite update programs with $P_0 = P_{UP}$ and $P_n = P'_{UP}$ such that for each $i \in \{1, \ldots, n\}$,

$$M_{UP}[P_{i-1}, s_0](p(\vec{t})) = M_{UP}[P_i, s_0](p(\vec{t}))$$

holds for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}|_{P_{UP}}$.

In every step, take a rule r_0 that is not normalized and replace one of the innermost complex subgoals of the rule body by the method of Theorem 6.10 which keeps the minimal model semantics modulo p. p is not contained in $Pred_{DU}|_{P_{UP}}$ and thus the equality above holds. The auxiliary rule will be normalized, and the complexity of the remaining rules will strictly decrease. Thus, termination of the rewriting process is guaranteed. Note that the infinity of $Pred_{DU}$ is essential to ensure that it is always possible to find an unused auxiliary predicate. □

Theorem 6.14 may be advantageous, when an operational semantics for the ULTRA concept has to be developed and implemented. The evaluation methods can be designed to work on normalized update programs, which are generated by a preprocessor at compile-time. The representation and manipulation of normalized update programs will simplify the run-time system and could even lead to a greater efficiency. Additionally, the theoretical investigations about the operational semantics, e.g. correctness and completeness proofs, could be restricted to normalized update programs, too.

Example 6.15 [Normal Form] The update program

$$act \leftarrow [\,[xstep(1) : pickup] \;:\; [xstep(-1) : putdown]\,] \;\vee\; [\,pickup : putdown\,]$$

taken from the robot domain can be rewritten into the normal form

$$\begin{aligned}
act &\leftarrow p_3 \vee p_4 \\
p_4 &\leftarrow pickup \;:\; putdown \\
p_3 &\leftarrow p_1 \;:\; p_2 \\
p_2 &\leftarrow xstep(-1) \;:\; putdown \\
p_1 &\leftarrow xstep(1) \;:\; pickup
\end{aligned}$$

using the method sketched in the proof of Theorem 6.14. The predicates $p_1, \ldots, p_4 \in Pred_{DU}$ are auxiliary predicates which must not occur in the original program. □

6.3.3 Instantiated Rules

Update programs can be rewritten by replacing non-ground rules by their (possibly ground) instances. The number of implicitly quantified variables is reduced, possibly up to zero. In this latter case, all remaining variables will occur in the rule bodies and always in the scope of an explicit quantifier ($\exists$ or $\#$). In practice, the instantiation of rules is only viable, if the Herbrand universe $\mathcal{U}$ is finite. Otherwise infinite update programs would be produced.

Proposition 6.16 [Instantiation of Rules] Let P_{UP} an update program, and let P'_{UP} be constructed from P_{UP} by instantiation of some subset $R \subseteq P_{UP}$ of rules as follows: if $r \in R$, then r is replaced by the set of all ground instances $r[\vec{X} / \vec{t}]$ with $(\vec{t}) \in \mathcal{U}^n$ where $X_1, \ldots, X_n$ is some finite sequence of variables. Then

$$M_{UP}[P_{UP}, s_0] = M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: Using Definition 4.83, it is easy to show that

$$T_{P_{UP}}(I) = T_{P'_{UP}}(I)$$

holds for all interpretations $I \in \mathcal{I}$. The assertion about the minimal models follows immediately. □

It is also legitimate to add redundant rules, i.e. rules that are instances of existing rules.

Proposition 6.17 [Adding Redundant Rules] Let P_{UP} an update program, and let P'_{UP} be constructed from P_{UP} by adding rules such that every (new) rule in P'_{UP}, is an instance of a rule in P_{UP}. Then

$$M_{UP}[P_{UP}, s_0] = M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: The proof is analogous to the proof of Proposition 6.16. □

Remark 6.18 Propositions 6.16 and 6.17 formalize equalities between minimal models. The program rewritings can be iterated and can also be applied in the reversed order. In particular, Proposition 6.17 allows also the elimination of a rule that is an instance of another rule in the same program. □

6.3.4 Elimination of Disjunction and Existential Quantification

In this section we show that it is possible to eliminate disjunctions and existential quantifications occurring in rule bodies. This could be exploited together with the results of Section 6.3.2 to facilitate the development of an operational semantics.

Proposition 6.19 [Elimination of Disjunctions] Let P_{UP} an update program, and let P'_{UP} be constructed from P_{UP} by replacing some rules of the form $p(\vec{t}) \leftarrow \varphi \vee \psi$ by the corresponding pairs of rules $p(\vec{t}) \leftarrow \varphi$ and $p(\vec{t}) \leftarrow \psi$. Then

$$M_{UP}[P_{UP}, s_0] = M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: Using Definitions 4.83 and 4.9, it is easy to show that

$$T_{P_{UP}}(I) = T_{P'_{UP}}(I)$$

holds for all interpretations $I \in \mathcal{I}$. The assertion about the minimal models follows immediately. □

Example 6.20 [Elimination of Disjunctions] Let us assume that we have to implement a new operation *act* for the robot world of Example 3.20: if the robot is empty it should try to pick up a block, otherwise it should try to lay down the grabbed block onto the floor. The desired operation can be specified in a natural way using the disjunction $\vee$ in the rule body:

$$act \leftarrow [\ empty\ :\ pickup\ ] \vee [\ NOT\ empty\ :\ putdown\ ]$$

However, Proposition 6.19 allows us to replace the single rule by the two rules

$$act \leftarrow empty\ :\ pickup$$
$$act \leftarrow NOT\ empty\ :\ putdown$$

without changing the semantics. □

Proposition 6.21 [Elimination of Existential Quantifications] Let P_{UP} an update program, and let P'_{UP} be constructed from P_{UP} by replacing some rules of the form $p(\vec{t}) \leftarrow \exists \vec{X}\, \varphi$, where the variables of $p(\vec{t})$ do not occur in $\vec{X}$, by the corresponding rules $p(\vec{t}) \leftarrow \varphi$. Then

$$M_{UP}[P_{UP}, s_0] = M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: The proof is analogous to the proof of Proposition 6.19. □

Example 6.22 [Elimination of Existential Quantifications] Recall our introductory example modeling a storage for transport devices (see Appendix A for details). Let us extend the set of operations by the new operation *eliminate* that completely removes a transport item I from the *store* table. The operation should find values for P and A such that $store(I, P, A)$ holds, and it should request the deletion of the corresponding tuple. The logically correct rule reads as follows:

$$eliminate(I) \leftarrow \exists P, A\ [\ store(I, P, A),\ DEL\ store(I, P, A)\ ]$$

However, it can be simplified to

$$eliminate(I) \leftarrow store(I, P, A),\ DEL\ store(I, P, A)$$

according to Proposition 6.21. Variables that occur locally in a rule body but not in the scope of an explicit quantifier are implicitly existentially quantified. □

Remark 6.23 Propositions 6.19 and 6.21 formalize equalities between minimal models. The program rewritings can be iterated and can also be applied in the reversed order. Note that a finite program is always transformed into a finite program. □

Corollary 6.24 Let P_{UP} be a normalized update program. Then there exists a normalized update program P'_{UP} such that no disjunctions and no existential quantifications occur in the rule bodies of P'_{UP} and

$$M_{UP}[P_{UP}, s_0] = M_{UP}[P'_{UP}, s_0]$$

holds.

Proof: As P_{UP} is normalized, its rules do not contain any nested complex goals.

Apply Proposition 6.19 to eliminate the disjunctive rule bodies. The modified rules are normalized.

Next, if there are rules having (implicitly quantified) variables in the rule head that also occur bound by an existential quantifier in the rule body, rename the concerned variables in the head.

Finally, apply Proposition 6.21 to eliminate the existentially quantified rule bodies. The modified rules are normalized. □

Theorem 6.14 and Corollary 6.24 together state that finite update programs can be rewritten into a normal form without disjunction and existential quantification. The minimal model semantics is kept w.r.t. the predicates that occur in the original program.

Example 6.25 [Local Variables and Existential Quantifications] Recall our introductory example (see Appendix A and also Example 6.22) and consider the following rule, which specifies an elimination of all transport devices having a low stock:

$$\begin{array}{lll} eliminate\_low & \leftarrow & \#\ I \\ & & [\ low(I)\ \mapsto \\ & & \quad \exists P, A\ [\ store(I,P,A),\ DEL\ store(I,P,A)\ ]\ ] \end{array}$$

The variables P and A must be existentially quantified, as they have to be instantiated *individually* for each I. The quantifier cannot simply be dropped. However, it is possible to produce the normal form

$$\begin{array}{lll} eliminate\_low & \leftarrow & \#\ I\ [\ low(I)\ \mapsto\ p_2(I)\ ] \\ p_2(I) & \leftarrow & \exists P, A\ \ p_1(I,P,A) \\ p_1(I,P,A) & \leftarrow & store(I,P,A),\ DEL\ store(I,P,A) \end{array}$$

with new auxiliary predicates p_1 and p_2. Now we can omit the existential quantifier. This leads to the following normalized update program:

$$\begin{array}{lll} eliminate\_low & \leftarrow & \#\ I\ [\ low(I)\ \mapsto\ p_2(I)\ ] \\ p_2(I) & \leftarrow & p_1(I,P,A) \\ p_1(I,P,A) & \leftarrow & store(I,P,A),\ DEL\ store(I,P,A) \end{array}$$

□

6.4 Semantics of Programs in Different Initial States

The model-theoretic semantics of an update program has been defined w.r.t. an arbitrary but fixed initial state (always called s_0 in Section 4). In this section we formalize relations between interpretations that refer to *different* initial states. The assertions may be helpful when formalizing an operational semantics that is based on immediate updates. Note that the immediate updates will change the physical state during the evaluation of composite operations and thus do not harmonize directly with the logical ULTRA semantics.

To be able to deal with multiple initial states, we have to use some modified notation that takes the initial state as an explicit parameter. Let an interpretation $I \in \mathcal{I}$ of the definable update atoms be given, and let $s \in \mathcal{S}$ be a state. We denote the interpretation of arbitrary update formulas as defined by Definition 4.9 w.r.t. the initial state s by I^s. Further, for a given update program P_{UP} we denote the immediate consequence operator w.r.t. the initial state s by $T^s_{P_{UP}}$. According to Definition 4.83, $T^s_{P_{UP}} : \mathcal{I} \to \mathcal{I}$ is formally defined by

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

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

Note that in general for different states $s_1, s_2 \in \mathcal{S}$, I^{s_1} and I^{s_2} do not coincide, and thus $T^{s_1}_{P_{UP}}(I)$ may differ from $T^{s_2}_{P_{UP}}(I)$.

Recall from Definition 4.94 that $M_{UP}[P_{UP}, s]$ implicitly means $M_{UP}[P_{UP}, s]^s$.

Now we are going to prove a relation between the interpretation of update formulas w.r.t. different initial states. This will lead to an essential property of the minimal models of update programs.

Lemma 6.26 Let $s_0, s_1 \in \mathcal{S}$ be arbitrary states and $\Delta_{0-1} \in \mathcal{T}_{Cons}$ be a consistent transition such that $s_1 = s_0 \oplus_E \Delta_{0-1}$. Let $I_0, I_1 \in \mathcal{I}$ be interpretations of the definable update atoms such that for arbitrary ground atoms $p(\vec{t}) \in \mathcal{B}_{DU}$

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} : \; (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0(p(\vec{t})) \Longleftrightarrow (\Delta_C, \Delta) \in I_1(p(\vec{t}))$$

holds. Then for arbitrary ground update formulas φ

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} : \; (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi) \Longleftrightarrow (\Delta_C, \Delta) \in I_1^{s_1}(\varphi)$$

holds.

Proof: We prove the assertion by structural induction. In each case shown below, we choose arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$ and show the desired equivalence using Definition 4.9.

Base cases:

1. DB literal

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

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s0}(A) \\
\iff & I_{DB}(s_0 \oplus_E (\Delta_{0-1} \oplus \Delta_C)) \models A \text{ and } \Delta = Log(A) \\
\iff & I_{DB}((s_0 \oplus_E \Delta_{0-1}) \oplus_E \Delta_C) \models A \text{ and } \Delta = Log(A) \\
\underset{(4.2)}{\iff} & I_{DB}(s_1 \oplus_E \Delta_C) \models A \text{ and } \Delta = Log(A) \\
\iff & (\Delta_C, \Delta) \in I_1^{s1}(A)
\end{array}
$$

2. NOP literal

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s0}(NOP) \\
\iff & \Delta = \Delta_\varepsilon \\
\iff & (\Delta_C, \Delta) \in I_1^{s1}(NOP)
\end{array}
$$

3. Basic update atom

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s0}(u(\vec{t})) \\
\iff & \Delta = Upd(u(\vec{t})) \\
\iff & (\Delta_C, \Delta) \in I_1^{s1}(u(\vec{t}))
\end{array}
$$

4. Definable update atom

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s0}(p(\vec{t})) \\
\iff & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0(p(\vec{t})) \\
\iff & (\Delta_C, \Delta) \in I_1(p(\vec{t})) \\
\iff & (\Delta_C, \Delta) \in I_1^{s1}(p(\vec{t}))
\end{array}
$$

   Note that the precondition is essential for this part of the proof.

Induction step:

By the induction hypothesis, the assertion holds for any direct proper subformula φ of the composite formulas analyzed in the following.

1. Concurrent conjunction

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s0}(\varphi, \psi) \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_{0-1} \oplus \Delta_C, \Delta_1) \in I_0^{s0}(\varphi) \text{ and } (\Delta_{0-1} \oplus \Delta_C, \Delta_2) \in I_0^{s0}(\psi) \\
 & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
\underset{(IH)}{} & (\Delta_C, \Delta_1) \in I_1^{s1}(\varphi) \text{ and } (\Delta_C, \Delta_2) \in I_1^{s1}(\psi) \\
 & \text{and } \Delta = \Delta_1 \sqcup \Delta_2 \\
\iff & (\Delta_C, \Delta) \in I_1^{s1}(\varphi, \psi)
\end{array}
$$

2. Sequential conjunction

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi : \psi) \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
 & (\Delta_{0-1} \oplus \Delta_C, \Delta_1) \in I_0^{s_0}(\varphi) \text{ and } ((\Delta_{0-1} \oplus \Delta_C) \oplus \Delta_1, \Delta_2) \in I_0^{s_0}(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
\scriptstyle (4.2) & (\Delta_{0-1} \oplus \Delta_C, \Delta_1) \in I_0^{s_0}(\varphi) \text{ and } (\Delta_{0-1} \oplus (\Delta_C \oplus \Delta_1), \Delta_2) \in I_0^{s_0}(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & \text{there exist } \Delta_1, \Delta_2 \in \mathcal{T}_{Cons} \text{ such that :} \\
\scriptstyle (IH) & (\Delta_C, \Delta_1) \in I_1^{s_1}(\varphi) \text{ and } (\Delta_C \oplus \Delta_1, \Delta_2) \in I_1^{s_1}(\psi) \\
 & \text{and } \Delta = \Delta_1 \oplus \Delta_2 \\
\iff & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi : \psi)
\end{array}
$$

3. Disjunction

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi \vee \psi) \\
\iff & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi) \text{ or } (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\psi) \\
\iff & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi) \text{ or } (\Delta_C, \Delta) \in I_1^{s_1}(\psi) \\
\scriptstyle (IH) & \\
\iff & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi \vee \psi)
\end{array}
$$

4. Existential quantification

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

$$
\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\exists\, \vec{X}\, \varphi) \\
\iff & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi[\, \vec{X} \,/\, \vec{t}\,]) \\
\iff & \text{there exists a ground term tuple } (\vec{t}\,) \in \mathcal{U}^n \text{ such that :} \\
\scriptstyle (IH) & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi[\, \vec{X} \,/\, \vec{t}\,]) \\
\iff & (\Delta_C, \Delta) \in I_1^{s_1}(\exists\, \vec{X}\, \varphi)
\end{array}
$$

5. Bulk quantification

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

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} : (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi[\, \vec{X} \,/\, \vec{t}\,]) \iff (\Delta_C, \Delta) \in I_1^{s_1}(\varphi[\, \vec{X} \,/\, \vec{t}\,])$$

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

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

$$T^{s_0}_{A, \Delta_{0-1} \oplus \Delta_C} := \{(\vec{t}\,) \in \mathcal{U}^n \mid I_{DB}(s_0 \oplus_E (\Delta_{0-1} \oplus \Delta_C)) \models A[\, \vec{X} \,/\, \vec{t}\,]\}$$

   and

$$T^{s_1}_{A, \Delta_C} := \{(\vec{t}\,) \in \mathcal{U}^n \mid I_{DB}(s_1 \oplus_E \Delta_C) \models A[\, \vec{X} \,/\, \vec{t}\,]\}.$$

   Since $s_1 = s_0 \oplus_E \Delta_{0-1}$, the equality

$$T^{s_0}_{A, \Delta_{0-1} \oplus \Delta_C} = T^{s_1}_{A, \Delta_C}$$

   follows by the properties of the transition system (see Definition 4.2).

The case $T^{s_0}_{A,\Delta_{0-1}\oplus\Delta_C} = \emptyset$ is trivial:

$$\begin{array}{ll} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longleftrightarrow & \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \\ \Longleftrightarrow & (\Delta_C, \Delta) \in I_1^{s_1}(\#\,\vec{X}\,[A \mapsto \varphi]) \end{array}$$

For the complementary case we get:

$$\begin{array}{ll} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\#\,\vec{X}\,[A \mapsto \varphi]) \\ \Longleftrightarrow & \text{there exists a function } f : T^{s_0}_{A,\Delta_{0-1}\oplus\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T^{s_0}_{A,\Delta_{0-1}\oplus\Delta_C} :\ (\Delta_{0-1} \oplus \Delta_C, f(\,\vec{t}\,)) \in I_0^{s_0}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus\ \bigsqcup_{(\vec{t}) \in T^{s_0}_{A,\Delta_{0-1}\oplus\Delta_C}} f(\,\vec{t}\,) \\ \Longleftrightarrow & \text{there exists a function } f : T^{s_1}_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ & \forall(\,\vec{t}\,) \in T^{s_1}_{A,\Delta_C} :\ (\Delta_{0-1} \oplus \Delta_C, f(\,\vec{t}\,)) \in I_0^{s_0}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus\ \bigsqcup_{(\vec{t}) \in T^{s_1}_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longleftrightarrow & \text{there exists a function } f : T^{s_1}_{A,\Delta_C} \to \mathcal{T}_{Cons} \text{ such that :} \\ (IH) & \forall(\,\vec{t}\,) \in T^{s_1}_{A,\Delta_C} :\ (\Delta_C, f(\,\vec{t}\,)) \in I_1^{s_1}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \\ & \text{and } \Delta = Log(A[\,\vec{X}\,/\,\vec{all}\,]) \ \oplus\ \bigsqcup_{(\vec{t}) \in T^{s_1}_{A,\Delta_C}} f(\,\vec{t}\,) \\ \Longleftrightarrow & (\Delta_C, \Delta) \in I_1^{s_1}(\#\,\vec{X}\,[A \mapsto \varphi]) \end{array}$$

6. Implication

$$\begin{array}{ll} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi \to \psi) \\ \Longleftrightarrow & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi) \Longrightarrow (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\psi) \\ \Longleftrightarrow & (\Delta_{0-1} \oplus \Delta_C, \Delta) \notin I_0^{s_0}(\varphi) \text{ or } (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\psi) \\ \Longleftrightarrow & (\Delta_C, \Delta) \notin I_1^{s_1}(\varphi) \text{ or } (\Delta_C, \Delta) \in I_1^{s_1}(\psi) \\ (IH) & \\ \Longleftrightarrow & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi) \Longrightarrow (\Delta_C, \Delta) \in I_1^{s_1}(\psi) \\ \Longleftrightarrow & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi \to \psi) \end{array}$$

7. Universal quantification

Consider a quantification $\forall\,\vec{X}\,\varphi$, where $\vec{X} = X_1, \ldots, X_n$.

$$\begin{array}{ll} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\forall\,\vec{X}\,\varphi) \\ \Longleftrightarrow & \text{for all ground term tuples } (\,\vec{t}\,) \in \mathcal{U}^n, \\ & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in I_0^{s_0}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \text{ holds} \\ \Longleftrightarrow & \text{for all ground term tuples } (\,\vec{t}\,) \in \mathcal{U}^n, \\ (IH) & (\Delta_C, \Delta) \in I_1^{s_1}(\varphi[\,\vec{X}\,/\,\vec{t}\,]) \text{ holds} \\ \Longleftrightarrow & (\Delta_C, \Delta) \in I_1^{s_1}(\forall\,\vec{X}\,\varphi) \end{array}$$

□

Theorem 6.27 [Minimal Model w.r.t. Different Initial States] Let $s_0, s_1 \in \mathcal{S}$ be arbitrary states and $\Delta_{0-1} \in \mathcal{T}_{Cons}$ be a consistent transition such that $s_1 = s_0 \oplus_E \Delta_{0-1}$. Let φ be a ground update formula. The following property holds for the minimal model of P_{UP} w.r.t. the initial states s_0 and s_1, respectively:

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} :\ (\Delta_{0-1} \oplus \Delta_C, \Delta) \in M_{UP}[P_{UP}, s_0](\varphi) \Longleftrightarrow (\Delta_C, \Delta) \in M_{UP}[P_{UP}, s_1](\varphi)$$

Proof: It is sufficient to prove the assertion for definable update atoms. The assertion for arbitrary update formulas follows by Lemma 6.26.

We prove the assertion by transfinite induction. For a given ordinal γ we show that the following condition holds for all ground definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$:

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} : \ (\Delta_{0-1} \oplus \Delta_C, \Delta) \in T^{s_0}_{P_{UP}} \uparrow \gamma\,(p(\vec{t})) \iff (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}} \uparrow \gamma\,(p(\vec{t}))$$

Base case: $\gamma = 0$

Let $p(\vec{t}) \in \mathcal{B}_{DU}$ be a definable update atom. Because $I_\bot(p(\vec{t})) = \emptyset$, the condition trivially holds.

Induction step: successor ordinal $\alpha + 1$

By the induction hypothesis, the desired condition holds w.r.t. $T^{s_0}_{P_{UP}} \uparrow \alpha$ and $T^{s_1}_{P_{UP}} \uparrow \alpha$. By Lemma 6.26, consequently, for all ground update formulas φ the following holds:

$$\forall \Delta_C, \Delta \in \mathcal{T}_{Cons} : (\Delta_{0-1} \oplus \Delta_C, \Delta) \in (T^{s_0}_{P_{UP}} \uparrow \alpha\,)^{s_0}(\varphi) \iff (\Delta_C, \Delta) \in (T^{s_1}_{P_{UP}} \uparrow \alpha\,)^{s_1}(\varphi)$$

Now choose a definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$. For arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$, the following equivalences hold.

$$\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in T^{s_0}_{P_{UP}} \uparrow \alpha + 1\,(p(\vec{t})) \\
\underset{(4.86)}{\iff} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in T^{s_0}_{P_{UP}}(T^{s_0}_{P_{UP}} \uparrow \alpha)(p(\vec{t})) \\
\underset{(4.83)}{\iff} & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
 & \text{such that } (\Delta_{0-1} \oplus \Delta_C, \Delta) \in (T^{s_0}_{P_{UP}} \uparrow \alpha\,)^{s_0}(U) \\
\underset{(IH)}{\iff} & \text{there exists a ground instance } U \to p(\vec{t}) \text{ of a rule } r \in P_{UP}, \\
 & \text{such that } (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}} \uparrow \alpha\,)^{s_1}(U) \\
\underset{(4.83)}{\iff} & (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}}(T^{s_1}_{P_{UP}} \uparrow \alpha)(p(\vec{t})) \\
\underset{(4.86)}{\iff} & (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}} \uparrow \alpha + 1\,(p(\vec{t}))
\end{array}$$

Induction step: limit ordinal β

By the induction hypothesis, the desired condition holds w.r.t. all ordinals $\alpha < \beta$.

Now choose a definable update atom $p(\vec{t}) \in \mathcal{B}_{DU}$. For arbitrary $\Delta_C, \Delta \in \mathcal{T}_{Cons}$, the following equivalences hold.

$$\begin{array}{ll}
 & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in T^{s_0}_{P_{UP}} \uparrow \beta\,(p(\vec{t})) \\
\underset{(4.75)}{\iff} & (\Delta_{0-1} \oplus \Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T^{s_0}_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\underset{}{\iff} & \text{there exists } \alpha < \beta \text{ such that } (\Delta_{0-1} \oplus \Delta_C, \Delta) \in T^{s_0}_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\underset{(IH)}{\iff} & \text{there exists } \alpha < \beta \text{ such that } (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\underset{}{\iff} & (\Delta_C, \Delta) \in \bigcup_{\alpha<\beta} T^{s_1}_{P_{UP}} \uparrow \alpha\,(p(\vec{t})) \\
\underset{(4.75)}{\iff} & (\Delta_C, \Delta) \in T^{s_1}_{P_{UP}} \uparrow \beta\,(p(\vec{t}))
\end{array}$$

Next, we have to prove the main assertion of the theorem. Let us choose an ordinal α that is greater than both closure ordinals $\alpha_{P_{UP}}^{s_0}$ and $\alpha_{P_{UP}}^{s_1}$ for the minimal model of P_{UP} w.r.t. s_0 and s_1 (cf. Theorem 4.87). The assertion follows directly by the inequalities

$$T_{P_{UP}}^{s_0} \uparrow \alpha \leq M_{UP}[P_{UP}, s_0] = T_{P_{UP}}^{s_0} \uparrow \alpha_{P_{UP}}^{s_0} \leq T_{P_{UP}}^{s_0} \uparrow \alpha$$

and

$$T_{P_{UP}}^{s_1} \uparrow \alpha \leq M_{UP}[P_{UP}, s_1] = T_{P_{UP}}^{s_1} \uparrow \alpha_{P_{UP}}^{s_1} \leq T_{P_{UP}}^{s_1} \uparrow \alpha$$

which can be derived from Theorem 4.87 using fixpoint properties (see [Llo87] for details). □

Theorem 6.27 states that the model-theoretic interpretations of an update formula φ w.r.t. different initial states are related to each other. The left part of the equivalences above treats s_1 as a hypothetical state represented by Δ_{0-1} w.r.t. the initial state s_0, while the right part treats s_1 as the initial state. The result is not very surprising but shows that the deferred update semantics is well-defined.

Finally, we present a consequence of Theorem 6.27 concerning possible transitions for an update query $\leftarrow U$.

Corollary 6.28 Let P_{UP} be an update program and let $s_0 \in \mathcal{S}$ be a fixed state. Let $s_1 \in \mathcal{S}$ be another state that is reachable from s_0 by a consistent transition $\Delta_{0-1} \in \mathcal{T}_{Cons}$, i.e. $s_1 = s_0 \oplus_E \Delta_{0-1}$. Let U be a ground update goal and $\Delta \in \mathcal{T}_{Cons}$ be a consistent transition.

Then Δ is a possible transition for the update query $\leftarrow U$ w.r.t. the initial state s_1, iff $(\Delta_{0-1}, \Delta) \in M_{UP}[P_{UP}, s_0](U)$.

Proof: The assertion follows from Theorem 6.27 with $\Delta_C := \Delta_\varepsilon$. □

Informally speaking, Corollary 6.28 states that the possible transitions w.r.t. an arbitrary initial state s_1 are expressible by the minimal model w.r.t. a fixed state s_0. Note that in many transition systems s_0 can be chosen as a trivial state, e.g. the state of the empty database. The model-theoretic semantics w.r.t. s_0 captures the semantics w.r.t. all states reachable from s_0, too.

7 Relations between ULTRA and other Approaches

In this section we are going to contrast the ULTRA approach with some related work that was already mentioned in Section 2. After summarizing the essential contributions of ULTRA, we will show in more detail how ULTRA can be compared to abductive logic programming and (Concurrent) Transaction Logic [BK94, BK96]. The last subsection is devoted to the principle of monadic programming in functional languages, which shows some similarities with the foundations of ULTRA.

7.1 Essentials of the ULTRA Approach

ULTRA has been defined as a rule-based update specification language. It allows the modular construction of complex operations with the possibility of reuse. We have defined constructs to build concurrent and sequential operations, further we enable the specification of set-oriented operations as known from the database world in a natural style. This way, we have created a universal language that integrates the programming features from various rule-based approaches, e.g. [BK94, Che97, MBM97, MW88b], as well as from procedural programming languages used for the implementation of information systems (cf. Sections 2.1 and 2.2). The compact syntax of ULTRA has been derived from conventional logic programming languages, but it could easily be replaced by a more verbose syntax. The main contribution of the ULTRA approach is the development of a logical semantics that assigns a unique minimal model to every update program. This model captures the meaning of all operations specified by the program. The minimal model is defined independently of any particular operational setting and remains unchanged when the program is transformed according to rewriting strategies valid in the field of logic databases. Consequently, the ULTRA semantics extends the declarative concepts of logic databases in a comprehensible way. The model-theoretic semantics and its fixpoint characterization generate a solid foundation for transactional execution strategies and run-time optimizations.

The second point that distinguishes ULTRA from most other approaches that deal with the specification of update operations is the framework concept. The generic ULTRA language abstracts from particular basic operations, and its semantics leaves the concrete notion of states and transition objects aside. Hence, when an instance of the framework is created, the missing objects can be defined according to the application domain. Many other approaches, e.g. [Che97, LHL95, MBM97, MW88b], define language and semantics in more concrete terms and merely for a database domain.

7.2 Abduction and View Updates

Abduction [EK89] is a form of logical backward reasoning that is used to find causes for observed or desired effects. In the more specific context of logic databases, effects correspond to IDB atoms defined by rules, and causes correspond to EDB atoms. In contrast to a deduction, where the facts of the EDB are given and the facts of the IDB are derived, an abduction searches for base facts which imply a given fact in the IDB. The rules are considered in a top-down fashion, and the *abducibles*, which form a subset of the EDB atoms, can be chosen as true or false in order to make a rule-body become true. Consistency constraints must be regarded such that no abducible is being chosen as true and false at the same time. In general, the result of an abduction is non-deterministic. An abductive framework can be expressed as a deductive system within a disjunctive logic programming environment (see [IS96] for details).

Example 7.1 [Abduction] Let r be an EDB predicate and p be an IDB predicate defined by the following rules:

$$\begin{array}{lcl} p(X) & \leftarrow & r(X,a),\ r(a,X) \\ p(X) & \leftarrow & r(b,X) \end{array}$$

Abductive reasoning on the query $\leftarrow p(c)$ results in the two solutions $\{r(c,a), r(a,c)\}$ and $\{r(b,c)\}$, provided that the atoms over r are contained in the set of abducibles. □

In the ULTRA semantics, case (BU) of Definition 4.9 introduces an abductive component. Consider, for example, the ULTRA instance based on partially ordered multi-sets (see Section 4.4), and assume that the mapping Upd^{act} is defined as the identity. Then for every ground basic update atom $u(\vec{t})$ and every consistent transition Δ_C, the pair $(\Delta_C, \{u(\vec{t})\})$ is contained in the interpretation of the formula $u(\vec{t})$. This can be interpreted as follows: to make the basic update atom $u(\vec{t})$ successful, the corresponding action $u(\vec{t})$ must be included into the pomset that represents the result. Recall that the singletons are combined by $\sqcup$ and $\oplus$ to build more complex pomsets. This corresponds to the accumulation of truth values chosen for abducibles. Furthermore, consistency constraints can be defined for transitions in order to exclude intractable combinations. Consequently, the ULTRA approach can be regarded as an extended form of abduction. A possible transition Δ for a query $\leftarrow \varphi$ submitted to an update program can be compared to an abductive result of the query $\leftarrow \varphi$.

Although there are still differences, the ULTRA instance for insertions and deletions, which is based on update request sets, is very closely related to abduction. In the following informal comparison, we will restrict ourselves to goals that are concurrent conjunctions of update literals, such that it is not necessary to deal with intermediate states. Moreover, we assume that no intensional DB predicates exist, that I_{DB} is two-valued in the initial state DB_0, and that no logging transitions different from the empty set are assigned to extensional DB atoms. We show how to transform an update program P (based on the restricted syntax) into a normal logic program P' which can be used for abduction and then has essentially the same semantics as the original program.

Let an update program P be given. In the following three steps we construct the abductive framework, i.e. the set of abducibles and the normal logic program P'. First, the EDB predicates are duplicated, such that for every predicate r also a second predicate r' with the same arity exists. The new predicates refer to basic update requests and thus to the next state. Atoms built over these new predicates are treated as abducibles. Secondly, we encode the truth interpretation of each ground atom $r(\vec{t})$ in the fixed initial state DB_0 – given by $I_{DB}(DB_0)$ – by generating additional facts for the resulting program P'. Finally, we rewrite the rules of P as follows, such that they can be included into P': each subgoal $INS\ r(\vec{t})$ is transformed to $r'(\vec{t})$, each subgoal $DEL\ r(\vec{t})$ is transformed to $NOT\ r'(\vec{t})$, and each subgoal NOP is removed. Note that the concurrent conjunction implicitly becomes a conventional conjunction. See Example 7.2 below for an illustration of the transformation.

Now let a ground update query $\leftarrow \varphi$ be given, where φ only consists of definable update atoms (w.l.o.g.). Then every possible transition Δ of $\leftarrow \varphi$ w.r.t. P is an abductive result (modulo renaming of the basic update requests $+r(\vec{t})$ and $-r(\vec{t})$ to $r'(\vec{t})$ and $\neg r'(\vec{t})$, respectively) of $\leftarrow \varphi$ w.r.t. the rewritten program P' and vice versa.

We would like to give an informal proof. Since there is no negation through definable update atoms and neither bulk quantifications, nor sequential conjunctions occur, we can apply a simple top-down resolution technique for P. Recall that the model-theoretic semantics of ULTRA is equivalent

to a fixpoint semantics w.r.t. an immediate consequence operator (cf. Section 4.6). This operator can also be used to reason backwards. For P' we can apply arbitrary standard techniques, since P' is a semi-definite logic program [Llo87]. EDB literals are resolved in ULTRA in the same way as in the abductive framework, where these literals do not belong to the abducibles. The only difference is that the base facts are explicitly encoded in the abductive framework, while the reference to the state DB_0 in ULTRA is implicit. Basic update atoms can always be resolved in ULTRA by collecting basic update requests. This corresponds to taking an assumption in the abductive framework in order to resolve the corresponding abducibles. In the ULTRA semantics as well as in the abduction semantics the results must not become inconsistent: the insertions and deletions must be consistent, the truth values for the abducibles must be chosen uniquely. Whenever an update request is collected more than once in ULTRA, these collections are idempotent. On the other side, an already chosen abducible can be resolved without further assumptions. A subgoal NOP behaves neutral in ULTRA and thus can always be resolved without any generation of update requests. Recall that NOP was already eliminated at the construction of the abductive framework.

So, we can see that both paradigms work similarly. The abduction technique looks simple, as we do not have negation through definable update atoms and no access to the result state. The latter would require additional frame rules and more consistency checks, if it had to be handled by abduction.

Example 7.2 [ULTRA **and Abduction**] Let the following update program P be given. Note that r is an EDB predicate.

$$\begin{array}{lcl} p(X) & \leftarrow & r(X),\ DEL\ r(X),\ INS\ r(a) \\ p(X) & \leftarrow & r(X),\ DEL\ r(b) \end{array}$$

Further, let $I_{DB}(DB_0) \models r(a)$ hold, and let $I_{DB}(DB_0) \models \neg A$ hold for all other DB atoms $A \in \mathcal{B}$.

The transformed program P' looks as follows:

$$\begin{array}{lcl} p(X) & \leftarrow & r(X),\ NOT\ r'(X),\ r'(a) \\ p(X) & \leftarrow & r(X),\ NOT\ r'(b) \\ r(a) & & \end{array}$$

Next, let the update query $\leftarrow p(a)$ be asked. In both programming environments it is possible to resolve $p(a)$ and $r(a)$. Then the first rule of the update program P will produce the update requests $-r(a)$ and $+r(a)$ which are conflicting with each other, while the abduction over the first rule of the rewritten program P' will require to make the abducible $r'(a)$ false and true at the same time. Thus, the first rule in both programs does not offer a solution. The second rule of the update program P will produce the update request set $\{-r(b)\}$, while the second rule of the rewritten program P' will generate the abductive result $\{\neg r'(b)\}$. □

In the following we are going to refer to the *view update* problem [Bry90, KM90]. When an update request on a view is given, then it has to be translated in a set of changes on the base tables. In contrast to *view computation* and *view maintenance*, which both have a deductive character, processing an update request on a view is an abductive problem (see [TU95] for a classification of problems concerning views). The view update problem is also relevant in the context of logic databases, since the IDB can be regarded as a set of views over the EDB. The approach of implicit

view updates only takes static information, namely the view definitions, into account, although these information do not describe changes on the view. The effects of an automatically performed view update are non-deterministic and often inadequate. Below we will show that an explicit treatment of view updates is more applicable in practice. However, the theoretical results of implicit view updates can be combined with the ULTRA paradigm. Update rules that specify view updates can be generated per default at view definition time. These rules explicitly describe the formerly implicit view updates. The rules can be made visible to the programmer of the views, who can also make some own modifications. For instance, the programmer may want to exclude inconvenient solutions or attach further operations to a view update. In a conventional database setting, these modifications could only be implemented using integrity constraints and trigger concepts. Manchanda and Warren [MW88b] describe a method of deriving rules for updating views whose definitions satisfy certain constraints. The results can surely be extended to more complex views using more sophisticated abduction techniques.

Example 7.3 [View Updates] Let us consider a simplified workflow application. The current instances of business processes are stored in a relation $bp(ProcNr, ProcType)$, where $ProcNr$ is the instance number and $ProcType$ is the process type, e.g. *order*, *review*, *maintenance*. Further, a relation $status(ProcNr, St)$ is provided that assigns a status St, e.g. *open*, *closed*, to each current process $ProcNr$.

Next, we assume that two views $orders(ProcNr)$ and $open\_orders(ProcNr)$ must be specified, which contain the process numbers of all orders and the orders that are not finished yet, respectively. The views may have to be created due to security issues or due to legacy software that is being used in the department responsible for the orders. The following rules are suitable to define the desired views.

$$\begin{array}{lcl} orders(X) & \leftarrow & bp(X, order) \\ open\_orders(X) & \leftarrow & orders(X),\ status(X, open) \end{array}$$

Using abductive reasoning techniques, a compiler could determine the following update rules (in ULTRA syntax) which describe some update operations on the views. Note that $INS\ orders(X)$, $DEL\ orders(X)$, and $DEL\ open\_orders(X)$ are *definable* update atoms in this example.

$$\begin{array}{lcl} INS\ orders(X) & \leftarrow & INS\ bp(X, order) \\ DEL\ orders(X) & \leftarrow & DEL\ bp(X, order) \\ DEL\ open\_orders(X) & \leftarrow & NOT\ orders(X) \\ DEL\ open\_orders(X) & \leftarrow & orders(X),\ DEL\ status(X, open) \\ DEL\ open\_orders(X) & \leftarrow & NOT\ status(X, open) \\ DEL\ open\_orders(X) & \leftarrow & DEL\ orders(X),\ status(X, open) \end{array}$$

From the semantical point of view, most of the rules are acceptable. However, the last rule is curious: instead of a natural change of the status, it votes for a unnatural deletion of the process instance. If the deletion on the view *open_orders* is used to complete an order, this is not a correct solution. In a database system where view updates are automatically performed, such inadequate effects can arise, but in this explicit setting, a programmer can delete the last rule manually and thus exclude the corresponding results.

Nevertheless, it should be mentioned that the view definitions alone do not contain enough semantics for realistic updates. In our example it is straight-forward to assume that the status must be

modified from *open* to *closed*, whenever an order has been completed. Similarly, when an order is inserted, both base relations should be modified adequately. For such sophisticated updates, view update techniques come to their borders. However, in ULTRA one can easily specify the following operations that perform the desired tasks.

$$\begin{aligned} add\_order(X) &\leftarrow INS\ bp(X, order),\ INS\ status(X, open) \\ close\_order(X) &\leftarrow open\_orders(X),\ DEL\ status(X, open),\ INS\ status(X, closed) \end{aligned}$$

Together with the operations *add_order* and *close_order*, the IDB relation *orders* can be seen as a data object. □

7.3 ULTRA versus (Concurrent) Transaction Logic

(Concurrent) Transaction Logic [BK94, BK96] is an update concept similar to ULTRA. The standard predicate logic is extended by some special connectives, and a new semantics is defined. In contrast to the ULTRA semantics, which is based on transitions Δ between states, the semantics of Transaction Logic [BK94] is based on paths $\langle s_0, \ldots, s_n \rangle$ of states. A rule-based fragment of the logic is defined together with a model-theoretic and a proof-theoretic semantics. Although the formal developments and the notation are different from those of the ULTRA approach, the overall concepts of rules, models, immediate consequences, etc. are essentially the same and generalize the well-known concepts of logic databases [Llo87]. The computable semantics is restricted to sequential update programs without negation through defined predicates, i.e. the rule bodies must be built using only the sequential conjunction $\otimes$, and negation may only occur at the base level. In Section 7.3.1 we show that under some minor restrictions, the computable semantics of Transaction Logic can be captured within the ULTRA framework. Concurrent Transaction Logic [BK96] is an extension of Transaction Logic to a parallel programming language. The extensions are significantly different from the extensions made within the ULTRA approach. This will be discussed in more detail in Section 7.3.2.

7.3.1 Sequential Operations

Next, we want to work out the similarities between ULTRA and Transaction Logic. For this purpose, we restrict ourselves to sequentially composed update goals in both languages. In ULTRA, we refer to the instance based on partially ordered multi-sets (see Section 4.4). The restriction to the sequential fragment, however, implies that the relevant pomsets are finite and linear and can thus be represented by lists (cf. Remark 4.48). We modify the logging transition assignment *Log* such that it always yields the neutral transition []. This is legitimate, since the read-isolation problem does not lie in the scope of this comparison. For the sake of clarity, we assume that Transaction Logic distinguishes between DB predicates $Pred_{DB}$, basic update predicates $Pred_{BU}$, and definable update predicates $Pred_{DU}$, i.e. we want to avoid an overloading of predicate symbols. ULTRA (although partially instantiated by the pomset semantics) and Transaction Logic can be both regarded as frameworks with some open parameters. If we have chosen the settings described above and fix the sets of predicate symbols and the Herbrand universe $\mathcal{U}$, the following parameters remain open: For the ULTRA instance we must provide the set of states $\mathcal{S}$, the set of actions Σ, the atomic execution function *do*, the state interpretation I_{DB}, and the mapping Upd^{act}. For Transaction Logic we must provide the set of states $\mathcal{S}$, a *data oracle* $\mathcal{O}^d$, and a *transition oracle* $\mathcal{O}^t$. Essentially, the

data oracle describes truth values of DB atoms in a state and is thus comparable to I_{DB}, while the transition oracle provides a relation between pairs of states and basic update atoms and is thus comparable to the function *do*. See [BK96] for the exact definitions of the oracles. Note that Σ and Upd^{act} rather have formal justifications in the ULTRA context.

However, to be actually comparable we must define two significant restrictions: the DB interpretation I_{DB} has to be two-valued in ULTRA, and the transition oracle $\mathcal{O}^t$ has to be totally functional in Transaction Logic. Otherwise there may arise problems with negated DB atoms or hypothetical executions.

Definition 7.4 A transition oracle $\mathcal{O}^t$ is called *totally functional*, if for every ground DB atom $u(\vec{t}) \in \mathcal{B}_{BU}$ and every state $s_1 \in \mathcal{S}$, there exists exactly one state $s_2 \in \mathcal{S}$ such that

$$u(\vec{t}) \in \mathcal{O}^t(s_1, s_2)$$

holds. □

In the following, we define a mapping from the ULTRA environment onto the Transaction Logic environment: Let $\mathcal{S}$, Σ, *do*, I_{DB}, and Upd^{act} be given, where I_{DB} is two-valued in every state $s \in \mathcal{S}$. We define the data oracle $\mathcal{O}^d$ by

$$A \in \mathcal{O}^d(s) \;:\Longleftrightarrow\; I_{DB}(s) \models A$$

for all ground DB atoms $A \in \mathcal{B}$ and all states $s \in \mathcal{S}$. Note that $\mathcal{O}^d$ is implicitly extended to negations, conjunctions, disjunctions, etc. We only deal with negated atoms $\neg A$ and their standard semantics

$$\neg A \in \mathcal{O}^d(s) \iff A \notin \mathcal{O}^d(s).$$

We define the transition oracle $\mathcal{O}^t$ by

$$u(\vec{t}) \in \mathcal{O}^t(s_1, s_2) \;:\Longleftrightarrow\; do(Upd^{act}(u(\vec{t})), s_1) = s_2$$

for all ground basic update atoms $u(\vec{t}) \in \mathcal{B}_{BU}$ and arbitrary states $s_1, s_2 \in \mathcal{S}$.

In the following, we define a converse mapping from Transaction Logic to ULTRA: Let $\mathcal{S}$, $\mathcal{O}^d$, and $\mathcal{O}^t$ be given, where $\mathcal{O}^t$ is totally functional. We define I_{DB} by

$$\begin{array}{lll} I_{DB}(s) \models A & :\Longleftrightarrow & A \in \mathcal{O}^d(s) \\ I_{DB}(s) \models \neg A & :\Longleftrightarrow & \neg A \in \mathcal{O}^d(s) \end{array}$$

for all ground DB atoms $A \in \mathcal{B}$ and all states $s \in \mathcal{S}$. We define $\Sigma := \mathcal{B}_{BU}$ and Upd^{act} as the identity mapping. Finally, we define *do* by

$$do(u(\vec{t}), s_1) := s_2$$

for all ground basic update atoms $u(\vec{t}) \in \mathcal{B}_{BU}$ and all states $s_1 \in \mathcal{S}$, where s_2 denotes the uniquely defined state for which $u(\vec{t}) \in \mathcal{O}^t(s_1, s_2)$ holds.

Lemma 7.5 Both mappings between the programming environments guarantee the following properties:

1. Let $A \in \mathcal{B}$ be a ground DB atom and $s \in \mathcal{S}$ be a state. Then the following holds:

   (a) $I_{DB}(s) \models A \iff A \in \mathcal{O}^d(s)$
   (b) $I_{DB}(s) \models \neg A \iff \neg A \in \mathcal{O}^d(s)$

2. Let $u(\vec{t})$ be a basic update atom and $s_1, s_2 \in \mathcal{S}$ be states. Then

   $$do(Upd^{act}(u(\vec{t})), s_1) = s_2 \iff u(\vec{t}) \in \mathcal{O}^t(s_1, s_2)$$

   holds.

Proof: The proof of the assertions is trivial. □

Transaction Logic formulas are interpreted over a *path structure*. A path structure I entails a set of ground formulas φ (denoted by $I, \langle s_0, \ldots, s_n \rangle \models \varphi$) for a given a path $\langle s_0, \ldots, s_n \rangle$ of states $s_i \in \mathcal{S}$. In analogy to ULTRA, the interpretation of the definable update atoms is directly given by I, whereas the interpretation of DB literals, basic update atoms, and the sequential conjunction is defined inductively. A path interpretation I is a model of a program, iff the entailment (w.r.t. a path) of the body of an arbitrary rule instance implies the entailment of the head. The formal definitions are slightly different, but correspond to this notion of a model. Transaction Logic does not talk about a unique minimal model but considers formulas that hold in all models. However, like in ULTRA it would be possible to define a model intersection and construct a least model. The proof-theoretic results for Transaction Logic show that the intended semantics of a program can be computed by an immediate consequence operator.

In the next step, we will show that the semantics of composed goals is essentially the same in both programming environments. Lemma 7.5 is applied to the cases of update literals, and the essential work lies in comparing the sequential conjunction ":" of ULTRA with the sequential conjunction $\otimes$ of Transaction Logic. Note that both conjunctions have turned out to be associative.

Definition 7.6 Let $s_0 \in \mathcal{S}$ be a fixed initial state. Let $I \in \mathcal{I}$ be an interpretation of update formulas (w.r.t. state s_0), and let I' be a path interpretation over $\mathcal{S}$. Let φ be a ground sequential update goal in the ULTRA syntax.

I and I' are called *coinciding* on φ, if for arbitrary sequences $s'_0, \ldots, s'_n$ of states $s'_i \in \mathcal{S}$ and arbitrary consistent transitions $\Delta_C \in \mathcal{T}_{Cons}$ with $s_0 \oplus_E \Delta_C = s'_0$ the following property holds, where φ' results from φ by replacing each occurrence of ":" by $\otimes$ and NOP by $true$.

$$I', \langle s'_0, \ldots, s'_n \rangle \models \varphi' \iff \begin{array}{l} \text{there exist actions } a_1, \ldots, a_n \in \Sigma \text{ such that :} \\ (\Delta_C, [a_1, \ldots, a_n]) \in I(\varphi) \\ \text{and } \forall i \in \{1, \ldots, n\} : do(a_i, s'_{i-1}) = s'_i \end{array}$$

□

Lemma 7.7 Let s_0, I, and I' be given as in Definition 7.6. If I is coinciding with I' on all definable update atoms $p(\vec{t}) \in \mathcal{B}_{DU}$, then I is coinciding with I' on arbitrary sequential update goals φ.

Proof: We prove the assertion by structural induction. In each case shown below, we choose arbitrary sequences $s'_0, \ldots, s'_n$ of states $s'_i \in \mathcal{S}$ and transitions $\Delta_C \in \mathcal{T}_{Cons}$ with $s_0 \oplus_E \Delta_C = s'_0$ and show the desired equivalence.

Base cases:

1. DB literal

   We only show the assertion for a positive DB literal $A \in \mathcal{B}$. The proof for a negative DB literal is entirely analogous.

   '$\Rightarrow$':

   Assume that

   $$I', \langle s'_0, \ldots, s'_n \rangle \models A$$

   holds. Since we do not allow an overloading of predicate symbols, A holds on a singleton paths conforming with the data oracle, i.e. $n = 0$ and $A \in \mathcal{O}^d(s'_0)$. By Lemma 7.5,

   $$I_{DB}(s'_0) \models A$$

   must hold. Since $s_0 \oplus_E \Delta_C = s'_0$ holds by precondition,

   $$(\Delta_C, [\,]) \in I(A)$$

   follows directly by case (DB) of Definition 4.9.

   '$\Leftarrow$':

   Assume that

   $$(\Delta_C, \Delta) \in I(A)$$

   holds. Then, by definition, $I_{DB}(s_0 \oplus_E \Delta_C) \models A$ and $\Delta = [\,]$ holds. So again, we only have to consider the case $n = 0$. Using the arguments of the direction '$\Rightarrow$' backwards, it is possible to show that

   $$I', \langle s'_0 \rangle \models A$$

   holds.

2. NOP literal

   Recall that NOP is replaced by *true* when moving from ULTRA to Transaction Logic.

   The special logic formula *true* is interpreted (by the data oracle) as valid for arbitrary singleton paths, and NOP always yields the neutral transition $[\,]$. Thus, the proof for the case of the DB literals can easily be adapted.

3. Basic update atom

   '$\Rightarrow$':

   Assume that

   $$I', \langle s'_0, \ldots, s'_n \rangle \models u(\vec{t})$$

   holds. Since we do not allow an overloading of predicate symbols, $u(\vec{t})$ holds on a path of length 2 conforming with the transition oracle, i.e. $n = 1$ and $u(\vec{t}) \in \mathcal{O}^t(s'_0, s'_1)$. By Lemma 7.5,

   $$do(Upd^{act}(u(\vec{t})), s'_0) = s'_1$$

must hold. In the ULTRA context, by case (BU) of Definition 4.9,

$$(\Delta_C, [Upd^{act}(u(\vec{t}))]) \in I(u(\vec{t}))$$

holds. Define $a_1 := Upd^{act}(u(\vec{t}))$. The desired conclusion follows immediately.

'$\Leftarrow$':

Assume that

$$(\Delta_C, \Delta) \in I(u(\vec{t}))$$

holds. Then, by definition, $\Delta = [Upd^{act}(u(\vec{t}))]$ holds. So again, we only have to consider the case $n = 1$ with $a_1 = Upd^{act}(u(\vec{t}))$. Provided that the precondition

$$do(a_1, s'_0) = s'_1$$

also holds, it is possible to show that

$$I', \langle s'_0, s'_1 \rangle \models u(\vec{t})$$

holds. The arguments of the direction '$\Rightarrow$' can be applied backwards.

4. Definable update atom

   The equivalence holds by the precondition.

Induction step:

We only have to consider sequentially composed goals. Recall that ":" is replaced by $\otimes$ when moving from ULTRA to Transaction Logic. The subgoals of a composed goal $\varphi : \psi$ correspond to the subgoals of the rewritten goal $\varphi' \otimes \psi'$, and we can apply the induction hypothesis w.r.t. φ and ψ.

'$\Rightarrow$':

Assume that

$$I', \langle s'_0, \ldots, s'_n \rangle \models \varphi' \otimes \psi'$$

holds. Then, by the definition of $\otimes$, there exists an index $i \in \{0, \ldots, n\}$ such that the conditions

$$I', \langle s'_0, \ldots, s'_i \rangle \models \varphi'$$

and

$$I', \langle s'_i, \ldots, s'_n \rangle \models \psi'$$

hold. By the induction hypothesis for the path $\langle s'_0, \ldots, s'_i \rangle$, there must exist actions $a_1, \ldots, a_i \in \Sigma$ such that

$$(\Delta_C, [a_1, \ldots, a_i]) \in I(\varphi)$$

and

$$\forall j \in \{1, \ldots, i\} : do(a_j, s'_{j-1}) = s'_j$$

holds. Define $\Delta'_C := \Delta_C \oplus [a_1, \ldots, a_i]$. Using Definition 4.53 inductively, it is possible to show that $s'_0 \oplus_E [a_1, \ldots, a_i] = s'_i$ holds. Consequently, $s_0 \oplus_E \Delta'_C = s'_i$ holds, too. Now we can apply the induction hypothesis for the path $\langle s'_i, \ldots, s'_n \rangle$: there must exist further actions $a_{i+1}, \ldots, a_n \in \Sigma$ such that

$$(\Delta'_C, [a_{i+1}, \ldots, a_n]) \in I(\psi)$$

and

$$\forall j \in \{i+1, \ldots, n\} : do(a_j, s'_{j-1}) = s'_j$$

holds. Note that the equality $[a_1, \ldots, a_i] \oplus [a_{i+1}, \ldots, a_n] = [a_1, \ldots, a_n]$ holds and that $[a_1, \ldots, a_n]$ is a consistent transition. Thus, by case (SCj) of Definition 4.9

$$(\Delta_C, [a_1, \ldots, a_n]) \in I(\varphi : \psi)$$

holds, which completes the proof of the desired conclusion.

'$\Leftarrow$':

Assume that for some actions $a_1, \ldots, a_n \in \mathcal{S}$

$$(\Delta_C, [a_1, \ldots, a_n]) \in I(\varphi : \psi)$$

and

$$\forall j \in \{1, \ldots, n\} : do(a_j, s'_{j-1}) = s'_j$$

holds. Then, by definition, there exist consistent transitions $\Delta_1, \Delta_2 \in \mathcal{T}_{Cons}$ such that

$(\Delta_C, \Delta_1) \in I(\varphi)$
and $(\Delta_C \oplus \Delta_1, \Delta_2) \in I(\psi)$
and $[a_1, \ldots, a_n] = \Delta_1 \oplus \Delta_2$

holds. Both Δ_1 and Δ_2 must be linear pomsets, and obviously equal to $[a_1, \ldots, a_i]$ and $[a_{i+1}, \ldots, a_n]$, respectively, where i is some index in $\{0, \ldots, n\}$. (The cases $i = 0$ and $i = n$ correspond to solutions where one of the pomsets is the empty list $[\,]$.) Using the induction hypothesis for the transition $[a_1, \ldots, a_i]$, one can show that

$$I', \langle s'_0, \ldots, s'_i \rangle \models \varphi'$$

holds. Further, for $\Delta'_C := \Delta_C \oplus [a_1, \ldots, a_i]$, the property $s_0 \oplus_E \Delta'_C = s'_i$ is provable (cf. direction '$\Rightarrow$'). Consequently, by the induction hypothesis for the transition $[a_{i+1}, \ldots, a_n]$,

$$I', \langle s'_i, \ldots, s'_n \rangle \models \psi'$$

holds. Applying the definition for $\otimes$, the desired conclusion

$$I', \langle s'_0, \ldots, s'_n \rangle \models \varphi' \otimes \psi'$$

follows.

□

With standard reasoning over the immediate consequence operators, it is possible to show that also the semantics of programs are essentially the same. The minimal model of an update program written in the ULTRA syntax is coinciding with the intended interpretation M of the same program (modulo renaming of ":" to $\otimes$ and NOP to $true$) on arbitrary definable update atoms and thus arbitrary update goals φ. $P, s_0, \ldots, s_n \models \varphi$ is defined to hold, if for all models M of P (or alternatively for the least model M of P) $M, \langle s_0, \ldots, s_n \rangle \models \varphi$ holds. Intuitively, this means that the formula φ can cause a sequential transaction going through the states s_0 to s_n. $P, s_0 — s_n \models \varphi$ holds, iff there exists a sequence $s_0, \ldots, s_n$ of states such that $P, s_0, \ldots, s_n \models \varphi$ holds.

Proposition 7.8 Let $s_0 \in \mathcal{S}$ be an arbitrary state. Let P_{UP} be an update program in the ULTRA syntax and P'_{UP} be the same program in the syntax of Transaction Logic. Let φ be a ground update goal in ULTRA and φ' the corresponding goal in Transaction Logic.

Then for arbitrary sequences $s'_0, \ldots, s'_n$ of states $s'_i \in \mathcal{S}$, and arbitrary consistent transitions $\Delta_C \in \mathcal{T}_{Cons}$ with $s_0 \oplus_E \Delta_C = s'_0$ the following holds:

$$P'_{UP}, s'_0, \ldots, s'_n \models \varphi' \iff \begin{array}{l}\text{there exist actions } a_1, \ldots, a_n \in \Sigma \text{ such that :} \\ (\Delta_C, [a_1, \ldots, a_n]) \in M_{UP}[P_{UP}, s_0](\varphi) \\ \text{and } \forall i \in \{1, \ldots, n\} : do(a_i, s'_{i-1}) = s'_i\end{array}$$

Proof: The entailment of a formula φ' by a program P'_{UP} in Transaction Logic obeys the derivation technique of an immediate consequence operator, i.e. $P'_{UP}, s_0, \ldots, s_n \models U$ implies $P'_{UP}, s_0, \ldots, s_n \models p(\vec{t})$ for arbitrary sequences $s_0, \ldots, s_n$ of states, if $U \rightarrow p(\vec{t})$ is a ground instance of a rule in P'_{UP}. Thus, by induction on the ordinal powers of the immediate consequence operators, the assertion can be shown: the interpretations I and I' that arise in every step coincide on all ground formulas. Note that Lemma 7.7 is relevant for the proof. □

Corollary 7.9 Let $s_0, s \in \mathcal{S}$ be arbitrary states. Let P_{UP}, P'_{UP}, φ, and φ' be given as in Proposition 7.8. Then the following holds:

$$P'_{UP}, s_0 — s \models \varphi' \iff \begin{array}{l}\text{there exists a sequence } a_1, \ldots, a_n \text{ of actions } a_i \in \Sigma \text{ such that :} \\ ([\,], [a_1, \ldots, a_n]) \in M_{UP}[P_{UP}, s_0](\varphi) \\ \text{and } s_0 \oplus_E [a_1, \ldots, a_n] = s\end{array}$$

In other words, there is a valid transaction for φ' leading from s_0 to s in the Transaction Logic environment, iff there exists a possible transition $\Delta := [a_1, \ldots, a_n]$ for the query $\leftarrow \varphi$ in the ULTRA environment, such that $s_0 \oplus_E \Delta = s$.

Proof: The proof follows easily from Proposition 7.8 with $\Delta_C := [\,]$. □

Corollary 7.9 formally demonstrates the equivalence of the meanings of a sequential update program in ULTRA and Transaction Logic.

7.3.2 Concurrency Concepts

ULTRA as well as Concurrent Transaction Logic are not restricted to sequential operations. Both languages feature constructs for the concurrent composition of operations. The semantics of these concepts, however, is significantly different. While the concurrency semantics of ULTRA is founded on consistent compositions of locally derived increments, the concurrency semantics of Concurrent Transaction Logic is based on interleaving as known from classical parallel programming languages. In this section we are going to compare both approaches. This will also give some more insights into the ULTRA semantics, which has been defined formally in Section 4.

At the syntactical level, both languages feature a concurrent conjunction to be used to combine two or more subgoals. The concurrent conjunction of ULTRA is denoted by ",", the one of Concurrent Transaction Logic by $|$. The latter language further provides an atomicity operator $\odot$, which is needed to preclude the interleaving of an operation with other operations that are composed by $|$.

Next, we will show the specific elements in the semantics of the different concurrent conjunctions. For the sake of illustration, we assume that two sequential goals φ and ψ are given and each of them specifies three subsequent state changes. Our objective is to explain the meaning of φ, ψ and $\varphi \,|\, \psi$. Dependent on the situation, φ and ψ are considered as goals either in ULTRA or (Concurrent) Transaction Logic.

Recall from Section 4.2 that the semantics of ULTRA formulas is defined in terms of transitions rather than in terms of states. The interpretations $I(\varphi)$ and $I(\psi)$ will contain pairs of consistent transitions, where the first components point to hypothetical current states and the second ones represent new increments. Let (Δ_C, Δ_1) and (Δ_C, Δ_2) be such pairs contained $I(\varphi)$ and $I(\psi)$, respectively. Note that both pairs refer to the same current state $s_{Curr} \in \mathcal{S}$. Δ_1 and Δ_2 describe *local* transitions to hypothetical final states without referring to the intermediate states anymore. A valid transition for the conjunction φ, ψ is the concurrent composition $\Delta_1 \sqcup \Delta_2$, provided that Δ_1 and Δ_2 are conforming with each other. Informally speaking, only the *increments* specified by φ and ψ are merged but not the *states* resulting by an execution of the increments. The merging semantics is visualized in Figure 11, where the upper branch corresponds to the semantics of φ and the lower one to the semantics of ψ. The state transition of φ, ψ from the current state s_{Curr} to the next state s_{Next} is represented by $\Delta := \Delta_1 \sqcup \Delta_2$.

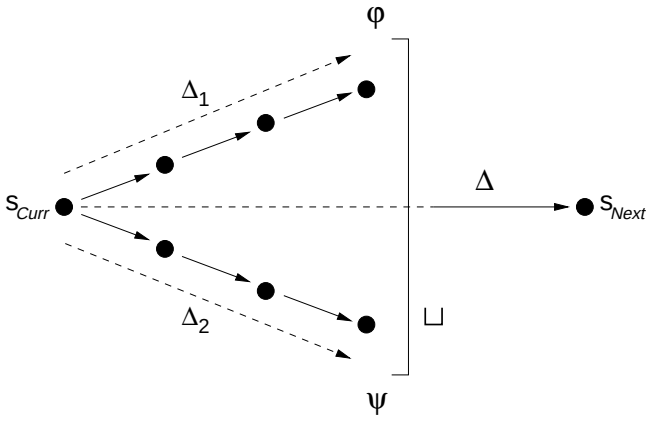

Figure 11: Concurrent conjunction φ, ψ in ULTRA

As described in Section 7.3.1, the semantics of sequential update formulas in Transaction Logic is defined in terms of state paths. However, this notion is too weak to capture interleaving. Thus, in Concurrent Transaction Logic the semantics of state paths is generalized to a semantics of multi-paths. A multi-path $\langle \pi_1, \ldots, \pi_n \rangle$ is a finite sequence of state paths π_i. Each state path π_i represents a transition through a number of explicitly mentioned states, while the state is assumed to change arbitrarily between two subsequent state paths in a multi-path. These changes must be specified by the context in which the multi-path is used. For instance, if a formula φ is entailed by a multi-path $\langle \langle s_0, s_1 \rangle, \langle s_2, s_3 \rangle \rangle$ containing the states $s_0, \ldots, s_3 \in \mathcal{S}$, then it specifies a transition first from s_0 to s_1 and subsequently from s_2 to s_3. Between these two single transitions a *non-specified* state change from s_1 to s_2 is required. To be applicable at the top-level, an interleaving with another operation that specifies the missing transition from s_1 to s_2 must take place.

Two multi-paths can be combined concurrently to a new multi-path such that the elements of both multi-paths occur in the new multi-path respecting the given orders. This combination is non-deterministic and corresponds to a possible interleaving of the multi-paths. A sequential com-

position of multi-paths is defined by the classical concatenation. If the final state of a path π_i in a multi-path coincides with the first state of the next path π_{i+1}, then the multi-path can be reduced by melting π_i and π_{i+1} to one state path. Iterated reduction may lead to a singleton multi-path $\langle\pi\rangle$, which correspond to a state path π.

Concurrent Transaction Logic implicitly defines two semantics of update formulas: an *open* and a *closed* one. The open semantics of a formula is based on multi-paths and thus enables interleaving with other operations that are specified by the context. The closed semantics is based on singleton multi-paths (i.e. state paths) and precludes further interleaving. This semantics is relevant for top-level goals and formulas explicitly closed by the atomicity operator $\odot$ (in the following also called $\odot$-formulas). In a given interpretation I, a formula φ is entailed by a set S of multi-paths. This set is closed under reduction, i.e. if S contains a multi-path $\langle\pi_1,\ldots,\pi_n\rangle$ that reduces (iteratively) to a multi-path $\langle\pi'_1,\ldots,\pi'_m\rangle$ (with $m<n$), then the latter is contained in S, too. To interpret φ at the top-level or to interpret $\odot\varphi$, only singleton multi-paths in S are considered. The concurrent conjunction of $\odot$-formulas leads to arbitrary serializations as known from transaction theory (cf. Section 2.5). However, the semantics does not state how the operations can be processed in parallel. For two $\odot$-formulas $\odot\varphi$ and $\odot\psi$, the semantics of $\odot\varphi \mid \odot\psi$ is equivalent to the semantics of the disjunction $[\odot\varphi\otimes\odot\psi] \vee [\odot\psi\otimes\odot\varphi]$. The interpretation of general formulas that are composed by $\mid$ is based on interleaving. Figure 12 illustrates the semantics of the example goal $\varphi\mid\psi$ for one possible interleaving. If φ is entailed by the upper and ψ by the lower multi-path depicted in part (a), the conjunction $\varphi\mid\psi$ is entailed by the merging of the multi-paths. Provided that the states at the joined positions coincide, the multi-path can be reduced to the singleton multi-path shown in part (b). This multi-path entails $\varphi\mid\psi$, too. Moreover, it represents a valid execution path for $\varphi\mid\psi$ at the top-level and entails the $\odot$-formula $\odot[\varphi\mid\psi]$.

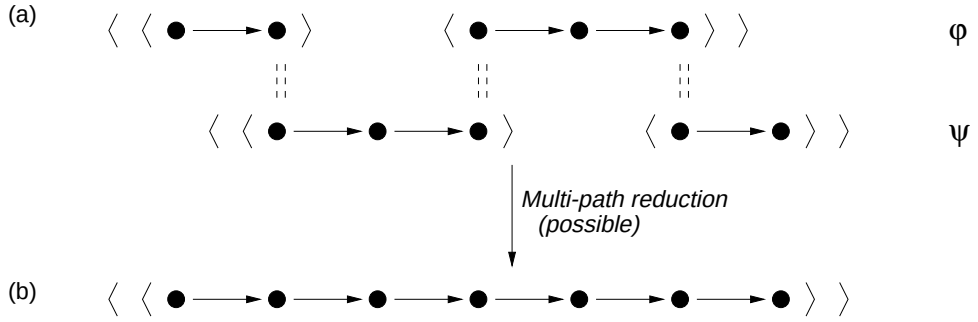

Figure 12: Concurrent conjunction $\varphi\mid\psi$ in Concurrent Transaction Logic

In the remaining of this section we will discuss merits and drawbacks of the different semantics of concurrency. The semantics of ULTRA, which is based on increments rather than on states is compact and easy to understand. Additionally, it is compositional and thus well-suited for verification by formal means, e.g. using an extended version of Hoare's logic [Win93] developed for the verification of sequential programs. The open semantics of Concurrent Transaction Logic can also be used for formal verifications, but the overall meaning of a formula is more complicated than in the ULTRA context, as it has to take arbitrary interleaving with non-specified operations into account. Which state changes an update formula actually implies, can only be seen in the closed semantics, where the restriction is made to state paths. This is also the reason, why top-level update goals are interpreted by the closed semantics. The closed semantics is implementable (see [BK96] for a proof-theoretic semantics), but it is not compositional anymore, since further

interleaving is excluded.

It must be admitted that the modeling power of Concurrent Transaction Logic lies above that of ULTRA. The former enables interleaved operations that may communicate with each other, while the latter focuses on isolated executions. Two ULTRA goals composed by the concurrent conjunction do not see the changes of each other and thus cannot communicate via state access. Only some form of interaction at the logical level is possible by using shared variables. The local change requests are merged at a later time, and the results of the isolated computations can just be checked against the consistency property of the underlying transition system. This corresponds to an optimistic parallel execution. In contrast, what Concurrent Transaction Logic offers is merely the classical parallel programming paradigm with all its benefits and problems. A crucial problem in ULTRA arises only, whenever concurrently composed goals access their intermediate states (see e.g. Figure 11). These states only exist hypothetically and do not depend on the concurrently specified changes. It is questionable under which conditions read access to these states should be allowed. In the ULTRA instances presented in Section 4 we have permitted full access, as the use of the concurrent conjunction is under control of the programmer – as opposed to concurrency arising from independent transactions, which has been treated in Section 5. If this view appears dangerous for other settings, one might strengthen the consistency constraints (see Section 4.5.3 for technical details) such that conformity of transactions implies read-isolation properties as defined in Section 5.2. In this case, concurrent operations can only be successful, if they appear fully isolated. However, this might decrease the profit and usability of the concurrency semantics.

Although deferred materialization is not the final objective for an operational semantics, the ULTRA approach has been designed with such a strategy in mind. It further has been conceived for distributed and pipelined transaction processing architectures. All these operational conditions are in contradiction with the semantics of Concurrent Transaction Logic: In this approach, the operations cannot be hypothetically computed in isolation, as they must synchronize using the intermediate states. Otherwise they would have to take every possible interleaving into account (according to the open semantics), but this is intractable for an operational semantics. If basic operations that are encountered during (top-down) evaluation are processed by an independent component and possibly in combination with other transactions, it is unclear, whether the operational results harmonize with the logical state path semantics: a scheduler, which is allowed to exchange compatible basic operations, will generate other state paths than those the formal semantics describes. In distributed environments, problems can arise from the definition of the semantics over global states. In ULTRA, however, the semantics refers to transitions built from basic operations and thus harmonizes better with commonly used scheduling and logging techniques at the level of operations.

Even though Concurrent Transaction Logic can be regarded as a viable parallel programming language, ULTRA has many advantages due to the compact semantics, the verifiability, and the flexibilities w.r.t. an operational environment. Indeed, the ULTRA semantics of concurrency is peculiar in some points and has a touch of elements known from quantum physics. Note, however, that the paradigm of strict sequential system evolution is not the only one allowed, in particular for distributed and unsynchronized systems. More liberal paradigms can be also viable, as long as their semantics is clearly defined and complex systems can be understood, analyzed, and verified. The bulk quantifier defined in ULTRA, for instance, is based on the special concurrency semantics and can capture the semantics of update statements written in pure SQL (cf. Section 2.1). These SQL statements would have highly non-deterministic and unpredictable meanings, if their semantics was built on an interleaving concept.

7.4 Monadic Programming in Functional Languages

In the area of functional languages, a powerful programming paradigm, called *monadic programming*, has emerged. Essentially, the well-known concept of function application is extended such that its semantics can deal with objects that are not explicitly shown in the syntax. As sketched in [Wad95b], it is easy to build a framework founded on a (polymorphic) abstract data type. An instance of the framework is created by providing a concrete implementation for the abstract data type. This data type must satisfy a few algebraic laws, such that it behaves as a *monad*. During the development of the framework there is no need to provide an implementation of the data type, the framework can even be "tested" using a trivial monad. The instantiation can then be done without modifying the framework itself. This enables programming in a modular and polymorphic style. The principles of programming with monads have been implicitly adopted during the development of the generic ULTRA concept. Probably, under some constraints, the ULTRA concept can be efficiently implemented in a functional language using monadic programming techniques.

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 [Wad95a] 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 corresponds to the ULTRA semantics, which is also (semantically) based on deferred updates. In contrast to languages like Lisp and SML, which consider I/O operations just as side effects of the evaluation, in the monad approach they are modeled as functional results and thus compatible with legal program rewritings and different evaluation techniques established for pure functional languages. 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]. Consequently, one can implement reactive systems in a pure functional style. 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. In particular, ongoing work on the operational model for ULTRA might profit from the results obtained for functional programming. 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.

8 Implementation of the ULTRA Language

In the previous sections, ULTRA has been designed as a logic-based specification language for complex update operations. We have put emphasis on the *semantical* aspects, but we have left out the question of how to implement the language, such that the specified operations can actually be executed as transactions. Only the optimistic transaction processing strategy described in Section 5 can be seen as an operational model, although it is founded on abstract results at the semantical level. In this section, we want to refer to *operational* aspects in more detail. In contrast to the model-theoretic semantics, the operational model strongly depends on the currently chosen instance of the ULTRA framework, because the framework is too general for an efficient implementation and specific properties of the instances have to be taken into account.

The following fundamental settings are possible for the ULTRA language:

- ULTRA can serve as a *pure modeling language* without an operational semantics. In this case, it is possible to specify complex operations and then discuss about them thanks to the formal semantics. However, the operations must be re-implemented, if they are assigned for executable applications.
- *Compiler techniques* can be applied to transform ULTRA programs into evaluable programs of an imperative programming language, e.g. Java with calls to SQL.
- An architecture for the *direct evaluation* of update queries against ULTRA programs can be designed. In addition to a component for syntactical program analysis, a dedicated *run-time system* has to be developed.

In the remaining of this section we will mainly adopt the last point of view. The ULTRA architecture described in Section 8.1 can also be seen as a compiling approach. However, it uses straight-forward transformations, the target language is again a logic programming language, and the transaction processing has to be performed outside the logic programming environment. Thus, we can count this operational model to the last point, too. Note that we can just give some outlines and examples of the processing techniques. The precise and formal development of one or more operational semantics for the ULTRA approach is out of the scope of this thesis. It is rather conceived as one objective of the ULTRA project for the next two years.

8.1 The Two-Phase Strategy based on Deferred Executions

A straight-forward method to execute ULTRA transactions is the two-phase strategy, which has already been in the focus of Section 5. A transaction invoked by a top-level update query is processed in two phases: an *evaluation phase*, where the initial state is kept unchanged and all references to intermediate states are handled by hypothetical reasoning, followed by a *materialization phase*, where the transaction may execute one possible transition for the underlying update query. Multiple independent transactions can be handled using the optimistic protocol presented in Remark 5.17.

Although the two-phase strategy may be applicable in other settings as well (cf. Section 5.5), a natural candidate for this operational model is the database-oriented ULTRA instance of Sections 3.2, 4.3, and 5.4. The deductive reasoning that is necessary to compute the semantics of the IDB

rules can easily be combined with the hypothetical reasoning about intermediate states. Recall that these states are represented by update request sets $\Delta_C \in \mathcal{T}_{Cons}$ w.r.t. the initial EDB instance. The references to these update request sets can be explicitly encoded into the IDB rules. For this purpose, new IDB predicates corresponding to the relevant EDB and IDB predicates of $Pred_{DB}$ are defined. The new predicates have an augmented arity, such that the additional position can accommodate the reference to a hypothetical current state. The hypothetical reasoning is possible, as the semantics of $\oplus_E$ is expressible by simple axioms, which take the initial EDB instance as well as the increments to the hypothetical states into account. This technique resembles the logical characterization of action effects in the situation calculus [Rei95]. Let us give an example of the encoding.

Example 8.1 [Transformation of EDB and IDB] Recall Example 3.17 (see Appendix B for more details) and consider the 3-ary EDB predicate *entry*.

In the extended program, there will be two IDB rules specifying the truth values of EDB atoms over *entry* in hypothetical states. *hyp_entry* must be a new 4-ary IDB predicate.

$$\begin{array}{lcl} hyp\_entry(DeltaC, D, S, ID) & \leftarrow & entry(D, S, ID),\ \mathsf{not\_deleted}(entry(D, S, ID), DeltaC) \\ hyp\_entry(DeltaC, D, S, ID) & \leftarrow & \mathsf{inserted}(entry(D, S, ID), DeltaC) \end{array}$$

While *entry* captures the semantics in the initial database state, *hyp_entry* can deal with arbitrary states represented by $DeltaC$. Note that not_deleted and inserted are built-in predicates for checking absence or presence of update requests in the update request set referred to by the variable $DeltaC$.

Next, have a look at the IDB rules that define the IDB predicate $free$.

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

They lead to the following IDB rules, where hyp_free is a new 4-ary IDB predicate:

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

□

Not only the hypothetical reasoning but also the creation and combination of update request sets according to the semantics of Section 4.2 can be encoded into the logic program. The IDB rules must explicitly specify which pairs of update request sets are contained in the model-theoretic interpretation of an update goal. Technically, the definable update predicates are augmented by two positions to accommodate references to the current state Δ_C and the new update request set Δ, and the transition assignments *Log* and *Upd* as well as the composition constructs $\sqcup$, $\bigsqcup$, and $\oplus$ are realized by built-in predicates. The implementation of the bulk quantifier can be efficiently built on aggregation, since the semantics refers to a collection of DB atoms entailed in a hypothetical state. A crucial problem arises from the fact that the semantics is defined w.r.t. consistent transitions.

Theoretically, the consistency of update request sets would have to be checked within every rule. However, due to the properties required in Section 4.1, update request sets created for update literals are consistent, and the sequential composition of consistent update request sets leads to consistent update request sets. Thus, inconsistent update request sets can occur only in presence of concurrent compositions. Corresponding checks are sufficient to guarantee that all update request set that are created and possibly occur at the Δ_C-position of the new predicates during an evaluation are consistent. The fixpoint semantics of semi-definite logic programs [Llo87] harmonizes with the fixpoint semantics of update programs (see Theorem 4.87). Consequently, the transformed rules will explicitly characterize the update semantics of Section 4 for the original update program. We now show the transformation of an update rule taken from the calendar example.

Example 8.2 [Transformation of Update Rules] The rule

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

defining the operation *do_insert* in Example 3.17 (see also Appendix B) is transformed into the following IDB rule:

$$\begin{aligned} & hyp\_do\_insert(DeltaC, Delta, D, S, L, T) \\ & \quad \leftarrow \ \mathsf{newid}(ID), \\ & \quad\quad hyp\_do\_allocate(DeltaC, Delta1, D, S, L, ID), \\ & \quad\quad \mathsf{collect\_ins}(description(ID,T), Delta2), \\ & \quad\quad \mathsf{conc\_comp}(Delta1, Delta2, Delta),\ \mathsf{consistent}(Delta) \end{aligned}$$

hyp_do_insert and *hyp_do_allocate* are new IDB predicates for the reasoning about the semantics of the definable update predicates *do_insert* and *do_allocate*, respectively. collect_ins, conc_comp, and consistent are static built-in predicates for the implementation of $Upd(INS\ldots)$, $\sqcup$, and $\mathcal{T}_{Cons}$, respectively. □

Some words should follow about the evaluation task. The rules generated from the EDB/IDB specifications and those generated from the update program can be evaluated within a single logic program. IDB rules containing negation require adequate evaluation methods (cf. Section 2.3). However, all the evaluation methods for the well-founded semantics can be used for semi-definite rules and thus for the transformed update program as well. Unfortunately, the transformed programs are not range-restricted, but as soon as a (transformed) query is provided, the rules can be computed in a bottom-up fashion after applying the magic set transformation [BR91]. This yields the possible transitions for the given query according to Definition 5.2. The conventional sideways information passing strategy fits with the natural adornments of the built-in predicates, such that an efficient evaluation is possible. The semantics of a bulk quantifier can be correctly computed by an aggregation engine, if the sets of result tuples for the condition atom are always finite. Otherwise there might be semantical problems, but the evaluation would not terminate anyway.

A prototype system based on the methods described above has been implemented on top of the deductive database system *LOLA* [FSS91, ZF97]. As the transaction processing is not integrated into the *LOLA* deduction engine, the *LOLA* system can easily be replaced by other systems, e.g. XSB [RSS+97]. The only requirement is an interface that allows read access to the database

system keeping the EDB. Soon, we will be able to use a new version of *LOLA*, which is based on a significantly improved evaluation concept [ZBF97, ZF99] for the well-founded semantics. In the context of two subsequent diploma theses [Köh96, Rim98], a compiler for IDB programs and update programs was designed and implemented. The compiler analyzes the source programs and produces IDB rules as illustrated in Examples 8.1 and 8.2. Also update queries can be transformed. Due to a grammatic-driven implementation supported by JLex and JavaCup (see [App98] for details), the compiler can easily be tailored to a more user-friendly ULTRA syntax, and syntactical extensions in case of refinements of the ULTRA instance are feasible.

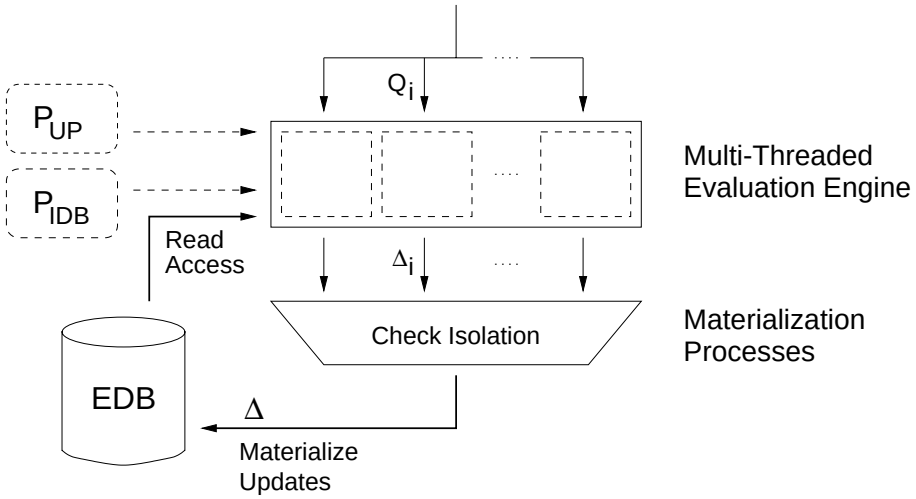

Figure 13: Architecture of the first ULTRA prototype

Figure 13 illustrates the system architecture for the evaluation and materialization of transactions invoked by top-level update queries. Let us tacitly assume that the programs and the queries are compiled and that a magic set transformation is applied for every query. Transactions for the given queries now are processed in two phases according to the optimistic transaction protocol described in Remark 5.17. Multiple transactions can be evaluated independently by multiple evaluation threads. Synchronization between different threads is not necessary, as all queries are evaluated w.r.t. the same physical database state and no updates on the EDB are actually performed. Consequently, no blocking can arise, provided that the underlying database system allows shared read access. The independently derived update request sets must be checked against the read-isolation property: a possible transition taken as a solution must be read-isolated from all solutions collected so far, which are going to be materialized at first. Since the read-isolation property is defined at the predicate level (see Section 5.4 for more details), it is easy to check. Due to the set-oriented bottom-up evaluation, all possible transitions for a query are generated. This allows the system to try the choice of another result, if the isolation checks fail. In the common materialization phase, all existing solutions are subsequently materialized as write-only transactions on the database system. For every successful materialization, a commit message is returned to the top-level. All transactions that have lead to a definite failure in any of the phases are aborted.

We assume that the extensional database system does not allow *local* transactions [BHG87], i.e. transactions not executed under control of the ULTRA system. Such local transactions can confuse the evaluation threads or invalidate the computed results. Moreover, local transactions can lead to deadlocks, which are precluded in the optimistic transaction processing strategy.

8.2 Immediate Executions in a Nested Transaction Environment

In the ULTRA approach, the execution of a transaction consists of two types of processing: the *evaluation* of update goals according to the logical semantics (for binding variables and generating basic operation requests) and the *execution* of selected basic operations (*materialization*). The model-theoretic semantics describes solutions in terms of transitions but does not state anything about the materialization task. Neither the choice strategy, nor the materialization time is restricted. Consequently, an obvious goal is to do the materialization in parallel with the logical evaluation. Under some conditions mentioned below, this goal can be achieved, such that the materialization can proceed as soon as possible.

As also broadly discussed in [FWF00, WFF98a], the strategy considered in Section 8.1, which strictly separates evaluation and materialization, has several drawbacks.

- There is a need for hypothetical reasoning when referring to intermediate states: as the operations leading to an intermediate state are known but not carried out yet, their effects on the state are not visible and thus must be computed by a reasoning component. An axiomatization of the observable effects is necessary to enable such hypothetical reasoning at the logical level. Unfortunately, this is only tractable for simple basic operations like insertions and deletions. When operations are permitted, whose semantics is not fully specified or whose semantics is too complicated to reason about, the hypothetical reasoning becomes impossible.
- A second practical problem results from performing a transaction in two strictly separated phases (evaluation and materialization). Such a system does not show a continuous behaviour during the evaluation and thus is not suitable to be extended by e.g. interactive components. It merely implements a batch mode, where actions are collected to be performed later. For instance, let a multimedia session be modeled using the ULTRA language. It is obviously not sensible that each display action is deferred to the end of the session. It is only natural that logical reasoning and physical changes (actions) are interleaved.
- The standard bottom-up evaluation as proposed for the ULTRA semantics always computes *all* possible transitions in the evaluation phase. Especially in presence of non-deterministic specifications and much hypothetical reasoning this may lead to a lot of unnecessary work, and even small examples may not be tractable anymore.

To solve these problems we use a top-down-oriented evaluation strategy similar to the well-known SLD resolution of Prolog [MW88a]. Our operational model features the immediate execution of operations in combination with backtracking, recovery by compensation [KLS98], and nested transactions [BBG89, Mos85, WS92]. It is conceived w.r.t. the second ULTRA instance, which has been presented in Sections 3.3 and 4.4. During the evaluation of update queries within a top-level transaction, requests for basic operations are not collected for later execution, but executed immediately using database techniques. These techniques ensure that actions can be undone in the failure case and concurrent actions are scheduled according to isolation conditions. This enables us to guarantee the ACID properties (see Section 2.5) for the top-level transactions. Since hypothetical states are physically reached, there is no need for hypothetical reasoning. Evaluating queries in a top-down fashion results in a resolution tree, which is mapped onto a nested transaction tree (see Figure 14). Subtransactions are used as rollback spheres to implement a backtracking that fits with the logical semantics. If a branch in the resolution tree fails, the actions performed along this

branch will be undone up to the last choice point. Then the next choice can be tried or the rollback can be cascaded. As we do not deal only with simple database operations and want to extend the operational model to open nested transactions (see below), a classical rollback by restoring a backup of the old state is not sufficient. Instead we use the more general concept of compensation [KLS98]: for every operation, a corresponding undo operation must exist, such that the execution of both operations leaves the state (semantically) unchanged.

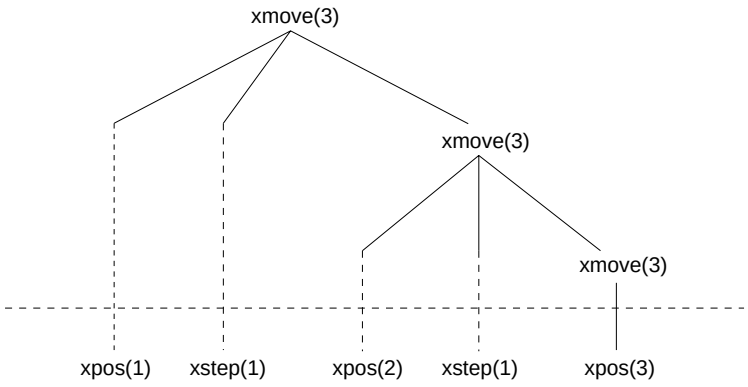

Figure 14: Resolution tree corresponding to a nested transaction tree

Using subtransactions only to aid backtracking is not satisfying. Nested transactions were invented in the database community to, among other reasons, increase the possible amount of concurrency between transactions by using the additional semantical knowledge of complex operations. This can of course also be done in the case of logical update languages as ULTRA. For this purpose, a scheduler for *open* nested transactions is needed.

An important aspect in the treatment of open nested transactions is the *meta information* that is required to schedule operations and to compensate already committed subtransactions. The need for meta information arises from the fact that the operations being in the focus of the scheduler are not only the predefined basic operations but also programmed complex operations. In several cases, meta information can be deduced automatically, but the results are either trivial or they require sophisticated reasoning techniques at the semantical level. Thus, we adopt the point of view that the relevant meta information is provided in form of declarations together with the update program. On the one hand, compatibility information must be declared to allow correct scheduling, on the other hand, compensation information is needed for recovery. The compensating operations can be specified in terms of update rules, this leads to extended update programs.

The architecture currently under construction is shown in Figure 15. An ULTRA *evaluation engine* accepts top-level update queries and resolves them w.r.t. a given update program which is stored together with meta information in a repository. The resolution-based evaluation is not instantly performed down to the basic operations. Instead, basic operations as well as complex operations generated in a resolution step are given to a *scheduler*, which decides about their further processing in a subtransaction. The scheduler orders the operations from various transactions in such a way that they appear to run in isolation. It is based on a classical synchronization protocol and accesses the meta information about compatibility. The scheduler forwards basic operations to the *data manager* for execution on the underlying *external systems*. These can be database systems, which persistently store DB relations, as well as interfaces to external hardware and software. The data manager returns retrieval data (for DB atoms) and additional data needed for

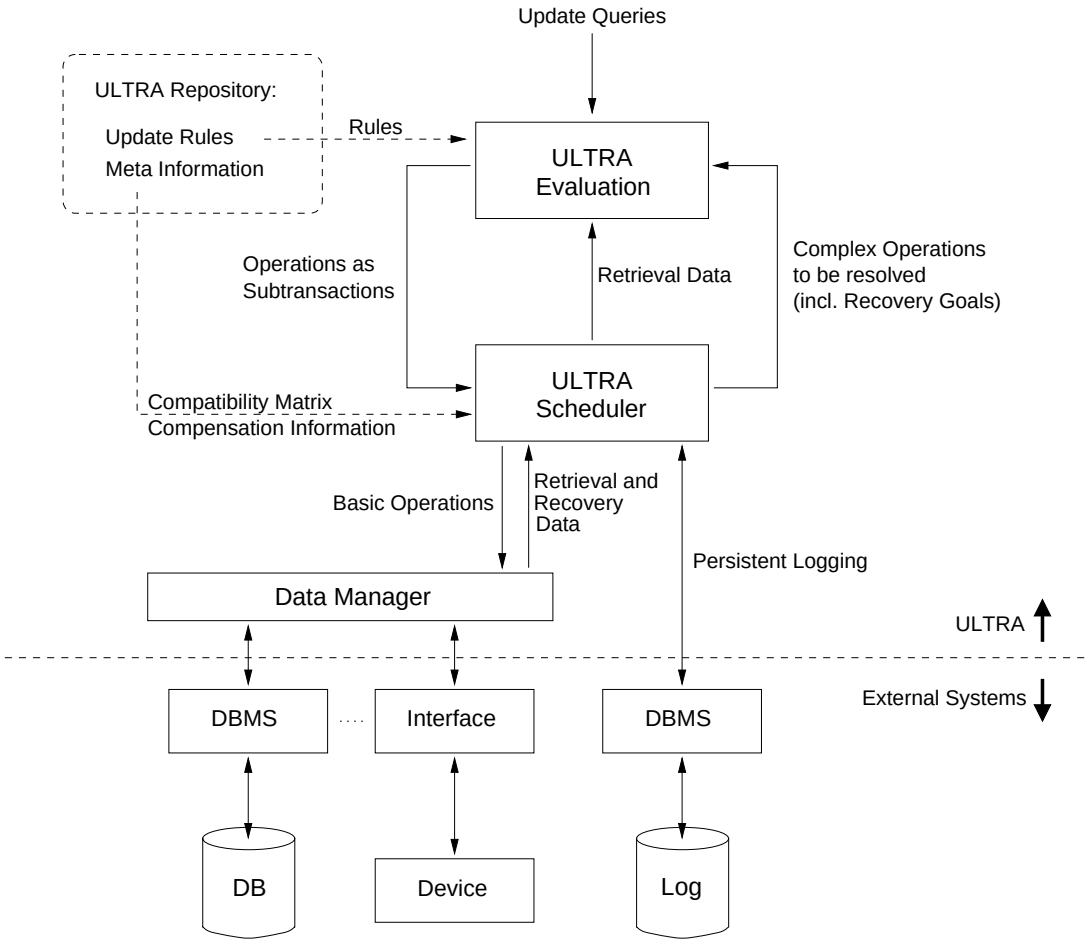

Figure 15: Architecture of the currently developed prototype

compensation of operations. Complex operations that have passed the scheduler are returned to the evaluation engine to be resolved in a new thread. All operations of the scheduler are logged in a persistent device. The log is considered backwards to undo already committed operations. While the compensation of a basic operation is assumed to be done by the data manager, a complex operation is compensated by executing another complex operation: the compensating update goal is extracted from the meta information and sent to the evaluation engine.

To this time, the scheduler, the data manager, and some communication components have been implemented in Java. The components are operable for closed nested transactions, but the extensions to open nested transactions appears feasible. For test purposes, a top-down resolution engine for update goals has been written in Common Lisp. It can deal with arbitrary basic operations and features a connection to the the *LOLA* prototype [FSS91, ZF97], such that additional IDB rules can be evaluated. However, the ULTRA evaluation engine is restricted to sequential goals and a single evaluation thread. We have currently started with the design of a new evaluation engine in Java. However, fundamental work about an operational semantics for ULTRA is still necessary to implement a correct and efficient evaluation strategy.

8.3 Outlook

A central point of ongoing work is the conception and design of a new ULTRA evaluation engine for the architecture shown in Figure 15. For this purpose, we are going to develop a proof-theoretic semantics for the ULTRA language. The theoretical results will lead to an implementable evaluation strategy for update goals. The central problem lies in the treatment of the concurrent conjunction and the bulk quantifier, whose semantics essentially rely on the merging of deferred transitions and do not directly harmonize with immediate materializations. In this context, it is probably necessary to tune the instances of the ULTRA framework described in this thesis or to specify restricted program classes for which an effective evaluation is possible. Presently, 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. A hybrid approach may also help to deal with the special notion of concurrency in ULTRA: concurrently composed and set-oriented operations could be locally handled by a deferred approach, and for sequential operations (at the top-level) global checkpoints could be materialized.

As mentioned above, the use of open nested transactions requires a lot of meta information that is specified simultaneously with the update rules. Thus, another objective within the ULTRA project is to provide repository tools and reasoning techniques for meta information. The programmer should be supported at the declaration task, and the system should be able to propose rudimentary defaults that can be refined later.

8.4 Example Applications

To conclude this section, we mention some practical experiences with the ULTRA language and the prototype implementations.

The personal calendar, which has been chosen as one of the running examples, was the first non-trivial application for the ULTRA prototype based on the two-phase strategy (see Section 8.1). The operations implemented in terms of update rules were motivated by a classical implementation of a calendar tool, which was the task of two diploma theses [Bay98, Hir97]. The application was written in Java, but the database operations are carried out by SQL statements. A comparison between the collection of extensive SQL statements and the corresponding ULTRA rules revealed the advantages of a clear and compact logic language: the ULTRA implementation turned out to be better understandable and communicatable. Even the programmer had some difficulties to debug his own SQL code, when corrections or modifications were requested.

With the objective to test the modeling power and practical relevance of the ULTRA language, we designed a workflow engine based on a petri-net representation of workflows. The update rules define dynamic operations at the level of workflow instances, e.g. creation of a new instance, progress of an instance, assignment of resources, notifications, etc. Several solutions taken from a diploma thesis [Mär98] (among them refinement of workflows, instance variables, locking of resources and variables, computation of decision parameters, and role hierarchy) could be condensed to an ULTRA program within a few days. The programming style was compact and foreseeable, local modifications and tests of variants appeared as uncritical. The constructs of the ULTRA language turned out to be sufficient for the tasks, in particular, the bulk quantifier could often be used for set-oriented operations (e.g. removing of all tokens enabling a transition, locking of all relevant

variables, notification of all participants of a workflow instance, etc.). Moreover, the combination of update programs and deductive reasoning was helpful: the role hierarchy and its exceptions could be expressed compact and comprehensible using IDB rules. To make the system operable, a simple graphical user interface was added. Multiple users can start a local copy of this interface, and then communicate with the global workflow engine. In effect, predefined update queries are accepted and performed by the two-phase prototype described in Section 8.1. In addition to the classical EDB relations that store the dynamic data, some EDB relations are considered as message containers. The messages are sent to the local user interfaces during the materialization phase. This way, the users get requested information or notifications about the progress of the workflow system.

The robot example listed in Appendix C (in a slightly modified version, because the current prototype can deal with sequential programs only) was our key application to test the second prototype, which has been discussed in Section 8.2. In contrast to the applications above, the ULTRA architecture must deal with an external device different from a database system. The robot is simulated and visualized by a software component written in Java as a practical course, the communication with the data manager of the ULTRA prototype is based on a standard protocol. The robot software, which behaves as a black box, performs the basic operations (*xstep*, *ystep*, *pickup*, and *putdown*) and returns state information (*xpos*, *ypos*, and *empty*). As the operations of the robot are backtrackable (e.g. $xstep(-1)$ can be compensated by $xstep(1)$ and vice versa), it is possible to execute complex robot operations like *move_block* (see Appendix C) as atomic transactions. It should be noted that the robot software is constructed modularly and the visualization component can thus easily be replaced or complemented by a hardware robot having the same functionality.

9 Conclusion

In this thesis we have developed the rule-based update language ULTRA, which serves for the specification of complex update operations at a high logical level. The language and its semantical foundations have been formulated as a framework, such that instantiations can be defined for specific environments and applications. ULTRA allows the modular construction of complex operations with the possibility of reuse. The constructed update programs have a unique meaning, which has been defined as a model-theoretic semantics and also has a fixpoint characterization. This distinguishes ULTRA from other practical extensions of database languages that only have operational meanings but no overall semantics. Moreover, the logical semantics of ULTRA can be seen as declarative and compositional. In contrast to many other approaches that define dynamics in terms of (sequences of) states, the ULTRA semantics is based on the concept of deferred updates. This allows us to define a comprehensible and verifiable notion of concurrency. Furthermore, we claim that the semantics dealing rather with increments than with global states is better applicable for distributed architectures. However, the operational aspects have only been discussed as a side-issue. Referring to a fixed initial state, the minimal model of a program just assigns possible transitions to update queries. For each query, one of these transitions may be selected and materialized in order to complete the corresponding transaction.

As mentioned above, ULTRA has been defined as a framework. The generic language abstracts from particular basic operations, and its semantics leaves the concrete notion of states and transition objects aside. When an implementable instance of the framework is created, the missing objects must be specified, and a collection of required preconditions has to be proved. Then, all properties that we have shown for the ULTRA framework, in particular those concerning the model-theoretic semantics of update programs, also hold for the instance. For the sake of illustration, we have presented two essentially different instantiations of the ULTRA framework: one tailored to logic databases, the other one to arbitrary external operations. While the semantics of the former instance is based on rather simple update request sets and the integration with the classical deductive rules stands in the foreground, the semantics of the latter instance is based on more complex data structures, namely partially ordered multi-sets [Pra86]. Note that these instances can be modified, extended, or composed in order to meet special requirements. We have sketched some techniques for composing or constraining valid instances without proving the required properties from scratch.

Another section has been devoted to optimistic transaction processing. In this setting, we assume that the materialization of updates is strictly deferred and the whole evaluation according to the logical semantics is done by hypothetical reasoning. Since read/write conflicts between concurrently processed transactions cannot be detected during the evaluation phases, we have presented sufficient conditions to enable a detection at materialization time. The transitions representing the deferred operations must include enough information about the read access that may have occurred during the evaluation. In this case, a read-isolation relation that entails the absence of read/write conflicts can be defined at the level of deferred transitions. The formal results have been exploited for an optimistic transaction protocol.

The development of the ULTRA language has been motivated by some example applications. Further, a lot of related work in the context of logic programming and transaction theory has been considered. The merits of most approaches for the specification of update operations by logical means can also be found in the ULTRA approach. We have presented a more detailed comparison of ULTRA with view update concepts and the most prominent rule-based update language Transaction Logic [BK94] and its extension w.r.t. concurrency [BK96]. The obvious limits of implicit view

updates have been discussed, and the sequential version of Transaction Logic has been formally simulated by the second ULTRA instance presented in this thesis. Only the concurrency concepts of Concurrent Transaction Logic are beyond those of ULTRA. Nevertheless, we have listed some good arguments that justify the ULTRA semantics: these mainly concern compositionality, verifiability, and implementation aspects.

At the end of this thesis, we would like to add some remarks about the usability of the ULTRA language. Due to the genericity of the concepts and the definition of ULTRA as a framework, the language is not restricted to logic databases. Furthermore, the logic-based formalism is neither platform- nor application-dependent and especially worthy in heterogeneous environments. The concurrency semantics within a transaction is well-suited for distributed systems, in particular, when the concurrent sub-operations are isolated or information that is concurrently accessed keeps unchanged. So we can identify the following possibilities for the use of ULTRA in practice:

- Transactional database applications
- Multi-database applications
- Databases in the world-wide web
- Databases and external actions
- Databases and user interfaces
- Data migration and data warehousing
- Applications outside the database world

The major topic of ongoing work is the formal development of an implementable operational semantics, which has been taken out of the scope of this thesis. Here we just have described some ideas and sketched the architectures presently under construction. The next steps comprise the development of a proof-theoretic semantics for the ULTRA language, i.e. for a sufficiently general instance of the framework. A fine tuning of the instances presented in this thesis may be necessary, such that the concurrency semantics of ULTRA harmonizes with the nested transaction models chosen for the evaluation. In the context of open nested transactions, the problems about meta information must be recognized and solved. We aim at the development of intelligent tools that support the programmer. Topics for future work are optimization issues and methods to reason about update programs. The clear and abstract semantics with its declarative properties forms a suitable basis to start with these investigations.

References

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

[ABFS97] G. Alonso, S. Blott, A. Fessler, and H. J. Schek. Correctness and parallelism in composite systems. In *Proc. 16th ACM Symp. on Principles of Database Systems*, pages 197–208, 1997.

[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.

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

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

[App98] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[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.

[Bay98] T. Bayer. Entwurf und Implementierung eines verteilten Kalendermanagementsystems in Java. Diploma thesis, Universität Passau (FMI), 1998.

[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.

[BC93] F. Benhamou and A. Colmerauer. *Constraint Logic Programming: Selected Research.* MIT Press, 1993.

[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.

[BK98] 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.

[CM93] S. Ceri and R. Manthey. Consolidated specification of Chimera (CM and CL). Technical Report IDEA.DE.2P.006, IDEA Esprit Project 6333, 1993.

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

[CWH99] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial Continued.* Addison-Wesley Publishing Company, 1999.

[Das92] S. K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley Publishing Company, 1992.

[DBC96] U. Dayal, A. P. Buchmann, and S. Chakravarthy. The HiPAC project. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 177–206. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

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

[DSW94] A. Deacon, H. J. Schek, and G. Weikum. Semantics-based multilevel transaction management in federated systems. In *Proc. 10th Int. Conf. on Data Engineering (ICDE '94), Houston, Texas*, pages 452–461, 1994.

[EK89] K. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In *Proc. 6th Int. Conf. on Logic Programming (ICLP '89), Lisbon, Portugal*, pages 234–254, 1989.

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

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[FLMW90] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41(1):65–156, 1990.

[FSMZ95] B. Freitag, B. Steffen, T. Margaria, and U. Zukowski. An approach to intelligent software library management. In *Proc. 4th Int. Conf. on Database Systems For Advanced Applications, Singapore*, pages 71–78, 1995.

[FSS91] B. Freitag, H. Schütz, and G. Specht. *LOLA* – a logic language for deductive databases and its implementation. In *Proc. 2nd Int. Symp. on Database Systems For Advanced Applications, Tokyo, Japan*, pages 216–225, 1991.

[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.

[GHJV98] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1998.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

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

[GSZ95] S. Greco, D. Saccà, and C. Zaniolo. DATALOG queries with stratified negation and choice: from P to D^P. In *Proc. 5th Int. Conf. on Database Theory (ICDT '95), Prague, Czech Republic*, pages 82–96, 1995.

[HAD97] U. Halici, B. Arpinar, and A. Dogac. Serializability of nested transactions in multidatabases. In *Proc. 6th Int. Conf. on Database Theory (ICDT '97), Delphi, Greece*, pages 321–335, 1997.

[Här84] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9:111–120, 1984.

[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.

[Hir97] P. Hirschenauer. Datenbankgestützte Terminplanung und -verwaltung. Diploma thesis, Universität Passau (FMI), 1997.

[IS96] K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.

[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.

[KLS98] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, pages 444–465. Prentice-Hall, 1998.

[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.

[Köh96] M. Köhrmann. Realisierung komplexer Updates in deduktiven Datenbanken. Diploma thesis, Universität Passau (FMI), 1996.

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

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[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.

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

[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.

[Mär98] W. Märkl. Workflow-Management für die Projektabwicklung im Baugewerbe. Diploma thesis, Universität Passau (FMI), 1998.

[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.

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

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

[Rim98] B. Rimmel. Entwurf und Realisierung einer Compilerarchitektur für die Datenbank-Update-Sprache ULTRA. Diploma thesis, Universität Passau (FMI), 1998.

[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.

[RU95] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[Sie96] J. Siegel. CORBA *Fundamentals and Programming.* John Wiley and Sons, New York, 1996.

[Som96] I. Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, 5th edition, 1996.

[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.

[Tho98] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, 1998.

[TU95] E. Teniente and T. Urpí. A common framework for classifying and specifying deductive database updating problems. In *Proc. Int. Conf. on Data Engineering (ICDE '95), Taipei, Taiwan*, pages 173–182, 1995.

[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.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, 1991.

[Wex89] J. Wexler. *Concurrent Programming in* Occam 2. Ellis Horwood Ltd., Chichester, UK, 1989.

[WF96] C. A. Wichert and B. Freitag. Logical specification of bulk updates and sequential updates. In *Proc. 4th Workshop on Deductive Databases and Logic Programming (DDLP '96) in conj. with JICSLP '96, Bonn, Germany*, pages 79–94, 1996.

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

[WFF98a] C. A. Wichert, 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.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995.

[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.

[ZBF97] U. Zukowski, S. Brass, and B. Freitag. Improving the alternating fixpoint: The transformation approach. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 40–59. Springer-Verlag, Berlin, Germany, 1997.

[ZF96] U. Zukowski and B. Freitag. Adding flexibility to query evaluation for modularly stratified databases. In *Proc. Joint Int. Conf. and Symp. on Logic Programming (JICSLP '96), Bonn, Germany*, 1996.

[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.

[ZF99] U. Zukowski and B. Freitag. Well-founded semantics by transformation: The non-ground case. In *Proc. Int. Conf. on Logic Programming (ICLP '99), Las Cruces, New Mexico*, pages 456–470, 1999.

Appendix A

Here we show the complete introductory example.

EDB Schema and Instance (DB_0):

$store(Item, Price, Amount)$ contains the price $Price$ and the current stock $Amount$ of a transport item $Item$.

$journal(Item, Amount)$ records the modification $Amount$ of the current stock of a transport item $Item$. (In this simplified example, *journal* should be considered as a multi-set or list.)

store	*Item*	*Price*	*Amount*
	box	5	2
	barrel	20	13
	bucket	8	5

journal	*Item*	*Amount*
	box	−1
	barrel	−1

IDB Program (P_{IDB}):

$low(Item)$ determines whether a transport item $Item$ is low on stock.

$$low(I) \leftarrow store(I, P, A),\ A < 10$$

Basic Update Predicates:

$INS\ldots$ insertion into an EDB relation.

$DEL\ldots$ deletion from an EDB relation.

$send\_mail\_order(Item, Amount)$ invokes the e-commerce system to order $Amount$ pieces of transport item $Item$.

Update Program (P_{UP}):

order_low orders 20 pieces of each transport item with a low stock.

$$order\_low \leftarrow \#\ I\ [\ low(I) \mapsto order(I, 20)\ ]$$

$order(Item, Amount)$ orders $Amount$ pieces of transport item $Item$.

$$\begin{aligned} order(I, A) \leftarrow\ & store(I, P, A0),\ A1 = A0 + A, \\ & DEL\ store(I, P, A0),\ INS\ store(I, P, A1), \\ & INS\ journal(I, A),\ send\_mail\_order(I, A) \end{aligned}$$

$deliver(Item)$ books the delivery of one piece of transport item $Item$ to a worker.

$$\begin{aligned} deliver(I) \leftarrow\ & store(I, P, A0),\ A0 > 0,\ A1 = A0 - 1, \\ & DEL\ store(I, P, A0),\ INS\ store(I, P, A1), \\ & INS\ journal(I, -1) \end{aligned}$$

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 $Slot$.

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

$duration\_of(ID, L)$ 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.)

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

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

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

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.

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

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

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

$do\_move(ID, Day, Slot)$ 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.

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

Appendix C

Here we show the complete robot example.

DB Predicates:

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

$xpos(X)$ determines the x-position X of the robot on the grid.

$ypos(Y)$ determines the y-position Y of the robot on the grid.

Basic Update Predicates:

$xstep(D)$ moves the robot by one step in x-direction. (D can be either -1 or 1.)

$ystep(D)$ moves the robot by one step in y-direction. (D can be either -1 or 1.)

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

Update Program (P_{UP}):

$xmove(X)$ moves the robot in x-direction to the x-coordinate X.

$$
\begin{array}{lcl}
xmove(X) & \leftarrow & xpos(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X < X0 \ : \ xstep(-1) \ : \ xmove(X) \\
xmove(X) & \leftarrow & xpos(X0) \ : \ X > X0 \ : \ xstep(1) \ : \ xmove(X)
\end{array}
$$

$ymove(Y)$ moves the robot in y-direction to the y-coordinate Y.

$$
\begin{array}{lcl}
ymove(Y) & \leftarrow & ypos(Y) \\
ymove(Y) & \leftarrow & ypos(Y0) \ : \ Y < Y0 \ : \ ystep(-1) \ : \ ymove(Y) \\
ymove(Y) & \leftarrow & ypos(Y0) \ : \ Y > Y0 \ : \ ystep(1) \ : \ ymove(Y)
\end{array}
$$

$move(X, Y)$ moves the robot to position (X, Y) on the grid.

$$
move(X, Y) \ \leftarrow \ xmove(X), \ ymove(Y)
$$

pickup_at_position(X, Y) moves the empty robot to position (X, Y) and picks up a block (idle movement not successful).

$$pickup\_at\_position(X, Y) \leftarrow [\ empty,\ move(X, Y)\ ] : pickup : NOT\ empty$$

putdown_at_position(X, Y) moves the non-empty robot to position (X, Y) and puts down the grabbed block (idle movement not successful).

$$putdown\_at\_position(X, Y) \leftarrow [\ NOT\ empty,\ move(X, Y)\ ] : putdown : empty$$

move_block$(X1, Y1, X2, Y2)$ moves a block from position $(X1, Y1)$ to position $(X2, Y2)$. The predicate fails, if actually no proper movement of a block is performed.

$$move\_block(X1, Y1, X2, Y2) \leftarrow pickup\_at\_position(X1, Y1) : putdown\_at\_position(X2, Y2)$$

test_position(X, Y) tests whether there is a block lying at position (X, Y) while the blocks world is left unchanged. The robot must be empty before the test.

$$test\_position(X, Y) \leftarrow [\ empty,\ move(X, Y)\ ] : pickup : NOT\ empty : putdown : empty$$