On the Compositionality and Transactional Execution of Electronic Services

Carl-Alexander Wichert
msg systeme gmbh
Dr. Hans-Kapfinger-Str. 30
94032 Passau, Germany
e-mail: Carl-Alexander_Wichert@msg.de
Tel: +49 (0)851 56069 32   Fax: +49 (0)851 56069 13

Alfred Fent
Burkhard Freitag
Universität Passau
Fakultät für Mathematik und Informatik
94030 Passau, Germany
e-mail: {fent,freitag}@fmi.uni-passau.de
Tel: +49 (0)851 509 3019   Fax: +49 (0)851 509 3002

Universität Passau
Technical Report MIP-0010

Abstract

Electronic commerce plays a more and more important role in today's economy. An increasing number of firms sell their products online, and portals bundle the services of other firms and offer them to the user thereby acting as one single point of contact. An important aspect in this trend is the composition of various services to form value-added offerings. The new, complex services must be easy to specify and understand and must be executed transactionally. Yet, this is only partly addressed by the technology in use today.

The rule-based update language ULTRA covers the whole range from specification to execution and therefore is well-suited to overcome these shortcomings. It provides the necessary features, among them a high-level semantics and compositionality, and implicitly guarantees transactional execution.

After an overview of the general ULTRA framework we present an instantiation which can be used to combine an arbitrary set of basic services to obtain new complex services. Among others, the instantiation features a specialized data structure for the internal representation of composed services that can be used for concurrency control and recovery as well. We then develop a suitable evaluation model for this instantiation. It is based on the well-known notion of nested transactions, uses compensation for roll-back and can handle sequential as well as concurrent services. It turns out that the proposed evaluation model only requires that compensation of the basic services is supported and that they take part in an atomic commit protocol.

1 Introduction

Electronic commerce plays a more and more important role in todays economy. An increasing number of firms sell their products online in all kinds of online shops, in business-to-customer (B2C) as well as business-to-business (B2B) applications. Moreover, new kinds of business arise with the advent of portals – sites that bundle the services of other firms and offer them to the user acting as one single point of contact. This trend became even stronger with the introduction of so called “mega-portals”, which not only bundle individual services, but a whole set of other portal sites.

Obviously, an important aspect in this trend is the composition of the various services to form value-added offerings. This is essentially what a portal does: Either, similar services from various service providers which differ e.g. in the actual interface are encapsulated and offered through the single interface of the portal. WWW meta-search engines which send a user’s search request to different other search engines are an example of this “horizontal integration”. An alternative application of service composition is the reuse of existing services in the definition of a new complex service. An example for this “vertical integration” can also be found within WWW search engines: Some offer not only the service to search the web, but also to translate the search results into another language.

These examples of service integration have in common that there is no change involved on the side of the service provider, i.e. neither a search engine nor a translation service actually change a web side. Yet, more and more services already offered on the web *do* involve changes and real-world actions, as for instance any online shop does. When such existing services are composed into new ones, it is essential that they behave like transactions in databases, i.e. that they obey the ACID postulates [BN97]. Yet, currently the issue of executing complex services transactionally is only poorly addressed, and most real-world examples of complex services spanning several providers either implement transactionality only in parts, or ignore it at all. This leads to the situation that the results of the complex services are well-defined if everything worked as it was expected. However, if errors occur during the execution the outcome of the complex service is more or less undefined.

One reason for the lack of transactionality is that most of the services are only “specified” operationally, i.e. there is no formal specification of what a complex service should do, but only its implementation. Therefore, only if the programmer did his or her job with transactionality explicitly in mind, the resulting service may obey the ACID postulates. Moreover, the implementations may be – depending on the programming language used – hard to understand, maintain, and debug. Consequently, changes, which are necessary quite often in a dynamic medium as the web, may lead to new, unforseen side effects.

Another reason is that often it is not known which properties are required for an external service to be reused within transactions, and which properties a service actually has. Ideally, it should be possible to obtain the documentation of a service and then know whether and how it can be integrated into a higher-level transaction forming a complex service.

The ULTRA framework [Wic00, WFF98, WFF00] can help in this situation. It features a clear, high-level semantics for execution of arbitrary basic operations, it is compositional, and it guarantees transactional execution. The semantics of the language is captured within a flexible framework that can be instantiated for arbitrary external services. However, still the definition of a transition system and some mathematical proofs are required. In this paper we present an instantiation of the general ULTRA semantics that is already tailored to handling calls to external operations. Given a service specification in the rule-based ULTRA language, the ULTRA instance presented here is able to derive which external services have to be executed. To capture the ordering information of the service calls, e.g. when two services are composed sequentially, the instantiation uses the specialized data structure of partially ordered multisets (pomsets) to collect the necessary service calls.

As ULTRA defines the semantics of complex actions by simple, natural compositions of the semantics of the basic services, compositionality is achieved. This is even valid for concurrently composed actions, whose semantics does not rely on some form of interleaving parallelism, but on a common starting and ending state.

To actually use the rule-based language for the specification of value-added electronic services a model for the execution of rules is needed as well. We sketch an evaluation model using a variant of a top-down execution strategy. This strategy gives the correct results, i.e. it executes the actions that are also calculated by the formal semantics. As the semantics of the external actions themselves is not completely specified but only given through an execution function for the action, we cannot reason about what state changes are caused by the actions. Instead, we have to execute them immediately to be able to observe the state change. These immediate actions can be undone in our model during logical backtracking by adopting an open nested transaction system [BBG89, Mos85, WS92] and compensation [KLS98] for recovery. The scheduler that is responsible for transactional execution can also use the pomset data structure, which is used to represent the services on the logical side, for its log of executed actions. So, the two extremes of representing actions required by the semantics and logging actions actually executed meet in one single data structure.

Finally, we will identify the properties the actual calls to external services must have, especially concerning transactional behaviour. It turns out that the basic requirement to the external calls is that they can be undone later, and that the external system can take part in an atomic commit protocol like 2PC to ensure that the scheduler log always is consistent with the executed services. Therefore, any service that exhibits these two properties can be used in the ULTRA context.

The rest of the paper is organized as follows: After we present the running example in Section 2, we describe the ULTRA syntax and introduce the basics of the general ULTRA framework in Section 3. Section 4 then shows the instance for external actions, before we briefly sketch the execution model in Section 5. The paper is concluded with a short summary and outlook in Section 6.

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

2 The Running Example

As an example, let us assume that several credit card firms offer services online to charge a certain amount of money on a given card number, i.e. Visa offers a service chargevisa(Card, Amount), MasterCard has the service chargemaster(Card, Amount), American Express offers chargeamex(Card, Amount), etc. They also offer the services to cancel such debits (cancelvisa(Card, Amount), cancelmaster(Card, Amount), etc.). Moreover, the maximal amount of money that can be charged on a card can be requested by services availvisa(Card, Amount), availmaster(Card, Amount), etc.

Also, there is a service deliver(Address,Parcel) offered by a delivery agency to deliver a parcel to a given address. For reasons of simplicity we assume that it is not necessary to specify the sender of the parcel, but that this can instead be deduced from the given parcel number.

Now, our business is to combine these services into new services that can be offered to our customers. The new services will be

- a common interface to all the credit card firms, where the customer does not have to care whether the card is a Visa, American Express, Mastercard, etc.;
- a database of credit card numbers and the addresses of persons which can be used in conjunction with the charge- and deliver-services, such that it is enough to provide a key for a person (e.g. her email address), and credit card number or delivery address are obtained automatically;
- an integrated pay&deliver service which takes care of delivering a parcel to a person and charging the money for the delivery;
- a service that allows to charge several credit cards of a single person thus enabling her to pay an amount of money that is higher than possible with one credit card alone. That is, if a person has two credit cards with a limit of $1000 each, the new service allows to charge up to $2000 by debiting both credit cards transparently.

Obviously, these new services can be built using the basic services already available from the credit card firms and the delivery agency, resp. Compositionality is important for all these services: If, for example, a new credit card is to be supported, this must be possible without interfering with the existing services. Especially for the latter two types of composed services, transactionality is important: payment and delivery either have to be executed both, or none of them at all, and the same applies to charging several credit cards.

3 Overview of ULTRA

Before we actually start, some preliminaries have to be introduced, namely the general syntax of ULTRAprograms and the basic features of the generic framework.

We distinguish a set of *DB predicates*, a set of *basic update predicates*, and a set of *definable update predicates*. The DB predicates refer to observable state information, whereas the basic update predicates refer to executable (atomic) operations, i.e. calls to external services. The definable update predicates can be regarded as names of complex operations defined in the ULTRA language.

Example 3.1 For the services described in Section 2 the basic update predicates are constituted by the set of external services, e.g. chargeamex, chargevisa, deliver. The DB predicates are, for example, the names of the database relations containing the credit card and address data of a person, and the new complex services constitute the definable update predicates. □

3.1 Syntax

The syntax of ULTRA programs resembles well-known rule- or logic-based languages like Prolog. In addition to classical approaches, subgoals in our language can be connected using a *sequential composition* "**andthen**" or a *concurrent composition* "**and**". Sequential composition specifies that the subgoal on the right is to be evaluated in the state that results from executing the subgoal on the left, while concurrent composition states that the subgoals on the left and on the right can be evaluated concurrently in the same starting state. Where necessary, we use brackets "[. . .]" to emphasize precedence. Like always in logic- and rule-based programming, we denote variables with names starting with upper case letters, while constants have names with a lower case initial.

The syntax used in this paper differs slightly from the one used in our papers describing the formal ULTRA framework, like [Wic00, WFF00]: There, ":" is used for sequential composition and "," for concurrently composed goals. We believe that the syntax used here increases readability in the given context.

Example 3.2 [Rules for Services] The services described in Section 2 can be implemented by ULTRA rules as follows:

```
charge(Card,Amount) ←    type(visa,Card)  and  chargevisa(Card,Amount)
                      OR type(amex,Card)  and  chargeamex(Card,Amount)
                      OR type(master,Card)  and  chargemaster(Card,Amount)
```

We assume here that it is possible to deduce the credit card firm from the card number with an auxiliary predicate type(CardType, CardNumber). So, the rule for charge can be read "the service charge can be performed in one of the following ways: The type of the credit card is visa and the service chargevisa is executed, or the type of the credit card is amex and chargeamex is done, or the credit card is of type master and the service chargemaster is called".

Next, the rules for the services accessing a database of people's addresses and credit cards are shown. Here, we assume DB predicates cards and address representing database tables.

debit(Person,Amount) $\leftarrow$ cards(Person,C) **and** charge(C,Amount)
send(Person,Parcel) $\leftarrow$ address(Person,A) **and** deliver(A,Parcel)

Both rules are non-deterministic, as credit card or address must be chosen during execution if multiple cards or addresses are recorded for one person.

The service pay&deliver is simply a composition of the two above services. Note, that we use the concurrent composition, i.e. charging the credit card and sending the parcel can be done in parallel.

pay&deliver(Person,Amount,Parcel) $\leftarrow$ debit(Person,Amount) **and**
send(Person,Parcel)

All these services are quite straightforward to define, and they do not take special advantage of the expressive power of the rule language. This does not hold for the next new service, called multidebit, which can be defined recursively as follows:

multidebit(Person,Amount) $\leftarrow$ [cards(Person,C) **and** available(C,A') **and**
Amount $\leq$ A'] **andthen** charge(C,Amount)
OR [[cards(Person,C) **and** available(C,A') **and**
Amount $>$ A'] **andthen** charge(C,A')] **and**
multidebit(Person,Amount – A')

For simplicity, we assume that the various basic operations to check availability of money on a credit card have been encapsulated into a single service available which would essentially look like the service charge defined above. The first part of the rule captures the basic case of debiting on a card an amount of money that is smaller than the maximal available amount. The rule first checks the available amount of money against the one specified and then debits the card, i.e. it uses the sequential conjunction. This rule again is non-deterministic as it is not specified which of the available cards of a given person is charged. The second part is the recursive case which charges a card with the maximum amount and then calls itself again with the remaining amount. Note, that the recursive call is composed concurrently with the rest of the rule, as the different cards can be charged in parallel.

The rule for multidebit terminates successfully if the whole amount has been charged on the various cards. It fails if there is an amount of money left to charge, but no credit card which can be debited with it. □

3.2 The ULTRA Framework

The ULTRA framework is based on the definition of a transition system (Definition 3.3) and three functions to relate update and DB atoms to the transition system (Definition 3.4). On this basis, we can define a logical semantics with *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 this framework a number of theorems can be proven, among them the existence of a unique minimal model and a fixpoint characterization of this model. We omit all this here for the sake of brevity and only give the definitions for the transition system and the functions and refer the interested reader to [Wic00, WFF00] where all the theorems and proofs are given. Note that the full framework presented there additionally features a concurrent bulk operator which is also omitted here.

Definition 3.3 [Transition System] A *transition system* is a tuple $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \oplus)$ 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,
- $\sqcup : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ is a *concurrent composition* function
- $\oplus : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ is a *sequential composition* function

such that following algebraic properties hold:

1. $(\mathcal{T}, \sqcup, \Delta_\varepsilon)$ forms a commutative monoid, i.e. $\sqcup$ is commutative and associative and Δ_ε is a neutral element for $\sqcup$.

2. $(\mathcal{T}, \oplus, \Delta_\varepsilon)$ forms a monoid, i.e. $\oplus$ is associative and Δ_ε is a neutral element for $\oplus$.

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

$$\begin{array}{lllll} \text{(a)} & \Delta_1 \in \mathcal{T}_{Cons} \wedge \Delta_2 \in \mathcal{T}_{Cons} & \Longleftarrow & \Delta_1 \sqcup \Delta_2 \in \mathcal{T}_{Cons} \\ \text{(b)} & \Delta_1 \in \mathcal{T}_{Cons} \wedge \Delta_2 \in \mathcal{T}_{Cons} & \Longrightarrow & \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons} \\ \text{(c)} & \Delta_2 \in \mathcal{T}_{Cons} & \Longleftarrow & \Delta_1 \oplus \Delta_2 \in \mathcal{T}_{Cons} \end{array}$$

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:

$$\text{(a)} \quad (s \oplus_E \Delta_1) \oplus_E \Delta_2 = s \oplus_E (\Delta_1 \oplus \Delta_2)$$

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

$$\text{(b)} \quad s \oplus_E \Delta_\varepsilon = s$$

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

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.

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

Definition 3.4 [Mappings] 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 $\mathcal{B}_{BU}$, and $I^3_{\mathcal{B}}$ denotes the set of all (possibly three-valued) interpretations I over the Herbrand base $\mathcal{B}$.

A mapping $I_{DB} : \mathcal{S} \to I^3_{\mathcal{B}}$ which assigns a set of true and false DB atoms (observations) to each state, is called a *DB interpretation*.

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*.

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.

4 An ULTRA Instance for External Operations

While the ULTRA framework in general can handle arbitrary basic operations, in this section we define an instance that is already tailored to combining external operations, i.e. electronic services. The instance is completely specified with the actual set of basic operations left as the only variable. It especially features a datastructure for representing update requests that can also be used by a scheduler to ensure correct evaluation of concurrent transactions (see Section 5).

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 $I_{DB} : \mathcal{S} \longrightarrow I^3_{\mathcal{B}}$ be given as the projection of the states onto the observable properties represented by the DB atoms. Let $Log^{act} : \mathcal{B} \longrightarrow \Sigma$ be a given mapping from the ground DB atoms to the actions, and let $Upd^{act} : \mathcal{B}_{BU} \longrightarrow \Sigma$ 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.

Example 4.1 In our running example, the set Σ of actions is comprised by the available external service calls, i.e. chargemaster, availmaster, ..., deliver, ... □

Definition 4.2 [Compatibility] A relation $C \subseteq \Sigma \times \Sigma$ is called a *compatibility relation*, if it is symmetric, i.e. $(a_1, a_2) \in C \Leftrightarrow (a_2, a_1) \in C$, and

$$(a_1, a_2) \in C \ \Rightarrow \ do(a_2, do(a_1, s)) = do(a_1, do(a_2, s))$$

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

Two actions $a_1, a_2 \in \Sigma$ are called *compatible (w.r.t. C)*, if $(a_1, a_2) \in C$. □

Example 4.3 For our example introduced in Section 2 and Example 3.2 compatibility of the external services can be defined as sketched in Table 1. The table does not show all the

	chargevisa	chargeamex	deliver	cancelvisa	...
chargevisa	−	+	+	−	...
chargeamex	+	−	+	+	...
deliver	+	+	+	+	...
cancelvisa	−	+	+	+	...
...		...			

Table 1: Compatibility for some of the basic external services

external services but only a selection. Note that charge and cancel on the same credit card are not compatible in the sense of Definition 4.2 as cancelling a charge before it is actually done is undefined and may therefore lead to an arbitrary state. Also, two charges on the same credit card are not compatible, because if the sum of the two charges exceeds the credit limit, the final state depends on the execution order. □

The compatibility relationship of Definition 4.2 is similar, but in general not equivalent to the compatibility or conflict relationship used in database systems for transaction scheduling. Besides other differences, our compatibility relationship is defined only on actions, while concurrency control also takes pure retrieval into account.

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 [Pap86]. 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 an ULTRA instance, especially since the composition functions $\sqcup$ and $\oplus$ are easy to define with a commonly accepted semantics (see [Pap86] for more information about concurrent and sequential composition of processes).

Definition 4.4 [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*. $\square$

Definition 4.5 [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 $e_1 \leq e_2 \Longleftrightarrow f(e_1) \leq' f(e_2)$ holds for arbitrary events $e_1, e_2 \in V$ and $\mu = \mu' \circ f$. $\square$

Remark 4.6 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. $\square$

Definition 4.7 [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. $\square$

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 1.

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

Definition 4.8 [Concurrent Composition] The *concurrent composition* $\sqcup : \Sigma^{\ddagger} \times \Sigma^{\ddagger} \to \Sigma^{\ddagger}$ is defined by

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

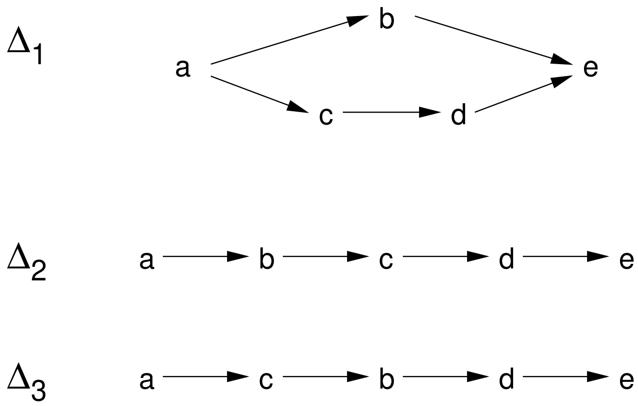

Figure 1: Some examples of partially ordered multi-sets represented as graphs

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 4.9 [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 4.10 [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.

□

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

Definition 4.12 [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 4.13 Let $\Delta_1,\Delta_2 \in \Sigma^{\ddagger}$ be arbitrary pomsets. Then the following equivalences hold:

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

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

Lemma 4.14 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 4.15 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:

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

$\leq_{\sqcup}$, $\leq_{\oplus}$, 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. $\square$

Lemma 4.16 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 $\leq \subseteq \leq'$. $\square$

Remark 4.17 [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 $\circ$. $\square$

Example 4.18 [Finite Linear Pomsets] The pomsets shown in Figure 1 on page 11 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:

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

$\square$

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 4.19 [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:

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

$\square$

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 4.20 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 4.8 and 4.9.

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 $\Delta_1 \oplus \Delta_2$. $\square$

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 4.21 [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:

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

$\square$

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 4.17.

Lemma 4.22 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 4.2 and 4.21. □

Lemma 4.23 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 4.22 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 4.16, 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:

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

$\square$

Definition 4.24 [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. $\square$

The definition of the execution $\oplus_E$ is well-defined due to Lemmata 4.14 and 4.23. Note that $\oplus_E^{lin}$ and $\oplus_E$ coincide for finite linear pomsets.

Lemma 4.25 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 4.15, $\Delta'_1 \oplus \Delta'_2$ 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 4.17).

Now we can reason as follows:

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

□

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 4.23 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. 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 in some sequential order. However, different from ULTRA pomsets, the transactions do not characterize a unique final state. The state that is actually reached depends on the exact execution sequence.

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 4.1. 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 4.26 [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 4.27 [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 $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \oplus)$ for the new ULTRA instance can already be considered as completely specified, because the missing parameters $\oplus_E$, $\sqcup$, and $\oplus$ have been defined in Sections 4.2 and 4.3. Nevertheless, we must verify the algebraic properties required in Definition 3.3. This will be done in the following theorem.

Theorem 4.28 [Algebraic Properties] The algebraic properties of Definition 3.3 hold for the transition system $(\mathcal{S}, \mathcal{T}, \mathcal{T}_{Cons}, \oplus_E, \Delta_\varepsilon, \sqcup, \oplus)$ defined for arbitrary external operations.

Proof: The assertion follows from the definitions of $\oplus_E$, $\sqcup$, 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 4.10.

Property 3 concerning consistency aspects follows directly from Lemmata 4.13 and 4.20.

Property 4 (a) has been shown in Lemma 4.25, while property 4 (b) holds by Definition 4.21.

□

In Theorem 4.28 we have shown that the concepts defined in this section are legal for the generic ULTRA framework, i.e. the requirements of definition 3.3 are met. Thus, we can apply the semantical results for the general framework (see [WFF00]) and get a particular semantics for the ULTRA language.

Example 4.29 We can now use the ULTRA instance to give some example answers to queries against the service definitions of Example 3.2. We do not show how the deltas are actually calculated here, but refer the reader to [WFF00] for details on the interpretation of rules in the semantical framework.

Let us assume that alice who lives in bobville has a Visa card with number 5 and a MasterCard with number 7. Her Visa credit limit is \$500, her limit on the MasterCard is \$400.

A service call debit(alice,100) then generates two deltas, namely

$$\begin{aligned}\Delta_1 &= [\{e\}, \emptyset, \{e \mapsto \mathsf{chargevisa}(5,100)\}] \\ \Delta_2 &= [\{e\}, \emptyset, \{e \mapsto \mathsf{chargemaster}(7,100)\}]\end{aligned}$$

Semantically, execution of either delta satisfies the specification of the service debit.

The deltas generated by a service call pay&deliver(alice,100,p) to deliver a parcel p are as follows:

$$\begin{aligned}\Delta_3 &= [\{e_1,e_2\}, \emptyset, \{e_1 \mapsto \mathsf{chargevisa}(5,100), e_2 \mapsto \mathsf{deliver}(\mathsf{bobville},\mathsf{p})\}] \\ \Delta_4 &= [\{e_1,e_2\}, \emptyset, \{e_1 \mapsto \mathsf{chargemaster}(7,100), e_2 \mapsto \mathsf{deliver}(\mathsf{bobville},\mathsf{p})\}]\end{aligned}$$

Note that both ordering relations are empty, i.e. the services can be executed concurrently. Again, either of the two deltas captures the semantics of the service call. As the specification of pay&deliver uses the complex service charge, Δ_1 and Δ_2 appear as subsets of Δ_3 and Δ_4, resp., due to the compositionality of ULTRA.

Finally, the deltas for the complex service multidebit(alice,700) are shown.

$$\begin{aligned}\Delta_5 = {} & [\{e_1,\ldots,e_4\}, \{(e_1,e_2),(e_3,e_4)\}, \{e_1 \mapsto \mathsf{availvisa}(5,500), e_2 \mapsto \mathsf{chargevisa}(5,500), \\ & e_3 \mapsto \mathsf{availmaster}(7,400), e_4 \mapsto \mathsf{chargemaster}(7,200)\}] \\ \Delta_6 = {} & [\{e_1,\ldots,e_4\}, \{(e_1,e_2),(e_3,e_4)\}, \{e_1 \mapsto \mathsf{availmaster}(7,400), \\ & e_2 \mapsto \mathsf{chargemaster}(7,400), e_3 \mapsto \mathsf{availvisa}(5,500), e_4 \mapsto \mathsf{chargevisa}(5,300)\}]\end{aligned}$$

This time some of the basic services are ordered as the rule for multidebit in Example 3.2 uses a sequential conjunction between the availability check and the actual charge. Note, that a delta with two charges of the same credit card is impossible for semantical reasons: this would lead to something like

$$\begin{aligned}\Delta_7 = {} & [\{e_1,\ldots,e_4,\ldots\}, \{(e_1,e_2),(e_3,e_4),\ldots\}, \{e_1 \mapsto \mathsf{availvisa}(\ldots), e_2 \mapsto \mathsf{chargevisa}(\ldots), \\ & e_3 \mapsto \mathsf{availvisa}(\ldots), e_4 \mapsto \mathsf{chargevisa}(\ldots),\ldots\}],\end{aligned}$$

which is not consistent according to Definition 4.19 because e_2 and e_4 are unordered in Δ_7 but not compatible (cf. Table 1 of Example 4.3). Consequently, Δ_7 can be ruled out already at the semantical level without having to know about the actual credit limits. □

5 Evaluating ULTRA Programs

In the previous section we have defined the formal logical basics for the evaluation of programs with arbitrary basic actions. To actually use the rule-based language for the specification of value-added electronic services we have to provide an execution model as well. This is briefly sketched in the following sections.

5.1 Evaluation Model

Our evaluation model consists of two major elements: A rule evaluation component acting as an interpreter for the rule language, and a scheduler ensuring that all complex actions and external services constitute a correct history even in the presence of concurrency (see Figure 2).

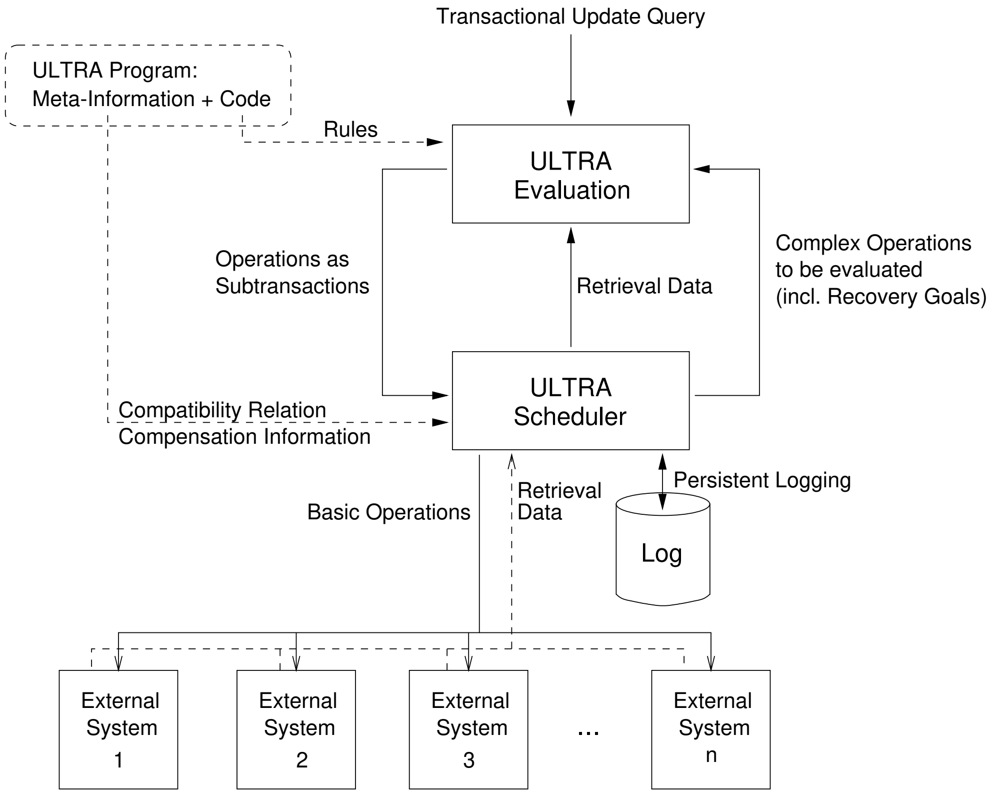

Figure 2: Evaluation is done in a handshaking fashion between rule evaluation and scheduler

The rule evaluation component uses a top-down strategy with backtracking and immediate updates. The latter is required for electronic services as no hypothetical reasoning about the effects of the services can be done, but they have to be executed during evaluation to be able to observe the effects. Queries are evaluated by resolving the query goal against the given rule program and then sending the resulting goals to the scheduler for execution: Sequential goals are sent one after the other, waiting for the result of the current goal before the next one is sent, while all concurrent goals are sent to the scheduler without waiting for the results. In both cases, a new subtransaction is started for the subqueries which can be used as a rollback sphere in case of failure. If all the (concurrent or sequential) subgoals succeed, the complex rule succeeds as well; if one of the subgoals fails, the whole rule must fail, too, and backtracking is necessary. As all the rules are evaluated within subtransactions, the backtracking of external services which may already have been executed is delegated to the scheduler by aborting the current subtransaction, thus ensuring that all traces of the failed rule are removed.

A few words concerning backtracking are in order: Backtracking occurs only if more than one definition of a service has been given, or if non-determinism is explicitly used by the programmer, as in the rule for debit in Example 3.2: The credit card actually debited is intentionally left unspecified. A deterministic rule would avoid backtracking there. However, base relations can be accessed without backtracking at all.

The task of the scheduler is to ensure that the concurrent execution of subgoals conforms to the restrictions of the ULTRA semantics, and to handle subtransaction aborts in case of failure. Additionally, it has to ensure serializability of concurrent top-level transactions [BHG87, BN97, WS92].

As the subtransactions executed during evaluation do not consist of ordinary read or write operations but contain arbitrary actions, standard procedures known from databases like restoring before-images cannot be applied. Instead, compensation [KLS98] as an advanced recovery technique is necessary. Compensation relies on semantically inverse "undo" operations to remove the effects of actions that have been performed by an aborted transaction. During rollback, the transaction log is scanned backwards and for every service call done within the aborted transaction the corresponding undo-operation is executed. The scheduler of our evaluation model uses this technique to rollback subtransactions with external actions.

Example 5.1 In our example of section 2, there already exist service calls to undo a charge to a credit card (cancelamex, cancelmaster, ...). On the other hand, for the delivery service there is no compensating service to get back erroneously delivered parcels. A corresponding service would have to be implemented, e.g. sending a person to the known address to pick up the parcel. □

The other task of the scheduler treated here is to ensure that the execution of concurrently composed subgoals is correct w.r.t. the ULTRA semantics. Therefore, the scheduler uses the pomset structure of the logical semantics also for its log. Every subtransaction then has its own "local log" which corresponds to one of the deltas of the query that is evaluated within the subtransaction. If it commits, the log is merged with the local log of the parent transaction, either using the concurrent composition if the parent transaction allows concurrent subgoals, or sequential composition if the parent is a sequential transaction. The scheduler ensures the correctness of concurrent ULTRA goals by guaranteeing that the merged logs stay consistent according to Definition 4.19. If a log would be inconsistent, this would mean that the corresponding delta is inconsistent, i.e. the actions of the goal are not valid w.r.t. the ULTRA semantics. Consequently, the transaction cannot be committed but must be aborted.

The scheduler may need an additional compatibility matrix like the one shown in Table 1 of Example 4.3 to make the consistency check for the logs. Note, that this compatibility matrix in general is different from the one used to schedule different top-level transactions, i.e. to guarantee serializability [BN97].

If a top-level transaction commits, the projection of its log onto the basic operations (note that the scheduler may log additional information) equals one of the possible deltas defined for the query by the logical semantics. While the logical semantics gives *all* possible deltas one of which has to be chosen for "materialization", the evaluation model sketched here performs this choice already during evaluation, as it is based on immediate updates. Using backtracking on the rule evaluation side together with subtransaction rollback on the scheduler side ensures that failing branches do not cause any actions.

Note that the architecture shown in Figure 2 differs significantly from the schema common in current database systems: The scheduler can no longer be independent from the query processor, because during query evaluation not only basic actions are sent to the scheduler, but also complex ones. While the former are passed on to the external system providing the service, the latter are passed back to the rule evaluation component after scheduling.

For a specialized fragment of the evaluation model which is restricted to sequential programs correctness has been shown in [FWF00].

5.2 Secure Calls to External Services

The formal semantics presented in Section 4 allows arbitrary actions without imposing additional requirements. Yet, the evaluation model sketched in the previous section has to restrict this freedom and adds two requirements on the services to ensure transactionality.

The first requirement is the existence of a compensation operation. As already mentioned above, this is necessary as no hypothetical reasoning about the effects of the services is done, but they are executed immediately during evaluation to be able to observe the effects. But note, that the compensation operation does not have to be supplied by the provider of the compensated-for service itself. Instead, a service from another provider can be used for compensation, or it can be programmed directly in the ULTRA language. Recall Example 5.1: The credit card firms explicitly offer services to compensate charges with the corresponding services cancelmaster, cancelamex, etc., but no direct compensation is offered by the delivery agency. Hence, a new service to get back an erroneously delivered parcel must be implemented, e.g. through a specialized collection firm.

The second requirement imposed by the evaluation model stems from the need to ensure transactionality even in the case of fatal errors, i.e. system crashes, etc. As partly executed queries have to be rolled back at system restart, it is necessary that the log which is kept in stable storage by the scheduler always contains all and only the executed services. In other words, all the executed services must be recorded in the log, and every recorded service must be executed. This property can be ensured if the services can take part in an atomic commit protocol like two phase commit (2PC). Then the execution of the service and writing of the corresponding log entry can be tied together and the log always contains all executed services.

If the two restrictions mentioned above are satisfied by an external service, the evaluation can be guaranteed to be transactional by our execution model. Note, however, that the issue of guaranteeing serializability of different top-level transactions is not addressed in this paper, although this may impose additional requirements like the partitioning of data on the external systems. See, for instance, [BGMS92].

The requirements resemble the situation given in transaction processing monitors [BN97, GR93]. However, the external systems are coupled more loosely to ULTRA than they would be to a TP monitor, and using compensation for recovery gives additional freedom.

6 Summary

We have shown an instance of the general ULTRA framework that is tailored to the execution of arbitrary external services. The instance was formally defined, and as the necessary properties for the general framework are satisfied, all the theoretical results for the framework hold for the instance as well. Consequently, it provides a well-defined semantical model for the specification of complex services.

We are now working on a complete formal treatment of the evaluation model as sketched in Section 5. Using pomsets as the data structure at the semantical level as well as in the scheduler is a milestone in this direction. A new equivalence relation between pomsets is developed that captures the semantical no-effect of compensated actions. This is necessary to show the equivalence of the scheduler's log with the logical delta. Another future work concerns ensuring correctness of concurrently composed subgoals: We will devise scheduling protocols for ULTRA concurrency that allow a more efficient check of whether the concurrent composition of two consistent pomsets again constitutes a consistent pomset. Similar to scheduling protocols used for top-level transactions, these protocols will be based on locking.

References

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

[BGMS92] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, 1992.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[BN97] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[FWF00] Alfred Fent, Carl-Alexander Wichert, and Burkhard Freitag. Logical update queries as open nested transactions. In *Transactions and Database Dynamics*, volume 1773 of *LNCS*, pages 45–66. Springer-Verlag, Berlin, 2000.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[KLS98] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In Vijay Kumar and Meichun Hsu, editors, *Recovery mechanisms in database systems*, chapter 15, pages 444–465. Prentice Hall, 1998.

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

[Pap86] Christos Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[WFF98] Carl-Alexander Wichert, Burkhard Freitag, and Alfred 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 134–165. Springer-Verlag, Berlin, Germany, 1998.

[WFF00] Carl-Alexander Wichert, Burkhard Freitag, and Alfred Fent. A logical framework for the specification of transactions. Technical Report MIP-0002, University of Passau (FMI), 2000. Available in the WWW at `http://daisy.fmi.uni-passau.de/papers/`.

[Wic00] Carl-Alexander Wichert. *ULTRA – A logic transaction programming language*. PhD thesis, University of Passau, 2000. (to appear).

[WS92] Gerhard Weikum and Hans-Jörg Schek. Concepts and applications of multi-level transactions and open nested transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, 1992.