Transactional Composition and Concurrency Control in Disconnected Computing

Tim Lessner*, Fritz Laux*, Thomas Connolly†, Malcolm Crowe†
*Fakultät Informatik, Reutlingen University, Germany, name.surname@reutlingen-university.de
†School of Computing, University of the West of Scotland, name.surname@uws.ac.uk

***Abstract*—Composition of software components via Web technologies, scalability demands, and Mobile Computing has led to a questioning of the classical transaction concept. Some researchers have moved away from a synchronous model with strict atomicity, consistency, isolation and durability (ACID) to an asynchronous, disconnected one with possibly weaker ACID properties. Ensuring consistency in disconnected environments requires dedicated transaction support in order to control transactional dependencies between software components and provide a scalable concurrency control mechanism. This paper contributes a simple expression language using Boolean operators to define transactional dependencies and further provides rules to derive an execution semantics that could be exploited by a transaction manager to control the interaction. This work also discusses the use of data classes that demarcate data based on concurrency control related aspects and apply a certain concurrency control mechanism to each class. Such a classification allows better trade-off between consistency needs and the overhead caused by the concurrency control mechanism.**

***Index Terms*—Transaction Management; Disconnected Transaction Management; Advanced Transaction Models; Concurrency Control; Optimistic Concurrency Control; Semantic Concurrency Control**

I. Introduction

The last few years have shown a need for mechanisms and technologies to easily compose scalable and everywhere available applications. Service Oriented Computing (SOC), specifically the composition of services as well as the automated execution of business processes, Cloud Computing, using infrastructure services via the Web in a pay as you need manner, and the growth in Mobile Computing solutions, available everywhere, all represent aspects of this development. In [1] we have presented our first idea of an optimistic and disconnected transaction model that supports local autonomy of software components – a key characteristic of SOC and Mobile Computing

The challenge for transaction management is to provide scalable mechanisms that ensure that data stays consistent across local boundaries between departments or even companies while the boundaries of transactions grow with the integration of new services –and their software components– to build new composite applications. In its widest form, data must be maintained consistently across several world wide distributed physical nodes due to availability demands and thus scalability is a key issue. Data often needs to be modified even if the connection is temporarily lost.

To facilitate loose coupling and increase autonomy, data should be modified in a disconnected and not in an online manner. This means that the set of proposed modifications to data is prepared offline and written back using a different set of transactions and not the same transactions that have been executed to read the data. This copes also well with the asynchronous message exchange that takes place in such an architecture. Data access is asynchronous too. In such an architecture, the traditional mechanism to keep locks on a database until the transaction has finished is no longer practical for the entire interaction. A locking isolation protocol, for instance, where other concurrent transactions read only committed results is not reasonable as it leads to long blocking time caused by the governing application's duration and the asynchronous message exchange.

Mobile Computing requires solutions for offline data processing despite the fast distribution and coverage of the mobile Web. Disconnected situations are frequent and users should be able to keep their data locally and synchronise the modifications back afterwards. Essentially, the circumstances in Mobile Computing are similar to that of SOC in that autonomy of software components is required including autonomy over the data they process.

A disconnected approach overcomes this challenge at the price of weakened isolation. The drawback of a weakened isolation is that other transactions can read pending results, which increases the danger for data to become inconsistent. To ensure consistency a validation must take place between (i) the phase a component (application) reads and modifies data locally and (ii) the phase the modifications are eventually written – merged with the database. Any transaction that is allowed to make an unverified change to data must specify a compensation transaction for restoring consistency if the process needs to be semantically undone later. For transactions that cannot specify a compensation it is therefore prohibited to make unverified changes.

Replication mechanisms that intend to increase the availability of data must scale and ensure that consistency of the different replicas is at least achieved eventually. An Eager replication [2] mechanism does not scale for highly replicated systems whereas lazy replication does, at the price that modifications of a transaction are not synchronised within the boundary of that transaction. Combining eager replication with a "master-slave" dissemination protocol scales and replicas can be made to converge within the transaction boundary [2].

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

But, in general, designing a highly replicated system requires a trade-off between the costs for consistency and scalability and thus availability [3], [4]. For some data, a mechanism for eventual convergence of replicas is sufficient, however, other data might require a much stronger consistency, possibly serializability. Some applications possibly require real-time behaviour, others may live with moderate availability. Recent research [3] shows that adaptive concurrency control based on a classification of data, leads to a better cost benefit ratio than one concurrency control to fit all needs. Thus, these considerations lead to a spectrum of different concurrency control protocols starting with no consistency at all and ending up with serializability.

The above review identifies the following characteristics of transactions that lead to complex transaction management:

1) Composition (dynamic): heterogeneous and autonomous software components represented as services are loosely coupled to create new composite applications. Due to the composition, transactional dependencies among the components arise.
2) Long-living nature: whereas the actual operations to read data and write modifications to the database are short lived[1], the overlying application (e.g., workflow) has a long-living nature. The result is a discrepancy between the time and the extent to which the physical operations (reads and writes) need ACID and the time the governing long-living application does. Isolation and consistency apply to both the read and the write phase of the application. However, compensation defeats durability.
3) Replication: nodes are physically distributed and replication must ensure that replicas converge depending on the data's semantics.
4) Disconnection: mobility requires disconnected transaction processing because of physical unavailability of the network connection. Also, to facilitate loose coupling, increased local autonomy is helpful to ease composition and due to an asynchronous message exchange, the set of proposed modifications to data is prepared offline for a later transacted sequence of operations on separate database connections. Notice, the long-living nature requires disconnected processing of data too, because keeping locks for the entire duration of the governing application would significantly reduce the concurrency.

In this paper, the focus is not on replication. We focus on composite, long-living, and disconnected transactions based on our first considerations published in [1]. We have removed much of the terminological overhead, and clearly demarcate concepts. One part specifically focuses on the transactional composition of transactions in a formalised manner using Boolean algebra (see Section III-B). We also added a section (see Section III-D) that deals with the classification of data based on the data's concurrency control (CC) properties and a different CC mechanism is applied to each class. The classification has been inspired by [3], [6]. We present a simple reference architecture (Section II), where we also introduce our idea of a "Disconnected Component". In Section IV, we present some existing transaction models and mechanisms as part of the related work, before Section V concludes this paper and outlines our future work.

[1]Molina et al. [5] state that to precisely classify a transaction as either short or long living is complicated. They define a transaction as long living if to lock data for the transaction's duration leads to an undesired decreased concurrency or even thrashing.

II. Architecture

We start by introducing a simple reference architecture (see Figure 1) that consists of three levels: database, middleware, and application level.

Database systems reside at the database level and they might be highly distributed as well as replicated. Also heterogeneous database federations, so called Multi-Database Systems (MDBS) [7], can exist. In an abstract view, the entire database level must be represented as one MDBS. Moreover, since mobile applications are also part of our architecture and mobile applications can use the mobile platform's database to increase their local autonomy to cope with frequent disconnections, the database can be logically also considered as a "Mobile MDBS" according to [8].

The middleware provides data access and owing to the assumed disconnected and asynchronous nature, data is read, copied, modified and synchronised back in a sequence of different independent transactions. A component (see Section II-A) starts a transaction (or a number of transactions) to read the data, disconnects, and locally modifies the data. After the modifications have been performed locally, the component sends just the changes back and based on these changes the middleware executes transactions to write the modifications. The middleware is allowed to use locks for reading and writing data from or to the database. Transactions in middleware and database layers are short-lived and locking is feasible, while retaining locks for the entire duration of the governing application is not practical.

The middleware plays a key role in this architecture. On the one hand it provides data access, on the other hand it has the role of a coordinator. Long running and hierarchically structured transactions involving many distributed, loosely coupled, heterogeneous, and autonomous systems require coordination. Also, interactions with external applications require transactional consistency. However, the middleware cannot enforce consistency of external systems. Often components are hosted by the middleware and composed together to build new applications as in SOC.

Application level refers to any component that implements concrete business logic. Components may also ship with their own, possibly replicated, database to increase their autonomy (see the "Composition Autonomy Pattern" in [9], for example). Mobile components are part of our reference architecture too. From a transactional point of view we do not believe that mobile components differ from stationary ones because both types of components have to cope with disconnection. For the remainder of this paper, we refer to a disconnected component

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

as a software component or just a component if the context is obvious.

The outer surrounding box in Figure 1 represents a transactional integration of components across the different levels.

A. Disconnected Component

An application consists of several software components that are either locally or remotely accessed. A component starts a number of transactions whereas the set of transactions to read the data is different from the set of transactions to eventually write the modifications. A component is defined to be either in its read, disconnected and working, or write phase, which is similar to Meyer's "Command Query Separation" pattern [10]. Figure 2 illustrates the idea.

To ensure consistency, the transition from disconnected and working to write requires validation. This validation is performed by a transaction manager but a component must ensure that the set of changes is passed to the transaction manager indicating at least the values or version read, and the new values of data. Technically, this means a component needs some book keeping functionality as defined in the Service Data Object (SDO) framework [11], for instance.

Components are allowed to call other component(s) to read data or to pass their modifications. Components used within a phase are said to be within this phase. Inside a component, the execution of flat transactions or calls to other components is not arbitrary and the implementation reflects an order in which execution takes place. We require a component to define this order if not defined elsewhere, e.g., by a workflow. Using another component is also represented as a transaction because these calls are transactional too. Their state is made persistent by writing the messages a component sends and receives to non-volatile memory. This concept is also known as persistent queuing. If each component specifies an order of execution, a composition order is the union of all the components' orders. If the read and write phases are separate, each component has to define two separate orders, one for the read and one for the write phase.

The Sagas [12] model discusses the notion of compensation to semantically undo the effects of long running transactions. Compensation has been introduced to cope with the requirement for weak isolation that arises if several transactions form a long living process but each of the sub-transactions is allowed to commit. Under this circumstance other transactions may read pending results. In case the transaction aborts, already committed sub-transactions need to be undone, which is only possible by executing a compensation, e.g., to cancel a flight is the compensation of booking a flight. If a sub-transaction is not compensatable, it is not allowed to unilaterally commit. The sub-transaction needs to pre-commit (promise) and wait for the global commit. If the global outcome is abort, the sub-transaction needs to rollback (there is no compensation). Compensation is discussed in Section III and here it is sufficient to introduce a compensation handler that points to another component that can implement the compensation.

DEFINITION II.1: (Disconnected Component):
A disconnected component is defined as a quintuplet $dc := (T^r, T^w, O^r, O^w, dc^{-1})$ with

1) a set of transactions T^r to read data,
2) a set of transactions T^w to write modifications,
3) a partial order for reading: $O^r = (V^r, E^e)$ with $V^r \subseteq T^r$ and the set of edges E^r is defined as $\forall t_n, t_o \in V^r : t_n \rightarrow t_o \Leftrightarrow e \in E^w$ with $e = (t_n, t_o)$. Transactions that do not belong to the subset V^r are said to be free transactions and hence can be executed in any order.
4) another partial order for writing: O^w be another partial order $O^w = (V^w, E^w)$ with $V^w \equiv T^w$ and the set of edges E^w is defined as $\forall t_n, t_o \in V^w : t_n \rightarrow t_o \Leftrightarrow e \in E^w$ with $e = (t_n, t_o)$. Transactions that do not belong to the subset V^w are said to be free transactions and hence can be executed in any order.
5) a compensation handler of dc.

If dc executes transactions T^r, it is in its read phase and if it executes transactions T^w it is in its write phase. Between these phases dc is in its disconnected and working phase. The write phase is not required for components that only read data.

The next section introduces the disconnected transaction model and provides a detailed definition for a transaction. A recursive model for transactional composition is the subject of this section too. The composition of disconnected components is eventually a composition of transactions. The resulting transactional dependency between two components is important and we provide a general applicable notion for them (see Section III-B).

III. TRANSACTION MODEL

The transaction model presented in this section is structured as follows: first, a general definition for a disconnected flat transaction is provided. The next part focuses on the composition of flat transactions and how to formalise the resulting transactional dependencies based on a Boolean expression. Based on such an expression, the third part discusses how to derive the execution structure of a composite transaction. The fourth part discusses different concurrency control protocols with a focus on optimistic and semantics based concurrency control (CC) mechanisms and defines different data classes according to the discussed CC mechanisms. This is the "Data View" of this model and its purpose is to demarcate data based on CC properties.

A. Disconnected, Flat Transaction

Our transaction model starts with the smallest unit: a flat transaction, the key building block. The following definition is based on the definition by Weikum and Vossen ([4],p.46) for a flat transaction.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

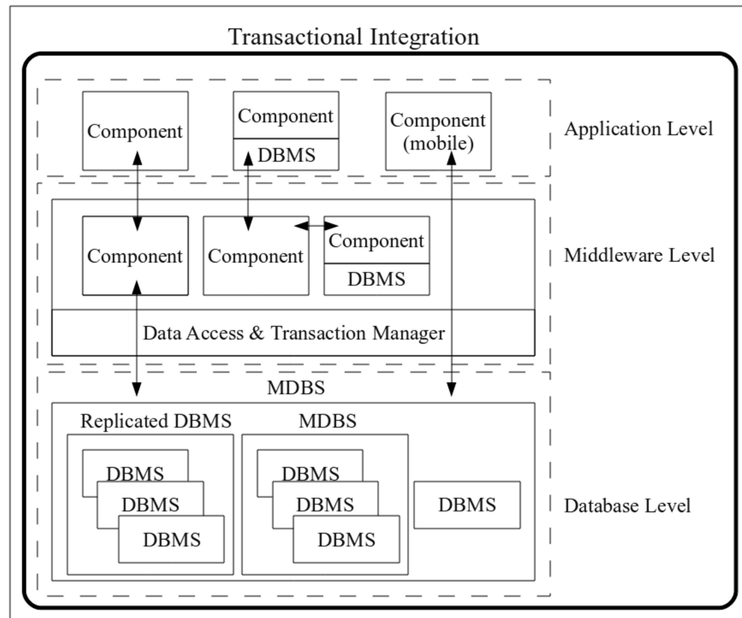

Fig. 1: Architecture

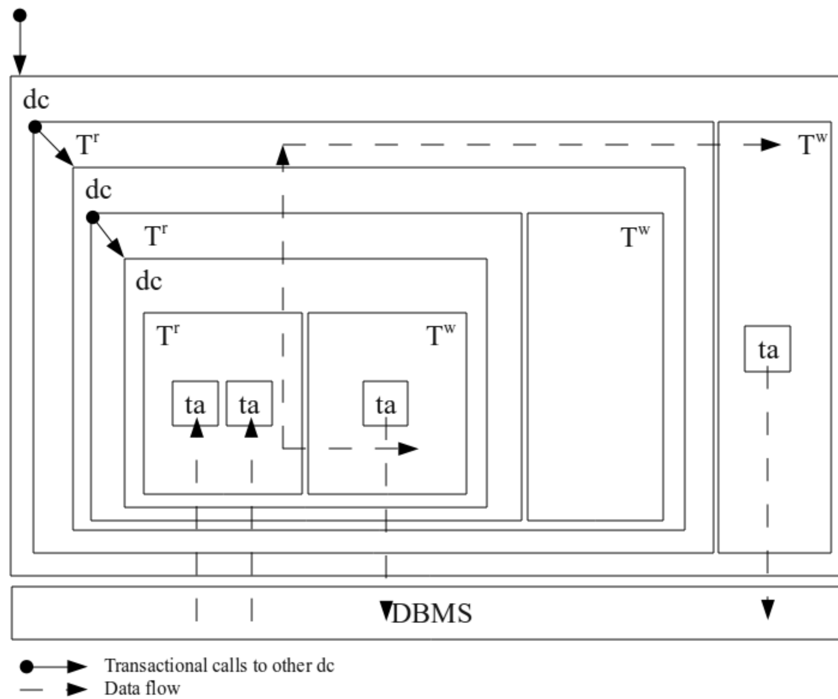

Fig. 2: Structure and composition of a dc. Dashed arrows show the data flow between the phases (shown by their related sets of transactions T^r, T^w) and upwards in the composition.

DEFINITION III.1: (Disconnected Flat Transaction I):

1) Let t be a flat transaction that is defined as a pair $t = (op, <)$ where op is a finite set of steps of the form $r(x)$ or $w(x)$ and $< \subseteq op \times op$ is a partial order.
2) A transaction is either in its reading (p1), disconnected and working (p2), validating (p3), or writing (p4) phase. The write phase is not required for read only transactions.

Section "Architecture" states that a software component uses different transactions to read and write data. This, however, requires a validation to ensure consistency. These phases have been introduced to avoid locking and support disconnection. They are similar to what is known as Optimistic Concurrency Control (OCC) [13], [14]. The only formally motivated difference is the explicit disconnected and working phase (p2). Usually, in the original OCC model, the actual work is done within the read phase. *"The body of the user-written transaction is in fact the read phase [...]"* [13]. Since we do not believe that Kung's reduction represents the actual phases of a disconnected transaction, these phases are made more explicit in our model. Later (see Section III-D) we introduce data classes and apply a certain CC mechanism. In this section also the phases, especially validation, are thoroughly discussed.

The next step is to provide a general notion for the composition of flat transactions. The idea is to consider a set of flat transactions and define their composition by a Boolean expression and based on this expression to derive an execution structure. The last step is to transform the model into a recursive model. With this recursive model in hand we have a general notion for the transactional composition or integration of software components too.

B. Composition of Transactions and Transactional Dependencies

In complex transaction processing scenarios, such as distributed or workflow transactions, the "Degree of Transaction Autonomy" [15] can be expressed by transactional dependencies.

One well known example is a distributed commit where, for example, two transactions have to either bilaterally commit or abort. This creates a transactional dependency between the transactions so that their autonomy is weak (becomes part of interpretation). In another situation, however, it might be possible that a transaction is allowed to commit even if other dependent transactions abort and the autonomy of this first transaction is high because it is independent of the others' outcome.

Consider, for example, a transactional workflow like the booking of a journey with several acceptable outcomes. The workflow consists of the booking of a hotel, a flight, and the booking of either a train or a car for a trip at the destination. So the satisfaction is that the booking of the hotel and the flight must succeed, whereas for the booking of the train or the car the allowed outcome is either the first or the second. Thus, it is required that only the first two transactions commit, whereas for the later ones only one is allowed to commit. In an auction, for example, where a user wants to purchase three items, it could be acceptable to buy just one, two, or all of them. Both examples show that applications can have different acceptable outcomes, a so called satisfaction.

Transactional dependencies have been investigated in the domain of nested and advanced transaction processing [7], [16], [17] where a parent, for example, depends on the commit or abort of its children – a property known as "vitality of a child". The opposite is known as "dependency of a child"; that is, a child depends on the commit of its parent. Notice, vitality and dependency affect the A of ACID.

The next section analyses transactional dependencies defined by a Boolean expression. We believe this is a useful reduction that makes transactional dependencies computable, and offers an execution structure.

1) Satisfaction of a Transaction: To represent transactional dependencies it is sufficient to define a satisfactory (acceptable) outcome for a set of transactions. For example, one satisfaction sf for $T = \{t_1, t_2, t_3\}$ could be $sf_1 = (c(t_1), c(t_2), c(t_3))$ another $sf_2(c(t_1), a(t_2), a(t_3))$ with c:=commit and a:=abort. Now, if after an execution of T (assumed, for example, a parallel one) one of the possible outcomes matches with the pre-defined outcomes sf_1 or sf_2, T can be committed. If not, T needs to be aborted and all (committed) $t \in T$ must be rolled back or compensated if committed already.

Another way of representing a satisfactory outcome uses Boolean expressions and interpret true as commit and false as abort. For example, the satisfactory outcome sf for $T = \{t_1, t_2, t_3\}$ could be $sf = (t_1 \wedge t_2 \vee t_3)$. For $T = \{t_4, t_5, t_6, t_7\}$ the satisfactory outcome could be $sf = (t_1 \vee (t_2 \wedge (t_3 \vee t_4)))$. Boolean expressions can express a nested behaviour, which is a key requirement to model transactional dependencies. Another benefit of Boolean expressions is that they can be verified. Boolean expressions would at least allow to compute the "Satisfiability" (SAT) or "Validity" of the expression itself. This information makes it easier to reason about the correctness of transactional dependencies.

DEFINITION III.2 (Satisfaction of transactions)**:**

1) Let $T_k = \{t_1, \ldots, t_n\}$ be a finite set of t where $T_k \subseteq T$ is a subset of the superset T of all transactions.
2) The set of satisfactory outcomes defining all acceptable outcomes for T is defined as: $SF(T) = \{sf_1(T_1), \ldots, sf_j(T_k)\}$ with $sf(T_k) = expr$.
3) Let
$expr := (expr)\ op\ (expr)$
$expr := c(t) \mid a(t)$
$op := \wedge \mid \vee \mid \oplus \mid p_l \mid p_r$
4) $v : TRUE \mapsto c(t)$ and $v : FALSE \mapsto a(t)$ with $c(t)$ being the commit and $a(t)$ the abort of a transaction.
5) Let $OUT(T) = \{out(t_1), \ldots, out(t_i)\}$ be the set of the

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

atomic transactions' outcomes after transaction processing with $out(t) \in \{c(t), a(t)\}$.

6) Let $\sigma : (OUT(T), sf(T)) \mapsto s(T)$, with state $s(T) = c(T)$ or $a(T)$, commit or abort of a set of transactions T. Each $out(t_i)$ is mapped to the according occurrence in each $expr$. Validation function σ validates each $expr$.

Concerning the number of allowed operators op compared to the number of Boolean operators, the number of allowed operators is limited to "AND" $\wedge$, "OR" $\vee$, "XOR" $\oplus$, and the two projections p_l, p_r. See Table I for a complete overview.

1) Operator $\wedge$: The logical "AND" is a common case and only if both transactions commit, the result is committed.
2) Operator $\vee$: The logical "OR" represents the situation where it is sufficient if at least one of the transactions commits.
3) Operator $\oplus$: The logical XOR where exactly one transaction is allowed to commit.
4) Operators p_l, p_r are projections and discussed as well as defined (see Definition III.3) below.

Other Boolean operators are less reasonable with respect to transactional dependencies. Implication, for example, would mean that the abort of two transactions, i.e., $f \rightarrow f = t$ validates to true even if the transactional system's state has not changed. Generally, operators validating abort and abort (false and false) to true are less reasonable. Except the "XOR", operators that validate commit and commit to false are less reasonable for the same reason. Generally, the semantics of the logic is that the abort of a transaction is not a correct result, even though it is consistent. Although the "XOR" operator is an exception, it is required because in some scenarios only one of two transactions is allowed to commit (book either the train or rent a car; buy either this item or the other).

The "NOT" operator is not listed in Table I. To define the satisfaction of a transaction as not commit (=abort) means the processing of the transactions is not intended at all. For example, the expression $expr = t_1 \wedge \neg t_2$ states that a commit of t_1 and an abort of t_2 is satisfying. This expression is equivalent to $expr = t_1$ because to start the execution of t_2 with the goal to let t_2 abort is not correct. To model a transaction with the intention to let the transaction abort is not reasonable. Even in a situation, for example, to test a system with the intention to throw an exception, the general semantics of a transaction requires the transaction to commit to throw the exception. The ambiguity with this operator is that for a single transaction t, the possible outcome is indeed $expr = t \oplus \neg t$.

To resolve this ambiguity and to comply with the correctness semantics of a transaction, the "NOT" operator is not allowed in expressions, but in case an expression needs to be optimised for validation and therefore transformed into a conjunctive or disjunctive normal form the "NOT" operator might be required. For instance, expression $a \oplus b$ can be transformed into the disjunctive normal form $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$

The function v maps true (T) to the commit of a transaction and false (F) to abort.

TABLE I: Operators for transactional dependencies.

#	t_i	t_j	$\wedge$	$\vee$	$\oplus$	p_l	p_r
1	c	c	c	c	a	c	c
2	c	a	a	c	c	c	a
3	a	c	a	c	c	a	c
4	a	a	a	a	a	a	a

To perform a validation $OUT(T_k)$ represents the set of outcomes of atomic transactions T_k. Based on these outcomes, the outcomes of composed transactions T_k are computable. Regarding σ it is important that the state after validation is actually pending since the final outcome might not be determinable yet as it possibly depends on the validation of other dependent T_l.

The projections p_l and p_r represent a transactional dependency where the outcome (result) of one transaction (operand) supersedes the other. For p_l the left argument supersedes the right, for p_r the right supersedes the left (see Table I).

Special cases of transactional dependencies have been investigated already and are known as "vitality" or "dependency" of a transaction. A transaction is said to be vital if its abort leads the parent transaction to abort too. Non-vital if the child's abort does not affect the parent. A transaction is said to depend on the parent, if the parent's abort leads the child to abort too. If not, the transaction is said to be independent.

DEFINITION III.3 (Projection operators)**:** Let the left argument of a projection be the parent and the right argument be the child. Then, according to the notions of vitality and dependency let projection p_l be a combination of non-vital and dependent and let p_r be a combination of vital and dependent.

Projection p_l is non-vital because even if the child (right operand) aborts the outcome is still commit. It is dependent because if the parent (left operand) aborts the global outcome is abort and the child must be rolled back. The interpretation of p_r is accordingly.

It is also possible to define an interpretation for the other operators according to the notions of vitality and dependency. Table II shows the possible combinations of vitality and dependency and how they map to the operators. Notice that only mixed outcomes are shown in the table as vitality and dependency relate to mixed outcomes only. The only exception –again– is the "XOR" operator because the behaviour is different in case both transactions commit, as discussed above, and vitality as well as dependency are not directly applicable to the "XOR" operator. Notice, our model subsumes the notions of vitality and dependency.

Strictness (see Table II), which is given if one of the concepts is dependent or vital, describes the autonomy of a transaction and a strict operator represents a weak autonomy whereas a non-strict operator represents autonomy. Although vitality and dependency are not applicable to "XOR", it is strict because the transactions are abort dependent. If both

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

can commit, both have to abort.

Concerning the validation of an expression, the projection operators have an interesting property. For example, consider the expression $expr = t_1\ p_l\ (t_2 \vee (t_3 \wedge t_4))$. If $expr$ is reduced to $expr = expr_1\ p_l\ expr_2$ where $expr_1 = t_1$ and $expr_2 = (t_2 \vee (t_3 \wedge t_4))$ it can be shown that in each possible distinct permutation of true and false, the outcome of $expr_1$ supersedes $expr_2$ (see Table I). Thus, there is a dominant part and projection operators (both operators show this behaviour) should be validated first and the remaining non-dominant part needs not to be considered for validation.

Based on the observations so far, the precedence, associativity, and commutativity can be defined as in Definition III.4.

DEFINITION III.4 (Precedence, Associativity, and Commutativity)**:** The operator precedence is defined from high to low as follows: projections p_l and p_r, conjunction $\wedge$, disjunction $\vee$, and XOR $\oplus$. The associativity convention is that operators associate to the right.

LEMMA 1: The projections p_l and p_r are not commutative.

Proof: By contradiction. Suppose the projections p_l and p_r are commutative. Given an expression $expr_i\ p_l\ expr_j$ and let $expr_i$ validate to true and $expr_j$ to false, hence the global result is commit (true). Due to commutativity the global result is commit too, if $expr_j$ commits and $expr_i$ aborts. As shown in Table I this is not true and the result is abort in case $expr_i$ aborts and $expr_j$ commits. Since the proof for $expr_i\ p_r\ expr_j$ is equivalent, the projections are not commutative. ■

Regarding projections and validation, an expression can be reduced and nondominant parts can be skipped for validation. Figure 3 shows two example syntax trees. In the first example the expression $expr = (t_1 \vee t_2) \wedge (t_3 \oplus (t_4\ p_l\ (t_5\ p_r\ t_6)))$ is reduced to $expr' = (t_1 \vee t_2) \wedge (t_3 \oplus t_4)$. Part $(t_5\ p_r\ t_6)$ is nondominant and does not affect the global outcome. Similarly, the expression $expr = (t_1 \vee t_2)\ p_r\ (t_3 \oplus (t_4 \wedge t_5 \wedge t_6))$ is reducible to $expr' = (t_3 \oplus (t_4 \wedge t_5 \wedge t_6))$.

DEFINITION III.5 (Reducibility of Projections)**:** In a projection the argument that is not projected is called non-dominant. For $expr_i\ p_l\ expr_j$ the non-dominant part is $expr_i$, for $expr_i\ p_r\ expr_j$ it is $expr_j$.

LEMMA 2: For validation, an expression $expr$ can be reduced by all nondominant expressions

Proof: By contradiction. Given an expression $expr_i\ p_l\ expr_j \neq expr_i$. From Table I it follows directly that this is not possible. ■

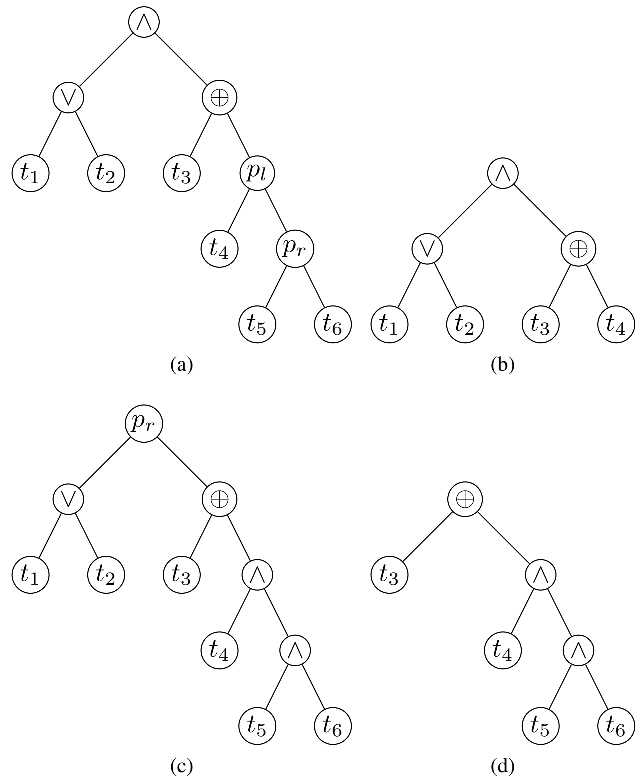

Fig. 3: (a) Complete and (b) reduced syntax tree of $expr = (t_1 \vee t_2) \wedge (t_3 \oplus (t_4\ p_l\ (t_5\ p_r\ t_6)))$. (c) Complete and (d) reduced syntax tree of $expr = (t_1 \vee t_2)\ p_r\ (t_3 \oplus (t_4 \wedge t_5 \wedge t_6))$

So far, a representation for transactional dependencies using a reduced Boolean algebra has been introduced as well as the required validation rules. The next step is to define the "Execution Structure" with respect to atomicity.

C. Execution Structure - Atomicity

Operators link transactions and define their transactional dependency, but to ensure atomicity as defined by a satisfaction expression requires some additional measures. For example, what is additionally required if two transactions state a strong dependency indicated by the $\wedge$ operator? What is required if two transactions are linked by a projection operator? The idea here is to derive the "Execution Structure" from the sf function.

For example, given the expression $sf = t_1 \wedge t_2$ requires an atomic commitment, a 2PC for instance, because both have to commit or abort. Usually, either t_1 or t_2 plays the role of the 2PC coordinator or an additional instance fulfils this role. But, in either case this additional measure of a coordinator is neither represented by t_1 nor by t_2. The operator, the $\wedge$ in this example, indicates the measure that needs to be taken, namely, to ensure that both transactions belong to the same composite transaction, which only commits if both children do. Technically, a transaction manager needs to create a context (a composite or boundary) to coordinate the interaction among

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II: Operators, Vitality, and Dependency.

op	t_i	t_j		vitality, dependency	strictness
$\wedge$	c	a	a	t_j is vital for t_i	strict
$\wedge$	a	c	a	t_j depends on t_i	
$\vee$	c	a	c	t_j is non-vital for t_i	non-strict
$\vee$	a	c	c	t_j is independent from t_i	
$\oplus$	c	c	a	-	strict
$\oplus$	c	a	c	-	
$\oplus$	a	c	c	-	
p_l	c	a	c	t_j is non-vital for t_i	strict
p_l	a	c	a	t_j is dependent from t_i	
p_r	c	a	a	t_j is vital for t_i	strict
p_r	a	c	c	t_j is independent from t_i	

transactions. Transactional context is a rather implementation related concept to ensure that each transaction is within the same boundary, i.e., has the same context, e.g., transaction id or possibly shared data. It is further possible to define an atomic completion (Atomic Sphere [18]) for a context. The boundary in turn demarcates a composite of transactions and may be a sub-transaction in a larger context. This should become clear if an application involving several transactions is modelled as a recursive tree.

So, deriving the composite structure means deriving the atomicity related execution semantics for the operators and the result is the actual execution structure itself. For example, $sf = t_1 \wedge t_2$ technically (which measures need to be taken to ensure atomicity) means $TA = (t_1, t_2)$ where TA is a composite requiring an atomic outcome.

Before providing a definition for the derivation, the idea for the definition is motivated first.

One motivation for a demarcation of sub-transactions is as follows: Let us assume that an application is not divided into sub-transactions and represented as one transaction instead. Then the entire transaction needs to be processed as one unit of work, which increases the blocking time (in case locking schema concurrency control protocols are used) of resources due to the fact that more things need to be processed within one atomic step. In case OCC is used resources are not blocked, however, the problem is still the aforementioned discrepancy between the time the transaction lasts and the probability of conflicts (the longer the transaction lasts, the higher the conflict probability). If sub-transactions could be demarcated though, these units could be processed individually and interleaved with others to increase the concurrency. The price in turn is that to individually process sub-transactions leads to a weakened atomicity because some sub-transactions possibly have committed before the entire global transaction has terminated. In case the global transaction aborts, already committed sub-transactions need to be compensated. This is only possible if compensations for the already committed sub-transactions exist[2]. Compensation can be interpreted as a necessity to conform to reality like the cancellation of a booked flight. Here, we are only interested in motivating our idea, which is based on the notion of compensation, so just the basics are discussed. For further details on compensation, see Garcia Molina [12] and Leymann [19].

[2]Depending on the isolation (open or closed, see next section) a child's result might be visible to all transactions or just the parent. In the latter case the commit of the parent publishes its children's results.

A sub-transaction (ST) is allowed to unilaterally commit if: (i) a compensation for ST exists. If the component is a composite, a compensation for each member must exist; (ii) in case some members have no compensation, these members must be free of effects (e.g., read-only transactions are free of effects). If neither (i) nor (ii) holds, then a unilateral commit is not possible as long as the global outcome has not been determined. The outcome is determined if the satisfaction has been validated, but as long as the outcome has not been determined, the ST is in state "pre–commit" – a promise to eventually commit. An abort is always final and in case the global outcome is abort (satisfaction is false), each committed or pre-committed ST must compensate or abort and release its state ("State Release"). In case the global outcome is commit, each ST can retain its state ("State Retention"), also the aborted ones.

Applying the idea of "State Retention" and "State Release" to the operators, it follows that only the non-strict operator (see Table II) retains its state independent of the global outcome. The reason is the "OR" explicitly represents a transactional dependency where one of the transactions is allowed to abort although the global outcome is commit. This in turn means that transactions linked with the non-strict operator do not require to belong to the same **directly** higher ordered transaction (composite). Notice that due to the recursive nature they might be part of some higher ordered transaction (composite). Regarding strict operators, a situation is given where one of the sub-transactions commits even though the outcome is abort. Therefore, the committed transaction has to release its state and strict operators always require the same directly higher ordered transaction (composite) for their linked transactions.

First we define a composite transaction as follows:

DEFINITION III.6: (Composite Transaction) Let $T_i \in \{ta, TA\}$ be either a flat transaction ta or a composite transaction TA where $TA := \bigcup_{i=1}^{m} T_i$.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Notice, this is a re-definition of t and T. From now on, ta represents a flat transaction as defined in Definition III.1. Since T might be either a flat transaction or a composite, the model is recursive.

Next, we define the derivation of a composite as follows:

DEFINITION III.7 (Derivation of the execution structure)**:** Given an expression $T_i \ op \ T_j$ with $op \in \{\wedge, \oplus, p_l, p_r\}$ it follows that T_i, T_j are part of a higher (parent) TA with $T_i, T_j \in TA$. TA is called a derived composite transaction.

EXAMPLE III.1: Given the following expression:
$sf = ((ta_1 \wedge ta_2) \vee ta_3 \vee (ta_4 \wedge ta_5) \vee ta_6 \vee (ta_7 \wedge ta_8)) \oplus (ta_9 \wedge (ta_{10} \ pr_l \ ta_{11}))$.
Applying the rule from Definition III.7 the following structure is derived:
Due to the $\oplus$:

$$TA = \{TA_1, TA_2\} \tag{1}$$

Applying the rules on TA_1:

$$\begin{aligned} TA_{1,1} = \{ta_{11}, ta_{10}\}, TA_{1,2} = \{TA_1, ta_9\} \\ TA_1 = TA_{1,2} \end{aligned} \tag{2}$$

Applying the rules on TA_2:

$$\begin{aligned} TA_{2,1} = \{ta_8, ta_7\}, TA_{2,2} = \{ta_6\} \\ TA_{2,3} = \{ta_5, ta_4\}, TA_{2,4} = \{ta_3\} \\ TA_{2,5} = \{ta_2, ta_1\} \\ TA_2 = \{TA_{2,1}, ta_6, TA_{2,3}, ta_3, TA_{2,5}\} \end{aligned} \tag{3}$$

It is important to understand that none of the TA actually exists during the time the sf has been defined. A TA is a derived composite to ensure the atomic outcome of its components, its nature is purely technical, and it is created at the time the expression is validated.

Notice, in Example III.1 we can just write T instead of ta and leave the actual type (TA or ta) open because of the recursive nature.

The next step is to consider the situation in which a T_i exists more than once within the same sf. As said, the difference between a derived composite TA and a T is that TA is a transactional context, whereas a T exists with respect to a functional requirement. Assume, for example, given the following expression $sf = (T_1 \wedge T_2) \vee (T_1 \wedge T_3)$. After the derivation of the composite structure we have $TA_1 = \{T_1, T_2\}$ and $T_2 = \{T_1, T_3\}$. This means even if T_1 actually exists only once, the derived execution structure states it belongs to two different composites. This could be problematic, namely in case T_3 aborts T_1 has to abort too, and since T_1 belongs also to TA_1, TA_2 has to abort too. Thus, T_2 is not independent of T_3 and both composites need to be aggregated.

Another reason is that a satisfaction sf may have the structure $sf = sf_1 \vee sf_2 \ldots \vee sf_i$ to model different acceptable outcomes – a key requirement especially when dealing with compensation. For example, $expr = (T_1 \wedge T_2) \vee (T_1 \wedge T_1^{-1} \wedge T_2 \wedge T_2^{-1})$ where T_i^{-1} represents a compensation of T_i. Due to the rule in Definition III.7 each $\vee$ operator leads to the creation of its own boundary, which means there are eventually different representations for the same sf. There are different execution paths –this is why a satisfaction is required– but there is one execution structure only.

DEFINITION III.8 (Union of derived composites)**:** Whenever two derived composites TA interleave, $TA_i \cap TA_j \neq \emptyset$, they have to be joined. Formally, the transitive closure is defined as $TA'^{+} = \bigcup_{n>0}^{o}$ where $TA'_n = TA'_{n-1} \cup \{(T_i, T_k) \mid \exists \ T_j : (T_i, T_j) \in TA_{n-1} \wedge (T_j, T_k) \in TA_{n-1}\}$, $i \neq j \neq k$.

EXAMPLE III.2: Given the following expression:
$sf = T_1 \wedge T_2 \vee (T_3 \vee T_4) \wedge (T_5 \vee T_6 \vee (T_7 \wedge T_8 \oplus T_9 \wedge T_{10} \ pr_l \ T_1))$.
Applying the rule from Definition III.7 and III.8 the following structure is derived:

$$\begin{aligned} TA_1 = \{T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_1\}, \quad (1) \\ TA_2 = \{T_1, T_2\} \\ TA_1 \cap TA_2 = T_1 \Rightarrow TA_x = \{T_1, \ldots, T_{10}\} \quad (2) \end{aligned}$$

As shown, the entire expression requires one derived composite TA_x to control the execution.

EXAMPLE III.3: Given the following expression:
$sf = (T_1 \wedge T_2 \vee T_3) \vee (T_4 \wedge T_5 \vee T_6) \vee (T_1 \wedge T_7)$. Applying the rule from Definition III.8 recursively the following structure is derived:

$$\begin{aligned} TA_1 = \{T_1, T_7\} \quad (1) \\ TA_2 = \{TA_{2,1}, TA_{2,2}\} \quad (2) \\ TA_{2,1} = \{T_6\}, T_{2,2} = \{T_4, T_5\} \\ TA_3 = \{TA_{3,1}, TA_{3,2}\} \quad (3) \\ TA_{3,1} = \{T_3\}, T_{3,2} = \{T_1, T_2\} \\ TA_1 \cap TA_3 = T_1 \Rightarrow TA_x = \{T_1, T_2, T_3, T_7\} \quad (4) \end{aligned}$$

As shown, the entire expression requires two derived composites TA_1 and TA_x to control the execution.

Example III.3 defines the allowed outcomes and based on the sf, an execution structure is derived. The result are two derived composites TA_1 and TA_x, which means, that there are two independent composites. Assumed, the sf stands for a workflow, a representation of the workflow itself is missing. To represent this, a root composite TA^R needs to be created. So the actual execution structure of sf is: $TA^R := (TA_1, TA_x)$. Notice, a root composite is only required if there are more than

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

one derived composites after the last step of the recursion. We come back to the root composite briefly.

The next step is to provide a definition for an atomic completion protocol of a derived composite TA according to the "Open Nested Transaction Model" [20].

DEFINITION III.9 (Atomic completion)**:** Each member T of a derived composite TA (except the root) has to retain its state in case the outcome of TA is commit. In case the outcome of TA is abort, T has to release its state by either rolling back if T is in state pre-commit or by compensation if T has locally committed already. T is only allowed to unilaterally and locally commit if it has a compensation T^{-1}. A transaction retains its state by a local commit, which is only required if no T^{-1} exists. TA' is aborted if all T have released its state in the opposite order.

The reason why the root has been excluded in Definition III.9 is that the root is only required if there are more than two derived TA left after the last step of recursively applying the rules. This can happen only if independent TA exist, which in turn means even in case one TA aborts the other may still commit and retain its state. TA^R serves a different need and technically it just reflects the results of all TA and hence all T.

An execution order of transactions has not been considered yet. The reason is that an execution order for transactions is usually given by the application's implementation. Sometimes the order is applied externally to an application, for example, a workflow model could define the execution order. For our model it is just important that the order is accessible by the transaction manager to perform compensation if required. The order has been defined in Definition II.1.

Summarising this section, based on a sf for a set of transactions it is possible to derive an execution structure. The result are derived composite transactions that are technically required to ensure an atomic outcome (compensation is semantically equivalent to an atomic outcome) as defined by the sf.

The next section focuses on Data Classes and concurrency control related aspects. Whereas transactional dependencies affect the atomicity, consistency and isolation are subject of the next section.

D. Data Classes

As said in the Introduction already, disconnected data processing requires a non blocking mechanism that is able to ensure a strong consistency where the state of the data modified by a transaction has not changed during the execution – the so called isolation property –. Locking over the entire lifetime of the governing application is an inadequate solution. The problem with locking lies specifically in the long duration and the longer the transaction lasts, the longer the data is locked, and the less is the concurrency because other transactions cannot obtain any locks for this data. Additionally, the longer the transaction lasts the higher is the probability that the transaction might fail. A proper time out setting must be considered as complicated because the database is left unaware about the intended duration of the application including the user interaction. Notice, even when using locking separately for the transaction(s) reading data and the one(s) writing modifications, inconsistencies might still arise because the transaction(s) to read and the transaction(s) to eventually write the modifications are logically independent. Another reason why locking is inadequate is for applications running on mobile devices. In mobile computing, network fragmentation and the resulting disconnection is considered as "being normal".

OCC is a solution for this issue. However, the problem with OCC is the discrepancy between the time a transaction lasts and the probability of of others wanting to modify the data. This is specifically crucial for update intensive entities –so called "Hot Spot Fields".

To provide consistency for fields that are subjected to this asynchronicity, one solution is to exploit the semantics of fields where the conflict probability is high[3]. Hot spot fields are usually numeric and operations performed on these fields are thus arithmetic and usually commutative. Also, constraints on such fields are common too. For example, the current stock of an item or the number of available seats in a flight is limited. O'Neil [21] discusses the use of an "Escrow" data type for such fields. For "Escrow" fields, transactions can request a kind of a guarantee at their start time to successfully perform their modifications at the end. The guarantee can even depend on a constraint. If a guarantee has been granted, the transaction can continue processing in a disconnected mode and with the escrowed guarantee in hand, the transaction can commit successfully, assuming there are no other conflicts with non-Escrow fields. Concurrency on these fields is increased with such an approach because locks are required only for the time the guarantee request is processed (a consistent view on the "Escrow" field is indispensable during this time). Hence, even if O'Neil's concept is pessimistic because actions are taken at the beginning of a transaction it fits well into a disconnected architecture.

Laux and Lessner [6] discuss a similar idea, however, instead of requiring guarantees at the beginning of the transaction their approach is optimistic (i.e., no measures at the beginning of the transaction) and performs a validation before writing. If the validation fails for a field, "Reconciliation" is possible if a "Dependency Function" for the conflicting entity is known and if the transaction wants reconciliation for the conflicting operations. "Reconciliation" describes the process of replaying an entire transaction or just the conflicting operation with the actual state of the database. To replay is only possible if the transaction or the conflicting operation is independent of further user input. This type of independence

[3]To precisely define such a probability requires statistical measures and usually a Poisson Distribution for conflicting transactions is postulated. See Kraska et al. [3] for a model for probability based consistency in replicated database systems.

is similar to the notion of "Logical Access Invariance" [22], [23].

A "Dependency Function" has the new value, the value read, and the current value of a field as possible input parameters and calculates a semantically correct state despite a conflict. For numeric fields, where operations are additions, subtractions, and multiplications, this function is a "Linear Dependency Function" (see [6] for further details).

The drawback of [6] compared to [21] is that in case of a constraint violation, the transaction has to abort possibly after a lot of work has been done. This is especially disadvantageous if the conflict rate is high on that field. Indeed, this reflects the optimistic behaviour and is therefore intended, but guarantees like O'Neil's mechanism would be preferable despite their pessimistic nature especially if the conflict rate on this field is high.

Another difference is that Laux and Lessner's approach has been designed for architectures where a "change-set" of proposed data is delivered by the client. The modifications are then passed to stored procedures or SQL transactions are generated and executed according to the changes. Change-sets, for example as used in Service Data Objects (see [11]), comply with the current nature of computing architectures where communication is asynchronous, message oriented, and disconnected. For example, also the work by Thomson and Abadi [24] is based on the observation that transaction processing has shifted away from a synchronous to an asynchronous mode. Asynchronous thereby means modifications are prepared offline and just the results of modifications or the new values are sent back to the database.

An additional difference between these concepts is that O'Neil foresees a classification based on a data type, but Laux and Lessner's approach is on a transaction or even an operation base.

Both [6], [21] have in common that their mechanism applies to a certain kind of data only, usually numeric data. Even if the concept of a dependency function is a formal improvement compared to O'Neil, because it defines a precise property for "Escrow" fields (linear dependent in case of numeric types), it seems rather questionable to find such functions for non-numeric types. So, a non-blocking solution for all "Non-Escrow" fields is needed, which ensures strong consistency (e.g., serializability) already at the beginning of the transaction's read phase, especially if the conflict probability on those fields is high. One option is to consider once again the semantics of those fields as well as the way the data is accessed. We believe that in many situations semantics are given so that the conflict rate is very low. For example, data belongs to a certain instance or node, or data is modified using insert semantics without even requiring an isolation level of "Repeatable-Read".

But after all, some data might be just subject to conflicts. In a very limited way we believe it is possible to use locking, namely, only if to lock is "logically motivated". For example, a salesman with his associated customers could have an owner role for these customer data and due to the ownership, locking is justified. Locking in this context should not be considered as a mechanism to control concurrency (transparent for developers), it is rather a mechanism to enforce an ownership of data due to application specific issues (not transparent for developers).

In case a lot of data requires a strict locking, our model is not adequate. For example, frequent calculations over a set of data with an isolation level of repeatable read.

According to these previous considerations and according to [6], [21] the following properties are distinguishable (we will use the following abbreviations in Table III):

- properties of the data: does a constraint ($cons$) exist, is the type numeric (num)?
- the operations' semantics: are operations on this field commutative (com), is a dependency function known (dep)?
- is user input independence (in) given?

Notice, a transaction is independent from the user input if a replay does not require any further user input. That is, the user's intention is to execute the transaction despite an existing conflict with the same input. In case a complete transaction is about to be replayed, this property must hold for each of the transaction's operations. In case a single operation is replayed only, this property must hold for the specific operation only.

Based on these properties five classes are defined (see Table III for an overview) where each class has a certain CC mechanism.

The second class "Reconcilable with constraint RC" introduces a conflict probability $P(X)$ and a threshold th. This is motivated by the consideration that to request a guarantee for an "Escrow" field leads to some overhead, which is only required if $P(X)$ is too high for validation to succeed. In case $P(X)$ is low, reconciliation should be used to reduce this overhead and to better comply with an optimistic nature. A definition for $P(X)$ is part of future work.

Eventually, Definition III.10 re-defines Definition III.1 and considers the different data classes.

DEFINITION III.10: (Disconnected, Flat Transaction II)

1) Let R, RC, NRE, NRO, NRL be data classes as defined in Table III
2) Let ta be a flat transaction that is defined as a triplet $ta = (op, <, u)$ where op is a finite set of steps of the form $r(x)$ or $w(x), x \in \{R, RC, NRE, NRO, NRL\}$. And $< \subseteq op \times op$ is a partial order, and u denotes the user input.
3) A transaction is either in its reading (p1), disconnected and working (p2), validating (p3), or writing (p4) phase.

Definition III.10 makes the semantics of a write operation not explicit. The reason is that we aim for a classification based on data.

The issue with the last definition is that ta now technically becomes a composite transaction of the form TA with children

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

$ta_R, ta_{RC}, ta_{NRE}, ta_{NRO}, ta_{NRL}$ since to use a different CC mechanism means to divide a transaction into a maximum of five separate transactions ($ta_R, \ldots, ta_{NRL}$). The operator of the satisfaction sf must be the $\wedge$ operator. In case data of only one data class is modified the additional composite can be omitted.

For the remainder, we continue to refer to the composite with children $ta_R, \ldots, ta_{NRL}$ as ta and other composites as TA.

E. Consistency

To ensure serializability for lengthy or disconnected processing validation is required. In locking scheme concurrency control, Rigorous 2PL [4], [5], [25] ensures serializability but since locking is not an option for classes R,RC,NRE, and NRO an equivalent mechanism guaranteeing serializability is required. Therefore, for R,RC, and NRO an optimistic validation [14] needs to be performed to test whether a set of modifications (write-set) of a transaction intersects with a write-set or read-set of other concurrently executed transactions. In case the write-set of a validating transaction intersects with the read- or write-set of a concurrent transaction, one of the pairwise conflicting transactions has to be aborted. Such a validation ensures conflict serializability (CSR) [4][4]. Reformulated the aforementioned means: if the write-set $WSet_i$ (data modified by a transaction i) intersects with the write-set WS_j or read-set RS_j (data read by a transaction) of another transaction, a conflict is present and one of the pairwise conflicting transactions should be backed out. Hence, to avoid intersections between two transactions ensures CSR. Following this observation a validation must ensure: $WS_i \cap RS_j = \emptyset \wedge WS_i \cap WS_j = \emptyset$.

Classes R and RC, however, have an interesting property and in case validation fails, reconciliation resolves the conflict and ensures semantic correctness. An algorithm for reconciliation is described in [6].

Transactions modifying escrow data NRE have to request guarantees during their read phase. Validation as discussed above is not required because the guarantee ensures consistency already. So, during the validation phase it just has to be ensured that a guarantee has been granted during the read phase. For further details it is referred to [21]. For class NRL it is referred to [25].

Before defining a validation schema, it is important to briefly discuss the interleaving of phases. During the validation of one transaction, a consistent view on the data is required. If during the validation the data changes, validation might be wrong and as a result inconsistencies would arise. Notice, if validations succeeds the data will be written. Therefore, it is usually not allowed that writing and validating data runs concurrently. So a transaction entering its validation phase requires exclusive access to its WS. An algorithm for validation is sketched out below (Figure 4). Notice, this is just a possible algorithm and one variation is to let a transaction wait and not abort in case of an intersection. A brief discussion about OCC and validation is provided in Section IV-A6.

[4] CSR means, two operations op_i of ta_i and op_j of ta_j conflict on the same data item if one of them is a write operation. If there is any cycle in the conflict graph, serializability is no longer possible. Bear in mind that serializability testing is NP complete [4].

Let

1) *dc be a disconnected component*
2) *T_i be the set of transactions to read the data and T_j to write the data.*
3) *$RS_i \subseteq R \cup RC \cup NRO$ be the set of data that is read by T_i*
4) *$WS_j \subseteq R \cup RC \cup NRO$ be the set of modifications.*

If T_j enters the validation it has to perform the following test:

$\forall T_k \mid T_k$ *is in its validation or write phase:*
  if$(RS_i \cap WS_k = \emptyset \wedge WS_j \cap WS_k = \emptyset)$
    $abort\ dc$
  else
  $\forall T_k \mid T_k$ *has committed already*:
    if$(RS_i \cap WS_k = \emptyset \wedge$
        $WS_j \cap WS_k = \emptyset \wedge$ $no\ constraint\ violation)$
      $write$
    else if$(RS_i \cap WS_k = \emptyset \wedge$
        $WS_j \cap WS_k \neq \emptyset \wedge$ $no\ constraint\ violation \wedge$
        $WS_j \subseteq R \cup RC)$
      $reconcile$
    else
      $abort$
    end if
  end for
  end if
end for

Fig. 4: Algorithm for validation

Owing to the recursive nature of our model, consistency of a composite TA is only successful if all children pass validation.

F. Isolation

Since transactions are composed together and the model allows for partial commits of a T if a compensation exists, other transactions may read a state that is invalidated by a compensation later. Such a situation can lead to a cascading behaviour of compensation or even worse, inconsistencies can result. In advanced transaction management, the concepts of closeness and its opposite openness describe the visibility of results. If a transaction's result is passed to its parent only a "Closed Nested Transaction" [26] is given, if also siblings, or even all unrelated transactions are allowed to read results, if the global outcome has not been determined yet an "Open Nested Transaction" [20], [27] is given. Both the open and closed transaction model have been subject of considerable research (see Section IV).

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE III: Data classes ("Reconciliation" refers to [6], "Escrow" to [21].)

	Class	Condition	recommended CC mechanism
1	Reconcilable R	$dep \wedge in \wedge com \wedge \neg cons \wedge num$	Reconciliation
2	Reconcilable with constraint RC	$dep \wedge in \wedge com \wedge cons \wedge num$	$P(X) < th$: Reconciliation, $P(X) \geq th$: Escrow
	Non-reconcilable NR	$(\neg dep \vee \neg in)$	
3	Non-reconcilable Escrow NRE	$(\neg dep \vee \neg in) \wedge com \vee cons \wedge num$	Escrow
4	Non-reconcilable OCC NRO	$(\neg dep \vee \neg in) \wedge (\neg num \vee \neg com) \vee cons$	OCC
5	Non-reconcilable Lock NRL	all NR where a lock is semantically required (justified)	Strict 2PL

The focus in the following paragraphs is on the relationship between the data classes and openness and closeness, respectively. We believe it would be beneficial to exploit the information given by a data class and to determine to use either a closed or an open isolation for a transaction. Usually, this is up to the application developer and may lead to unexpected inconsistencies. With this information in hand, however, a transaction manager could provide support accordingly. Another focus is to incorporate the results of [1] into this work.

For R and RC data, it is possible to use an open isolation. The state of data at read time does not affect the commit, thus, neither rollback nor a compensation affects the commit. Hence, for data classes R and RC isolation is not a real concern and transactions operating on R or RC data only, can release their results immediately and choose an open isolation. The compensation for reconciliation is defined by the inverse of its dependency function.

For escrow data NRE, it is irrelevant for other transactions if a transaction rolls back or compensates. To roll back means to recall the granted guarantee and the worst thing that could happen is that another transaction is not able to get a guarantee because this recalled guarantee denied a guarantee to another transaction. An inverse operation is also determinable for escrow data. Hence, transactions operating on NRE data only can release their results immediately and use an open isolation.

For NRO data, the situation is difficult and the isolation depends on the use case. Moreover, the conflict rate of an entity may be different depending on the time or location. And of course, if no compensation is definable a closed model is required.

For NRL data, where locking is justified due to functional requirements, a closed model should be used. Locking in this model is justified to prevent other transactions from reading until the transaction has terminated. So, pending results should also be protected. Assumed NRL data is pivotal for inconsistencies or cascading compensation, the requirements based justification to use locks is rather questionable. Recall, the motivation for locks in the above discussion was to support an owner role.

The next step is to incorporate the findings of [1] concerning openness and closeness into this work.

In [1], a closed nested transaction is used for all transactions that are executed against the database layer. Hence also for data of class R, RC, and NRE (depending on the use case for NRO possibly too), which is according to the last deliberations not required and a mixed isolation could be applied instead. For example, a transactions executes one transaction t_{NRE} to write the booking of a flight where entity A represents the available seats and is classified as NRE. And, another t_{NRO} to add the actual booking reflected by an entity B classified as NRO. Mixed isolation thereby means that t_{NRE} can unilaterally commit and in case of an abort needs to run a compensation. Transaction t_{NRO}, however, needs to await the global outcome to finally commit.

For the composition of software components, [1] suggests an open model. This complies with standards like the Business Activity protocol [28] designed for workflow support and specifically based on the ideas of the open nested transaction model. The composition of software components to construct new applications requires flexible transactional support and must cope with a long living nature and hence the compensation of transactions. This model adopts a utilisation of the open model at the application level. In case a disconnected component defines an open isolation but processes NRL data (or NRO with a high potential for conflicts), the transaction manager is able to take action and could either set the isolation to close automatically or just inform the developer about the potential risk.

Eventually, we adopt the findings of [1] with the exception to allow for a mixed isolation for composite transactions whose children run against the database layer.

IV. Literature Review and Related Work

Due to the amount of work that has been carried out in transaction management, a rather large amount of existing work relates to ours.

A. Literature Review

1) Nested Transaction Models: The "Nested Transaction Model" [26] was an influential extension of the flat transaction model and a transaction is modelled as a set of recursively defined sub-transactions resulting in a tree of transactions, where leafs are flat transactions representing data operations. In the nested model a child transaction is only allowed to start when the parent has started and a parent in turn can only terminate if all its children have terminated. If a child fails the parent can initiate alternatives, a so called contingency sub-transaction. However, if the parent transaction aborts all its

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

children are obligated to also abort. This in turn requires to rollback already committed results.

An extension to the nested transaction model was developed by Weikum and Schenk [27] who introduced the "Open Nested Transaction Model" that allowed, in contrast to Moss's model, other concurrent sub-transactions to read pending results. In Moss's model only a parent is allowed to read the result of its children, however, in the open version other concurrent sub-transactions are also allowed to read committed results. The results of a child transaction are only durable after the commit of the parent. To prevent inconsistencies only those sub-transactions that commute with the committed ones are allowed to read their results. Read transactions commute for instance.

From an implementation point of view, the nested transaction model can be emulated by savepoints, furthermore the model is a generalisation of savepoints (see [25], Chapter 4.7). Gray and Reuter also discuss the distinction between nested and distributed transactions and one difference is the nested structure is determined by the functional decomposition of the application, hence how the application views a "Sphere of Control" [18]. The structure of a distributed transaction is determined by the distribution of the data. For example, if a transaction must join two tables each stored at a different node, then of course, the transaction must access both nodes and can be modelled and executed as a nested transaction, but the dependencies are different. In the open variant nested model a transaction can commit and abort independently, whereas a commit or abort in the distributed model always depends on each other and only one outcome is possible.

The closed and open nested transaction model can be seen as the seminal work concerning Advanced, Workflow, or Business Transaction Models (see [7], [16], [17]).

An important transaction model that provides a formal correctness criteria for compensation is the Sagas model [12]. A Saga is a transaction that consists of a set of ACID sub-transactions, however with a predefined execution order, and further a set of corresponding compensating sub-transactions must be defined. Notice a compensation transaction is mandatory for each sub-transaction. A Saga completes successfully if either each sub-transaction has committed or the corresponding compensation sub-transaction commits. Generally, a Saga transaction differs from a Chained Transaction [25], where already committed results cannot be undone, but especially this is important to fit the requirements for long-lived transactions. Moreover, a compensation transaction allows for a relaxation of full isolation, also atomicity, and increases inter-transaction concurrency. Locks can be released as soon as a (sub-)transaction commits, even if the parent is still active because the rollback is performed by a separate compensation transaction which does not affect the serializability. And, since each transaction in the Sagas model must define a compensation transaction, also the cascading of errors can be handled, at least theoretically. An extension of the Sagas model is the "Nested Sagas" model [29] that provides mechanisms to structure steps involved within a long running transaction into hierarchical transaction structures. However, one drawback of each model that heavily applies compensation is the requirement for a compensation operation and as soon as non compensatable transactions exist, compensation is not feasible.

In this section, some important aspects of nested transactions and the compensation have been briefly explained. From a historical view, nested transaction models and the so called "Advanced Transaction Models" (ATM) [16] were the foundation for a new generation of transaction models, so-called workflow transaction models [30]. Advanced Transaction Model are sometimes claimed to be less general and more application specific compared to the ACID model. Workflow transaction models do not meet this criticism to the same extent.

Compared to a nested transaction where usually only leaf nodes represent data operations, within a workflow transaction model each node can modify data. A more general definition is: the flat transaction model has evolved vertically to transaction trees, whereas workflows or generally long-computations represent also a horizontally evolution. Both workflow transaction models and ATM have been rarely implemented in the database layer, rather they have been applied in transaction coordination protocols at the middleware level.

2) ConTracts: Another important transaction model for long-computations is the ConTracts model, introduced by [31] and revised in [32]. This is a *conceptual framework for the reliable execution of long-lived computations in a distributed environment.* The core module of the ConTracts model is the ConTract script that describes a long-lived computation similar to a workflow model. The steps involved within this computation are not single statements but represent programs, methods, or applications, which can be invoked through a call interface. In the ConTracts model, the application is responsible for what happens inside a step, and the ConTract script is responsible for keeping the control flow alive between the involved steps, i.e., applications. Similar to the Saga model, each step must define its compensation step to relax isolation, reduce blocking time and thereby increase parallelism. In addition, each step must define an pre- and post-condition (the ConTracts model calls pre- and post-condition entry- and exit-invariant. So, the scope of an in-variant is the step. However, we believe that these invariants are actually conditions as the scope of an invariant should be the governing application. That is, an invariant needs to be true over the entire computation). The pre-condition must hold (validate to true) before the step can be invoked. For instance, an pre-condition can check if the data required by the step is locked. So while compensation facilitates non-blocking, the pre-condition can ensure non-blocking. Beside the pre-condition an post-condition must also validate to true before control can pass to the next step. The concept of post-conditions allows other steps executed in the future to step back, thus an post-condition can be part of another pre-condition. The ConTracts model provides its own definitions of transactional properties and recovery, and to avoid the shortcomings of the atomicity property a two-layered recovery mechanism has been introduced. Recovery

and Serializability are derived from the classic serializability and strictness and the notion of "Prefix Reducibility" [4]. The difference, however, is, since ConTracts are not isolated, the model introduces conditions that define the structural dependencies among the different steps. Such a definition however is not as straightforward as for conflicts between write operations within a schedule. Not only the data an operation accesses must be considered, but the semantics of the whole step must be considered to determine any conflict with other activities.

Regarding recovery, the model distinguishes between recovery at the step and at the script level. Recovery at the script level is important to keep the overall computation consistent, which means, after a failure of a step occurs other steps active during this time need to be recovered, too; this so called forward recovery is compensation.

One strength of the ConTract model is its precise definitions for compensation. The general problem of compensation is that if compensation of a step is required no other concurrent transaction should violate the compensation itself. This leads to the definition of "indirect compensatability", "indirect compensation chain", and "absolute compensatability". Indirect compensatability defines that within an ordered execution of two steps $s1$ and $s2$ with their corresponding compensation steps $cs1$ and $cs2$ the compensation must follow the same order to maintain the consistency. The indirect compensation chain then consists of *"all steps for which the indirect compensation relation holds."*. Absolute compensatability defines a compensation step that is independent of any other compensation step, thus, even an arbitrary execution leads to a compensated result. The ConTracts model makes no further statements how far a running workflow must be rolled back. Based on the definitions provided within the model a complete roll back is foreseen. The partial rollback of workflows is particularly addressed in Leymann's concept of "spheres of joint compensation" [19].

Leymann's work [19] basically is built upon the concept of spheres of control, Sagas, and the ConTract model. The novelty of his concept is to explicitly enable a partial rollback of an active workflow and not to rollback the entire workflow, or more generally: the entire affected tree of transactions. This reflects more the real situation of long-lived computations in which not all work must be undone.

The idea is to define a sphere as: *"any collection of activities (steps) of a process is called Sphere of Joint Compensation (sphere) if either all activities have run syntactically successful or a all activities must have compensated"* [19]. The compensation itself is modelled by adding a compensation activity to each sphere and activity.

The compensation itself is basically performed by the execution of all compensation steps of each activity in reverse order. A composed activity, that is an activity consisting of other activities, can request a shallow or a deep compensation and in the latter case all associated compensation operations of the composed activities have to be executed too. If a shallow compensation is requested only the compensation associated with the root activity is executed. An integral compensation only performs the compensation associated with the sphere and running the compensation of each encompassed activity is referred to as discrete compensation. Leymann also provides compensation modes that define how compensation can impact the neighbourhood of the activity or sphere, the so called "Proliferation Property". When an activity must be compensated the proliferation property defines if the compensation of the whole sphere must be executed, too. This, in turn, must validate the integral property and either the sphere's compensation must be executed only, or the compensation of each activity. Since spheres can overlap, the model provides a definition of how to treat cascading compensation. Generally, this compensation model can be seen as very complete beside the general drawbacks of compensation discussed in the next section, and its concepts have been considered by the Business Process Execution Language (BPEL) and Business Process Modelling Notation (BPMN) standard.

3) Concurrency Control for Transactional Processes: The theoretical framework by Schuldt et al. [33] to reason about concurrency control and recovery in transaction processes is an attempt to unify the theory of concurrency control and recovery for transactional business processes (processes) or workflows. Schuldt et al. argue that the challenge we face is to design a single correctness criterion for both concurrency control and recovery that also copes with the added structure found in processes. They further observed that the flow of control introduced by processes is one of the basic semantic elements, and that a correct execution must obey the already existing ordering constraints among their different operations and alternative executions. These constraints determine how activities of the process can be interleaved during execution. They further state that the different atomicity properties among the involved systems can not always fit the strong requirements of models applying compensation, e.g. ConTracts, where each operation requires its inverse because it is not guaranteed to find an inverse.

Similar to other models, they extend the notion of atomicity by considering also alternatives or a partial rollback of already executed steps within the process. In practice, tasks are often executed in parallel to increase the time to market, and regarding concurrency control without considering recovery an ordering of these tasks is sufficient, however if recovery is taken into account, and for one of the steps no compensation exists, the situation becomes different. Their example is a production and a corresponding test within a manufacture where the production usually has no inverse function (at least no acceptable one). Thus, the production is only allowed to start if the test terminated successfully because a concurrent execution can lead, in the case of compensating the test, to an invalid production if both are executed concurrently.

The general point they address is more how to maintain correctness if no compensation is given. What follows in their paper is a theoretical model for correct process schedulers basically oriented on the Flex Model [34], [35]. The Flex model introduces, beside the notion of a compensatable sub-

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

transaction, a "retriable" sub-transaction that can be retried and eventually succeeds if retried a sufficient number of times, and a "pivot" sub-transaction that is neither retriable nor compensatable. Concerning the details of the framework and its definition for correct schedules, a more general explanation is provided here. At the database level the serializability of operations can be expressed by their conflict relationship, and similar a conflict relationship is defined between activities, however, at the process level. But, since the internals of an operation are usually not exposed by an activity the whole activity needs to be classified to define its conflict relationship to other activities. Based on the classes of retriable, compensatable and pivot sub-transactions, activities are classified and a commutativity, a compensation and a effect-free activity rule can be expressed. These rules in turn can be exploited by a process task scheduler to produce a correct process schedule. Summarising, their framework is based on the Flex transaction model and provides the theoretical foundation to ensure and reason about the correctness of process schedules by considering both, concurrency control and recovery within one model.

4) Web Services and Transactions: A model for the distributed management of concurrent Web service transactions is introduced in [36]. Alrifai et al. claim that in the *"open and dynamic Web service environment, business transactions enter and exit the system independently and under relaxed isolation transactional dependencies can emerge among independent business processes which must be taken into account when compensation is required in order to avoid inconsistency problems."* Within a closed environment inconsistencies can be controlled more easily because the dependent transactions are known. Their work combines an optimistic decentralised variant of the SGT (Serialization Graph Testing) protocol that applies an Edge Chasing algorithm to detect potential global waiting cycles with a transaction scheduling algorithm that selects the service provider based on their scheduling offers. The architecture foresees a multi-layered architecture consisting of a process, a Web service and a resource level and applies the Multilevel Nested Transaction Model [4], [27] where each leaf has the same distance to the root. Conflicts at the resource level are detected by a separate resource-level concurrency control module, at the service level a service-level concurrency control module is responsible to detect conflicts. Conflicts are usually defined by conflict relations expressed as a transactional dependency graph in their model and global consistency is ensured if each local system guarantees local consistency as in distributed database systems (see commit-order and rigorous scheduling [4]). They apply a 2PC as atomic commitment protocol.

Their model extends the OASIS Web-Service standards WS-Coordination [37], WS-Atomic Transaction [38], and WS-Business Activity [39] (see section IV-A5). However, their Multilevel Nested Transaction Model only allows commutative concurrent sub-transactions to read pending results. To define the commutativity of transactions each transaction type is divided into atomic steps with compatibility sets according to its semantics. Transaction types that are incompatible and are not allowed to interleave at all. Farrag and Özsu [40] already refined this method by allowing some interleaving for incompatible types and assuming fewer restrictions for compatibility. The problem is that finding the compatibility sets for each transaction step is a $\mathcal{O}(n^2)$ problem. Alrifai et al. apply a conflict matrix that defines the conflict sets of a transaction and hence their approach must deal with a quadratic complexity too, even their conflict predicates do not solve the problem because their purpose is only to enable a conflict detection across autonomous and independent systems.

5) Web Service Transaction Standards: This section briefly explains and mentions some specification for protocols that allow for distributed transaction processing in a XML Web service (WS) architecture. The reason for such a brief explanation is that the specifications are based on the ideas of the nested and long running transactions, which have been discussed already (see section IV-A1).

The so called "WS Transactions specifications" are the "WS Coordination" [37], the "WS Atomic Transaction" [38], and the "WS Business Activity" [39] specification. In terms of a WS architecture, they relate to the Quality of Service (QoS) layer and all these specifications are built on top of the Simple Object Access Protocol (SOAP) and Web Service Description Language (WSDL) standards.

The core element is the WS-Coordination (WS-CO) framework that is an *"extensible framework for providing protocols that coordinate the actions of distributed transactions"* [37]. The framework allows for a mechanism to register partners and to allow for a generic control of their interaction. If a transactional conversation is required an additional framework that provides a specific atomic commitment protocol, a so called completion protocol, must be plugged into the WS-CO framework. The WS Atomic Transaction or the WS Business Activity framework are such frameworks that provide completion protocols.

The WS Atomic Transaction (WS-AT) framework provides two completion protocols namely a volatile 2PC and a durable 2PC. Whereas the first one is intended for services that operate on volatile, i.e., non persistent data, the second one is, as the name suggests, for services operating on persistent data. Both protocols can be used within the same transaction. However, in such a case the volatile services must complete before the durable ones. Beside the extensions required to comply with the WS specification family, the WS-AT specification addresses the well-known 2PC as introduced by "The Open Group" [41] and allows for distributed "all-or-nothing" transactions.

In contrast to the all-or-nothing principle of WS-AT, WS Business Activity (WS-BA) allows for hierarchical nested scopes possibly requiring compensation, relaxed isolation, autonomous participants, or abort autonomy for instance. Generally, WS-BA is a specification for the management of nested transaction and a so called "mixed outcome" of a transaction is possible; for example some transactions terminate committed, some aborted, and others compensated.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

In summary, the WS transaction specifications, especially the WS-BA is an example for a technology that is able to cope with nested and complex transactions and for more details on WS-CO, WS-AT or the WS-BA it is referred to the specifications.

6) Optimistic Concurrency Control: In the early eighties, Kung [13] introduced OCC, which has not gained as much consideration as CC protocol in commercial database systems as locking has. The PyrrhoDB [42] is one database we are aware of that implements OCC as CC mechanism.

Härder [14] sees the challenge in merging the workspace of a transaction with the actual state of the database. *"Their essential problem consists of merging these copies during COMMIT processing thereby regaining a transaction-consistent database image"*. This asynchronism is especially challenging *"when these copies do not match with the units of transfer (pages)"* [14] and when different copies of the database have to be kept consistent. Notice, Härder's critique about the merging of copies seems obsolete and outdated as Multi-Version Concurrency Control protocols as widely used by Oracle or PostgreSQL, for example, are subject to the same problem of merging copies.

Härder goes even further and states that even if OCC has been defined for applications where conflicts are unlikely, *"locking also behaves quite well in such a particular environment (no wait or deadlock conflicts), there seems to be little reason to introduce a specialised control mechanism"*. Härder refers to an empirical comparison that shows that with OCC the abort rate of transactions is higher compared to locking. This is due to the property that in locking a transaction waits rather than restarts. In this case the restart of a transaction must be considered as being equivalent to an abort. In other words, as stated by Härder, to wait increases the probability for success. It is worthwhile to discuss this issue in a little bit more detail here. One particular problem with OCC is that it is not possible to get any guarantees in advance. "First come, first served" is the result of the optimistic behaviour. By contrast, in locking a sophisticated locking schema can guarantee a transaction that updates will succeed if constraints are not violated, and in the absence of physical errors. Hence, if the update rate on a field is high, the abort rate of locking is less than with OCC, but for the price that a transaction has to wait. In this sense waiting is not wasting.

Franaszek et al. [23] come up with the idea to investigate *"the potential reduction in the required concurrency level via the use of what might be viewed as the pre-fetch property of transactions which run but do not commit."* These transactions are said to run in a "virtual execution mode". Particularly this means a transaction is executed twice. The first execution is to determine the required data, calculate the access paths, and load the data into the cache. The second execution is to actually commit the transaction with all the data in cache already and a lock pre-climbing that requests all locks in one atomic step, which is feasible since the entire data set is known already. So, the idea of Franaszek's et al.'s mechanism is not to restart a transaction even though it is known that the transaction will fail at commit time and to benefit during the second execution from the information gathered at the first run instead. One of the most important considerations thereby is the notion of "Access Invariance" that is a transaction will *"with high probability perform the same operations on the same subset of objects without regard to the implied serial order of execution."* This assumption is adequate as it would mean the correctness notion of serializability would be inadequate else. The authors emphasise that there can be indeed conflicts between transactions or constraints that might permit transactions from commit, but neither of them does affect the access invariance in general.

Despite the rather sceptical note by Härder as well as by Mohan [43] and the fact that OCC has not been implemented by many commercial database vendors as a basic CC mechanism, OCC fits well into a disconnected computing architecture where consistency preserving mechanisms are required as part of a Middleware solutions (see [44]) Laux et al. [45] thoroughly analyse Row Version Verification (RVV) as an implementation of OCC if the database does not support "optimistic locking" per se and their patterns to implement RVV for some common databases and data access technologies at the MW layer fits well into a disconnected architecture.

B. Related Work

As presented in the last section, there are many transaction models that consider a nested and recursive transaction structure. The ACTA framework [46], [47] provides an independent language to describe these complex transaction models by demarcating aspects of atomicity and isolation. Its drawback is the large terminology and the missing execution semantics. Its nature is purely descriptive. Eventually, beside the aforementioned transaction models and their well understood concepts, the ACTA framework also inspired our work. However, we believe that to use Boolean expressions to describe transactional dependencies is easier to comprehend and reduces the large terminology, which can be found in ACTA. And, a satisfaction expressions is computable.

The second key piece of this work, the classification of data, is inspired by [3] and [6]. Kraska et al. use statistical measures to determine a conflict probability and adapt the CC mechanism accordingly. Since their work focuses on the Cloud, replication plays an important role too in their model. By contrast, this work does not consider replication. As described in Section III-D, the data classes are according to Laux and Lessner's work on Reconciliation [6] and ONeil's work [21] on the Escrow data type. Furthermore, we are specifically interested in the question how the classficiation of data can be exploited also concerning the isolation property.

Many domains, for example, object orientation, SOC, or Mobile Computing are confronted with disconnected situations if an increased local autonomy is required. The assumption in this work, that there is actually no difference, and all software components should just run in a disconnected mode with separate read and write phase. This is inspired by work on OCC [13], [14], [22], [23], [45] and to divide a

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

component into such phases is also similar to the Command Query Pattern [10], however, in a larger scale.

V. Conclusion, Contribution and Future Work

The contribution of this work is an expression language for transactional dependencies if transactions are composed together. An expression is based on Boolean algebra and can be used by a transaction manager to coordinate the transactional composition because the execution semantics is derivable.

Moreover, since we believe that an application of a CC mechanism should consider the semantics of data and operations to better trade-off the consistency needs and incurring costs, and not follow the one CC mechanism fits all needs paradigm, we have introduced five different data classes using a certain CC mechanism. In this context we have also analysed the implications for isolation in a nested transaction. Moreover, our classification complies with an optimistic attitude, which in turn complies with the needs of current disconnected and asynchronous computing architectures.

To impose a phase structure on disconnected components and analyse the implications on transaction management is another contribution.

Our goal in the long term is to allow for even better trade-offs similar to [3]. In our vision applications should express their transactional requirements like the required level of consistency, processing time, and costs. Due to the composition this also requires a model that copes with the composition of requirements, raising the needs for metrics. The classification of data and a language to express transactional dependencies are first steps. For the future, we also envision to run transactions in different lanes according to their semantics and requirements. Such an allocation and division could help to easier verify and trade-off scalability and consistency demands.

Simulation results that justify a classification of data are probably the most missing piece in this work. Currently, a prototypical transaction simulation and reasoning framework is still under development.

References

[1] Tim Lessner, Fritz Laux, Thomas Connolly, Cherif Branki, Malcolm Crowe, and Martti Laiho. An optimistic transaction model for a disconnected integration architecture. In *DBKDA 2011, The Third International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 186 – 191, 2011.

[2] Jim Gray and Pat Helland and Patrick E. O'Neil and Dennis Shasha. The Dangers of Replication and a Solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press, 1996.

[3] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proc. VLDB Endow.*, 2:253–264, August 2009.

[4] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[5] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[6] Fritz Laux and Tim Lessner. Escrow Serializability and Reconciliation in Mobile Computing using Semantic Properties. *International Journal On Advances in Telecommunications*, 2(2):72–87, 2009.

[7] Ahmed Elmagarmid, Marek Rusinkiewicz, and Amit Sheth, editors. *Management of heterogeneous and autonomous database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] Margaret H. Dunham, Abdelsalam Helal, and Santosh Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mob. Netw. Appl.*, 2(2):149–162, 1997.

[9] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[11] Michael Beisiegel et al. Service component architecture (sca) v1.00, 2010-11-08, 2007.

[12] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM, 1987.

[13] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[14] Theo Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9:111–120, November 1984.

[15] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *SRDS '95: Proceedings of the 14TH Symposium on Reliable Distributed Systems*, page 31. IEEE Computer Society, 1995.

[16] Ahmed K. Elmagarmid, editor. *Database transaction models for advanced applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[17] Sushil Jajodia and Larry Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

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

[19] Frank Leymann. Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems. In *BTW*, pages 51–70, 1995.

[20] Alejandro Buchmann, M. Tamer Özsu, Mark Hornick, Dimitrios Georgakopoulos, and Frank A. Manola. *A transaction model for active distributed object systems*, pages 123–158. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[21] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions On Database Systems*, 11:405–430, December 1986.

[22] P.A. Franaszek, J.T. Robinson, and A. Thomasian. Access invariance and its use in high contention environments. In *Data Engineering, 1990. Proceedings. Sixth International Conference on*, pages 47 –55, February 1990.

[23] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *Knowledge and Data Engineering, IEEE Transactions on*, 10(1):173 –189, jan/feb 1998.

[24] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3:70–80, September 2010.

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

[26] J. Eliot B. Moss. *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[27] Gerhard Weikum and Hans-Jörg Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[28] Web services business activity (ws-businessactivity) version 1.1. Online (accessed 15.08.2011), July 2007.

[29] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *Data Eng.*, 14(1):14–18, 1991.

[30] Ting Wang, Jochem Vonk, Benedikt Kratz, and Paul Grefen. A survey on the history of transaction management: from flat to grid transactions. *Distrib. Parallel Databases*, 23(3):235–270, 2008.

[31] Helmut Wächter and Andreas Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[32] Andreas Reuter and Kerstin Schneider and Friedemann Schwenkreis. ConTracts Revisited. In Sushil Jajodia and Larry Kerschberg, editors, *Advanced Transaction Models and Architectures*. 1997.

[33] Heiko Schuldt and Gustavo Alonso and Hans-Jörg Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 316–326. ACM Press, 1999.

[34] Sharad Mehrotra and Rajeev Rastogi and Henry F. Korth and Abraham Silberschatz. A Transaction Model for Multidatabase Systems. In *ICDCS*, pages 56–63, 1992.

[35] Ahmed K. Elmagarmid and Yungho Leu and Witold Litwin and Marek Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In Dennis McLeod and Ron Sacks-Davis and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 507–518. Morgan Kaufmann, 1990.

[36] Mohammad Alrifai and Peter Dolog and Wolf-Tilo Balke and Wolfgang Nejdl. Distributed Management of Concurrent Web Service Transactions. *IEEE T. Services Computing*, 2(4):289–302, 2009.

[37] Oasis. Web Services Coordination (WS-Coordination), 2009.

[38] Oasis. OASIS Web Services Atomic Transaction Version 1.2, 2009.

[39] Oasis. OASIS Web Services Business Activity Version 1.2, 2009.

[40] Abdel Aziz Farrag and M. Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, 1989.

[41] The Open Group. Distributed transaction processing: The XA specification, 1992.

[42] Malcolm Crowe (University of the West of Scotland). The Pyrrho database management system (http://www.pyrrhodb.com/, 2010-11-02), 2010.

[43] Mohan. Less optimism about optimistic concurrency control. In *Proceedings of the 2nd International Workshop On Research Issues In Data Engineering*, pages 199–204, 1992.

[44] Philip Bernstein and Eric Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[45] Fritz Laux, Martti Laiho, and Tim Lessner. Implementing row version verification for persistence middleware using sql access patterns. *International Journal on Advances in Software, issn 1942-2628*, 3(3 & 4):407 – 423, 2010.

[46] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: a framework for specifying and reasoning about transaction structure and behavior. *SIGMOD Rec.*, 19(2):194–203, 1990.

[47] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using acta. *ACM Trans. Database Syst.*, 19:450–491, September 1994.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org