Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

Wijnand Derks

Ph.D. defense committee:

Prof. dr. W. Jonker, University of Twente, Enschede, Netherlands (promotor)
Prof. dr. ir. A.J. Mouthaan, University of Twente, Enschede, Netherlands
Prof. dr. P.M.G. Apers, University of Twente, Enschede, Netherlands
Prof. ir. R. Pieper, University of Twente, Enschede, Netherlands
Prof. dr. K. Aberer, EPFL, Lausanne, Switzerland
Prof. dr. G. Alonso, ETH, Zurich, Switzerland
Prof. dr. H. Schuldt, UMIT, Hall, Austria

CTIT Ph.D.-thesis Series No. 05-75
Centre for Telematics and Information Technology (CTIT)
P.O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series No. 2005-21
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN: 90-365-2278-1
ISSN: 1381-3617

Printed by: PrintPartners Ipskamp

IMPROVING CONCURRENCY AND RECOVERY IN DATABASE SYSTEMS BY EXPLOITING APPLICATION SEMANTICS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 16 november 2005 om 15.00 uur

door

Wilhelmus Lambertus Adrianus Derks

geboren op 13 januari 1973
te Beers

This dissertation has been approved by:

Prof. dr. Willem Jonker

Preface

After completing my MSc. in computer science, I was eager to learn about data management challenges in real-life. My first employer, KPN Telecom, provided an excellent environment in which to learn about the complexity of data management in a telecommunications environment.

The research department of KPN Telecom provided an excellent base from which to explore the business driven challenges of data management in a large enterprise. I conducted my first research projects in the database group of Prof. dr. Willem Jonker. This was a very stimulating time where my interest in data management matured. One of the most prominent projects here addressed the scaleability problem of very large databases, embedded in the EURESCOM VLDB program [33, 31, 30, 86].

Another challenging problem faced by KPN was (and probably is) business process integration [48]. This problem was addressed in the ESPRIT Crossflow project [43], which developed support for cross-organizational workflow management. I participated in this project in close co-operation with Prof. dr. ir. Paul Grefen to develop transaction management support for business processes [27]. The transaction model developed in this project is simple though powerful for many applications [78]. It was this experience that inspired me to develop a more general transaction management theory that would improve concurrency and recovery by exploiting application semantics. Initial work appeared in [25, 26, 28, 29, 32] which eventually evolved into this dissertation.

Writing a dissertation is a process that takes years. I have learnt that such processes can only be completed if it is supported by people that understand long term research. For this, I would like to thank my promotor Willem Jonker for his efforts to create space and time for my research and his willingness to follow my quest into the more fundamental corners of transaction management theory. In addition, I would like to thank Prof. dr. Peter Apers for providing me with the opportunity to conduct this research in his group. I am especially grateful for the very flexible working conditions. Also, many thanks go to Paul Grefen who encouraged me to do research in transaction management and always had time to answer urgent questions and provide me with valuable advice.

Also, many thanks go to Prof. dr. Hans-Jörg Schek to receive me at the ETH Zürich to discuss preliminary thoughts on my work. In particular, I would like to thank Prof. dr. Heiko Schuldt for inviting me to the UMIT in Innsbruck. The months spent at UMIT have provided me with much inspiration and appeared crucial in the completion of the work. In addition, his feedback on early ideas have greatly helped improving the quality of the work.

I would like to thank Prof. dr. Gustavo Alonso, Prof. dr. Karl Aberer, Prof. ir. Roel

Pieper, Prof. dr. Heiko Schuldt, Prof. dr. Peter Apers and Prof. dr. ir. Ton Mouthaan for accepting membership of my Ph.D. defense committee.

Last but not least, I would like to thank my colleagues in the IS and DB group at the University of Twente. Foremost, I would like to thank Maarten Fokkinga for always making time for discussions, and showing genuine interest in my work and ideas. In addition, I would like to thank Jochem Vonk for being a great project co-member that always had time for a coffee-break, and my room-mates Rik Eshuis, David Janssen, Dennis Krukkert and Wiebe Hordijk for creating an excellent working atmosphere. I should not forget to thank Sandra, Suse and Leonie who always gave me the feeling to be home.

True home is however with my wife Ragna who supported me all years through with great patience and encouragment. Her desire to clap hands on the first row was really the best motivation of all.

Wijnand Derks
19 October 2005

Contents

Chapter 1

Introduction

In this thesis we propose an extended transaction management theory. The theory provides a basis for improving concurrency and recovery of transactions over existing approaches, while preserving the correctness of the database. This is achieved by exploiting the semantics of transactional applications.

In this chapter we motivate the need for an extended transaction theory in section 1.1. We provide a brief overview of existing work in section 1.2 and introduce our problem statement in section 1.3. Then, in section 1.4 we introduce our approach. We conclude this chapter with an outline of this thesis in section 1.5.

1.1 Motivation

This section introduces basic notions such as a database, transactional programs and transactions. We motivate the need for transaction management and introduce the particular challenges for long transactions.

1.1.1 Transaction management

A database can be used to model part of the real world that is relevant to its users. Users can query the database to retrieve part of the database state and update the database to reflect changes that occur in the real world. A combination of queries and updates to reflect a change in the real world in the database is referred to as a *transaction*. Users perform transactions on the database through computer programs. We refer to these programs as *transactional programs*.

The state of a database is generally considered *correct*, if it reflects the result of all transactions. In theory, this can be easily achieved by executing the transactions in sequence on the database.
However, in practice transactions are often executed concurrently on the database. This means that a database system must deal with concurrent access to data. Another complication is that database systems and application servers are vulnerable to failures. It is the main goal of transaction management to guarantee that correctness of the database is

preserved, even in the presence of concurrency and failures. We reference the mechanisms to deal with concurrency and failures as *concurrency control* and *recovery*.

The requirement to preserve correctness in the presence of concurrency and failures is captured by means of the *ACID properties for transactions*. As long as the execution of each transaction is ACID, correctness of the database is preserved [46]. The ACID properties include:

- *atomicity:* either all results of each transaction should appear in the database, or none,
- *consistency:* each transaction leads from one consistent database state to another,
- *isolation:* concurrent transactions should not interfere with a transaction's execution,
- *durability:* the result of each transaction should be stored persistently.

Guaranteeing the ACID properties in the presence of concurrency and failures is a complex task. Therefore, the ACID properties are generally guarded by a dedicated system component, the transaction manager. The transaction manager is essentially a scheduler that allows operations of the transactional programs to execute on the database in a particular order. In addition, the transaction manager guarantees persistency of results. How operations are scheduled depends on the transaction model that is used by the system.

The most widely adopted transaction model is the flat transaction model. A transaction manager based on the flat transaction model preserves all ACID properties and can be implemented efficiently by a locking protocol and a persistent log [35]. The flat transaction model with the *strict two-phase locking protocol* implementation has had tremendous impact both in theory and in practice. This is witnessed by the fact that it is still the main transaction management technique today and is implemented in almost any commercial database system.

1.1.2 Advanced transaction management

Even though the flat transaction model has been highly successful for many applications, the model has been less successful for long transactions that may execute for days, weeks, or longer. This is mainly due to the fact, that strict two-phase locking blocks resources until the transaction completes and lock contention increases polynomially with the length of transactions [42].

Therefore, since the late eighties enhanced transaction models have been proposed to overcome the limitations of the conventional flat transaction model. Numerous extended transaction models and techniques have been proposed, but almost none are adopted in practice, with the exception of a limited use of the nested transaction model [60].

The most important reason for their limited success is that either performance is inadequate, or correctness guarantees are insufficient, or the model is too complex. Not surprisingly, voices have emerged in recent years to give up research on extended transaction management [19]. It has been argued that it be sufficient to support long running applications by conventional transaction management techniques. Missing transactional requirements should be encoded by the program designer himself.

However, explicit programming of concurrency and failure handling is complex and error-prone. One could even argue that these disadvantages are even the main reason for the success of the conventional transaction model.

The author therefore stresses that research into transaction models and techniques for long running applications is still required. To identify the opportunities for such research, the remainder of this section discusses related work to identify strengths and weaknesses of existing approaches. Based on this analysis, the goal and the approach of this thesis are formulated.

1.2 Existing approaches

We center the discussion of the existing approaches around the ACID properties. For each of the properties we give an overview of the particular problems involved and the most important solutions. We will not go into to much detail in this section. More detailed discussions on the specific problems addressed in this thesis are postponed to the related work sections of the appropriate chapters.

1.2.1 Atomicity

Atomicity of a transaction requires that either all results of the transaction appear in the database, or none. In the conventional transaction model, atomicity of transactions is preserved by recording a before image of each variable that is written, such that upon failure the database state can be restored to its value before transaction start by writing the recorded values back to the database. In addition, no concurrent transaction should observe aborted results, that are generally referred to as *dirty reads*. The problem can be approached optimistically where uncommitted results can be read. This results in higher degrees of concurrency at the expense of cascaded aborts. Alternatively, dirty reads can be prevented, but this comes at the expense of reduced concurrency. The latter approach is adopted in strict two-phase locking, where locks on results are held until transaction commit [21].

Neither approach is satisfactory for long transactions, because cascaded aborts may result in much lost work, and strict locking reduces concurrency considerably for long transactions.

In [41, 37] it was recognized that effects of an aborted transaction can also be erased from the database by executing compensating operations. Each operation is assumed to have a compensating operation with an inverse effect. The effect of an aborted transaction can be removed from the database, by executing the compensating operations of all executed operations in reversed order.

The SAGA model [38] is a typical example of this approach. A SAGA is a sequence of sub-transactions, called steps, where each step is associated with a compensating step. Upon failure, the compensating steps are executed in reverse order to remove the effects of the previous steps from the database. As each step can be arbitrarily interleaved by steps of other transactions, results can be observed that may be later compensated and thus only a weaker interpretation of atomicity is preserved. This weaker notion of atomicity is referred to as *semantic atomicity*.

One problem with this approach is that in general, it cannot be assumed that every effect can be compensated. In these cases, the only way to preserve atomicity after executing a

non-compensatable operation is to guarantee that the transaction completes. For this, the remaining operations of the transaction must succeed. Operations that are guaranteed to succeed after a finite number of retries are referred to as *retriable operations* [52, 57, 58]. Essentially, for atomicity it is only necessary that *some* retriable postfix of a transaction exists after a non-compensatable operation. Such structure is referred to as *well-formed* [70].

These ideas are captured by *flexible transactions* [89, 88]. Flexible transactions are essentially *sets* of transactions, that are ordered with a preference relation. The transaction succeeds, if one of the alternative transactions is able to commit, in order of preference. This type of atomicity is referred to as *semi-atomicity* [89, 88]. Semi-atomicity preserves atomicity of transactions, while allowing for non-compensatable operations.

Another, more theoretic approach to atomicity was taken in [77, 76]. In this work, the *unified theory of concurrency and recovery* was developed, that provides a single model for reasoning about concurrency and recovery. In particular, it is aimed at providing atomicity with compensating operations. Unfortunately, the basic theory does not account for non-compensatable operations. Therefore, the unified theory has recently been extended for non-compensatable operations [70]. However, the proposed model does not exploit knowledge about the forward recovery of transactions, which severely degrades concurrency for transactions with early non-compensatable operations.

1.2.2 Consistency

The consistency requirement means that each transaction transforms a consistent database state into another consistent database state. This is achieved if the transaction manager starts a transaction in a consistent database state and completes the transaction in another consistent database state. Consistency can be verified by the transaction manager by evaluating the database consistency constraints. Consistency can be preserved by removing all effects of those transactions that violate database consistency upon completion. If the effects of such malicious transactions can be erased, database consistency can be preserved if transactions execute serially. Concurrent execution of transactions can also be allowed as long as the interleaved execution is *equivalent* to a serial execution of transactions. Such executions are referred to as *serializable* [21].

The class of view serializable schedules satisfy the requirement that the observations of all transactions and the final database state after execution of the schedule are the same as some serial execution of the same transactions [87]. View serializability however cannot be efficiently implemented [64, 63]. Therefore, a sub-class of view serializable schedules was proposed that can be efficiently implemented: the class of conflict serializable schedules [63]. Conflict serializability is therefore the correctness criterion of choice for many database products. Conflict serializability is generally implemented by two-phase locking [35].

To enlarge concurrency, weaker notions of correctness may be used. The class of final state serializable schedules [64, 22] only requires that the final database state is the same as after some serial execution of transactions. It is not required that transactions observe a consistent database state. This weaker correctness criterion may be acceptable if only serializability with respect to the database is required, and not with respect to values returned to the user.

Another approach to improving concurrency by weakening correctness is predicate-wise serializability [49]. In this approach, it is assumed that the database consistency constraint

can be written in conjunctive normal form. Consistency is preserved by requiring serializability with respect to each conjunct, but the serialization order over each conjunct can be different.

1.2.3 Isolation

Conventional transaction management guarantees complete isolation, which is a sufficient condition to execute correctly. Isolation of transactions is guaranteed in any serializable execution [21]. As mentioned above, serializability is generally implemented by two-phase locking. However, two-phase locking is very inefficient for long transactions, as lock contention grows polynomially with the size of each transaction [42].

For this reason, the ANSI isolation levels were introduced [8]. For isolation levels other than *serializable*, locks are released earlier than two-phase. Although this results in higher degrees of concurrency, full isolation for transactions is lost. Therefore, other levels than serializable do not provide correctness for all applications [12]. Although the problem of identifying the appropriate isolation level for particular applications was addressed in more recent work [18, 54], the approach of isolation levels remains inflexible, because only a single isolation level can be chosen for each transaction, which should always be the most conservative one to ensure correctness.

Another approach to enlarge concurrency is by decomposing transactions into smaller units of isolation. An early proposal for decomposed transactions is the nested transaction model [60], where a transaction is decomposed hierarchically into sub-transactions. Sibling sub-transactions may execute concurrently and are isolated with respect to each other. The model allows for intra-transaction concurrency, while preserving isolation with respect to transactions. In this model, intermediate results of transactions are not visible to concurrent transactions. This way, concurrency at the transaction level is limited.

The limitation is overcome in the multi-level transaction model [81, 82]. Multi-level transactions exploit the higher-level semantics of groups of operations to prove that these operations can execute in arbitrary order without violating isolation. The name multi-level transaction model refers to the property that transactions are nested hierarchically with an equal number of layers. This makes the model especially suitable for application in layered systems. Its generalization to open nested transactions [84] releases the requirement for the layering of the transaction structure. More recently, the theory was further generalized for scheduling of open nested transactions over arbitrary system configurations [5, 4, 65]. These systems are referred to as composite systems [3].

One of the first approaches to relax isolation of transactions is the SAGA model [38]. As discussed, the SAGA model assumes a sequence of isolated steps for transactions. Between steps, no isolation is guaranteed. This property of either full isolation within steps, but no isolation between steps severely limits the applicability of the model. To overcome this limitation, models have been proposed that provide *some* degree of isolation between steps by means of restricting the interleaving of steps. Examples of these include the breakpoint model [56, 36], relative atomicity [1, 75] and the ConTract model [66, 79, 71, 72, 67]. The problem with these approaches is however, that no guidelines are provided what interleaving should be considered correct.

The problem of correctness of interleaving of steps has been addressed by Bernstein et al. [16, 17, 13]. Here, transaction semantics are exploited and a formal method is used to

derive assertions that need to be maintained between the execution of two consecutive steps. Only those concurrently running steps are allowed to interleave the two steps that do not invalidate the assertion. A concurrency control is proposed that allows only the correct interleaving.

1.2.4 Durability

In most transaction models, the commit operation marks the moment at which the transaction has terminated and its results must become durable. The approach where results are made durable only at the end of a transaction is generally undesirable for long transactions, because much work may be lost in case of failure. Therefore several approaches have been proposed in literature to limit the amount of work that is lost upon failure.

A classical model where durability of intermediate results is achieved is the chained transaction model [38, 74]. Chained transactions consist of a sequence of sub-transactions that are immediately committed after completion. Results of completed sub-transactions are therefore durable. However, the use of chained transactions is limited, because neither atomicity, nor isolation is guaranteed for transactions. In addition, for maintaining database consistency each sub-transaction is required to preserve consistency, which severely limits the decomposition.

Savepoints are another approach to improve durability [59]. Savepoints mark the database state at a certain point during execution. A partial rollback may be performed to a savepoint during execution, such that only the work after the savepoint is erased. A savepoint however does not provide durability, because an abort will remove all effects of the transaction from the database [62, 47, 68].

1.3 Problem statement

The previous discussion shows that a lot of work has been done to overcome the limitations of the flat transaction model [21]. Initial attempts focused on refining the notion of correctness without exploiting application semantics. Notable results in this direction include predicate-wise serializability [49] and multi-version serializability [73, 20]. However, it has proven difficult to define a single correctness notion that suits all applications.

Therefore, it seems that the most promising direction is the exploitation of application semantics for improving performance. Notable work in this direction involve the transaction models that are based on interleaving specifications of steps of transactions. Other important work with high practical relevance are the ANSI isolation levels [8]. In both directions significant performance improvements have been reported over the traditional model. However, the main disadvantage of these approaches is that they are not backed by a well-defined correctness criterion. This means that one of the great appeals of transaction management is lost: relieving the application designer from responsibility of concurrency and recovery related problems.

Therefore, we believe that a well-defined correctness criterion is crucial for successful transaction management theory. Therefore, we define the high-level goal of this thesis is as follows.

The goal of this thesis is to develop an integrated transaction model with a well-defined correctness criterion that improves concurrency and recovery over existing approaches by exploiting application semantics.

The question can be decomposed into several smaller questions. The first question to answer is:

What application semantics should be included in the transaction model?

Assuming that the transactional semantics are known, it must be possible to integrate them into a single model. Thus, the following question has to be answered as well:

How can the transactional semantics be integrated into a coherent model?

Transaction management techniques must be capable of handling the transactional semantics, such that concurrency and recovery are improved. This brings up the question:

What transaction management technique is suitable for scheduling?

For suitability of the technique we will consider the performance characteristics of the techniques and the common practice.

1.4 Approach

The main approach to answer the three questions is as follows.

The first question to address is what transactional semantics should be included in the transaction model. Previous work has identified many transactional semantics of applications that have proven relevant for improving concurrency and recovery. The main approach here is to identify these transactional semantics from previous work, and include them in our model.

The second question involves the integration of the transactional semantics into a coherent model. Previous work demonstrates the overwhelming success of serializability theory as a framework in which various transactional properties can be captured [61, 21, 76, 70]. Therefore, we will adopt this model as the basis for our model.

The third question also favors serializability theory as the formal basis. Previous work has demonstrated that it is often possible to derive efficient scheduling protocols from serializability-based theory [35]. It is likely that our serializability-based theory will have such property as well.

We already stressed the importance of a well-defined correctness criterion for transaction management theory. This means, that our model will be based primarily on work that successfully improved concurrency and recovery by exploiting application semantics while preserving a well-defined correctness criterion. In this direction we note the work of closed and open nested transaction model [61, 84], and the step decomposition approach for improving concurrency [13]. Improvement of recovery was achieved by the unifying theory of

concurrency and recovery and its extension to transactional processes [77, 76, 70]. We use this work as a starting point for our approach to recovery.

Next, we discuss our approach in more detail. We develop our theory in a number of steps. In every step, we add transactional semantics to our model to improve either concurrency, recovery or both. In each step, we sketch how we identify the relevant transactional semantics, how the transactional semantics are integrated in our model and why the extended model can still be scheduled efficiently. Our discussion is structured around the ACID properties.

1.4.1 Atomicity

Traditionally, strict atomicity is preserved by restoring before images that are stored in the log. Strict atomicity can however also be preserved by means of compensating operations. Although many models have included compensation as a means for recovery, only recently a theory was established with a well-defined correctness criterion: the unified theory of concurrency and recovery [69, 77, 76]. Extensions to the basic theory demonstrated how concurrency and recovery can be improved by exploiting the semantics of compensating operations. However, the theory assumes that all operations are compensatable. This limits the applicability of the theory. In [70] the theory has been extended for operations that are non-compensatable. Unfortunately, this theory does not exploit the semantics of forward recovery operations.
Our model will exploit the semantics of both backward as well as forward recovery operations. The extra semantics of the forward recovery operations are used to improve concurrency and recovery.

1.4.2 Consistency

A serial execution of transactions is correct, because each transaction then observes and preserves database consistency. However, for consistency it is sufficient to serialize sub-transactions, as long as each sub-transaction preserves consistency. This approach is adopted by the SAGA model [38], where a transaction is decomposed into a sequence of steps that observe and preserve database consistency. The consistency requirement on steps constrains the applicability of the model. To overcome this limitation, we propose to decompose the transaction *semantics*, instead of decomposing the transaction itself. This way, decomposition is not constrained by the transaction's implementation, such that concurrency can be improved.

1.4.3 Isolation

In classical serializability theory, view serializability is considered the prime correctness criterion for schedules, because it specifies that each transaction should observe the same states as in a serial schedule and the final database state should reflect the state after a serial schedule. This implies, that in any view serializable schedule each transaction executes as if it executed in full isolation.

Although full isolation for transactions is sufficient for correct execution, it is not always necessary. This is one of the reasons why advanced transaction models have been developed.

In particular, models have been proposed to decompose transactions into steps that are allowed to interleave. In most cases the application designer has to specify what interleaving of steps is correct.

The problem with many of the interleaving approaches is that no guidelines are available for the application designer to establish what interleaving is correct. This problem was addressed by recent work [7, 13]. Here, transaction semantics are used to formally derive what interleaving is correct.

These approaches however consider interleaving specifications for the concurrency control, which is incompatible with existing serializability theory. This means, that existing results from serializability theory cannot be reused.

Therefore, in our approach we align the interleaving approaches with serializability theory. We investigate how weaker correctness criteria than conventional serializability can be developed, based on the application semantics. The main approach here is to consider weaker operation semantics than the actual semantics of operations that are still sufficient from the perspective of the application. These weaker semantics are used to define new correctness criteria that do not imply total isolation. We interpret conventional serializability theory in the context of these weaker semantics and investigate how conventional concurrency controls can be reused.

In a second step, we generalize the approach to the case where higher-level semantics are exploited, similar to the case of open nested transactions. These extra higher-level semantics are expected to improve concurrency, similar to the case of conventional open nested transactions [84].

1.4.4 Durability

Previous work on extending durability for long running applications has focused on a minimal loss of work in case of failure. Typically, either intermediate commits are submitted or savepoints are assumed for partial rollbacks [59]. An intermediate commit however, releases all locks, such that isolation is compromised. On the other hand, savepoints generally do not provide durability.

Therefore, we investigate the possibility to provide durability of intermediate results, in combination with partial rollbacks, while isolation and atomicity are guaranteed.

1.4.5 Validation

Throughout the thesis we will validate our extensions to the model by means of definitions, theorems and proofs. To demonstrate practical applicability, we apply our model to a representative application. For this, we take the TPC-W benchmark application, because it is representative for novel applications with high transaction processing demands [24].

1.5 Outline

We start in chapter 2 with a discussion on the basic concepts and definitions.

In chapter 3 we discuss our approach to relax isolation. In the first part of the chapter, we introduce a means to describe the relevant semantics of operations in the context of

their application. Based on the operation semantics, the semantics of schedules are defined. The semantics of schedules can then be used to express correctness in terms of semantic serializability. The second part of the chapter discusses how transaction semantics can be exploited to define weaker classes of semantic serializability.

In chapter 4 the approach is extended to the tree model[1]. It is discussed how application semantics can be exploited in this model and subclasses are introduced that can be scheduled efficiently.

Then, in chapter 5 we introduce semantic decomposition of transactions. The novel aspect here is that transactions are not decomposed by partitioning their operations into steps, but by decomposing the *semantics* of the transaction. This allows for more flexible decomposition. We define a weaker notion of serializability with respect to the decomposed semantics.

In chapter 6 the atomicity and durability properties of ACID are addressed. The notion of atomicity is relaxed by introducing forward recovery. In addition, the notion of durability is extended by introducing durability operations. The theory developed is an extension of the unifying theory of concurrency and recovery [77, 76].

Chapter 7 demonstrates that the developed theory has practical relevance. For this, we analyze the TPC-W benchmark application and we demonstrate that its transactional application semantics can be captured by our model. We perform a theoretic analysis by comparing the conflict definitions of the application according to the conventional definitions and our definitions in this thesis.

Finally, in chapter 8 we conclude and suggest future work.

[1] Also referred to as the object model [85].

Chapter 2

Preliminaries

In this chapter we introduce the basic concepts and definitions required for the remainder of this thesis. We start with the introduction of the definitions of operations and transactions. Then, in 2.2 we discuss the basic theory of concurrency control with some basic protocols. In 2.3 we discuss the basic theory of recovery. In 2.4 we describe the concurrency and recovery theory in the context of the tree model.

2.1 Definitions

A *database* is defined as a set of variables. We consider a single database *DB*. The state of a database is an assignment of values to *DB*. For database *DB* a set of *data operations PDB* is defined, where each operation $p \in PDB$ atomically transforms a state of the database into another state of the database.

Instances of operations can be grouped into *transactions* to perform complex state transitions. In the context of transactions, we distinguish the following additional variables: *transaction parameters* (TP) and *program variables* (PV). The set of transaction parameters includes those variables that interact with the user during the execution of the transaction. The set of program variables are invisible to the user and are used by the program to achieve the result of the transaction. The set of database variables, transaction parameters and program variables is called the set of *application variables* (AV).

In addition to data operations, we distinguish *transactional operations*. For the moment we only distinguish the *termination* operations *commit* (Cm) and *abort* (Ab). A transaction always terminates with either a commit or abort operation. A commit operation denotes the fact that all results of the transaction remain in the database, whereas an abort operation denotes that none of the results remain in the database [21]. A transaction that terminates with a commit operation is called *committed*, and a transaction that terminates with an abort operation is called *aborted*.

In the context of a transaction, we will refer to operation instances simply as operations to improve readability.

Definition 2.1.0.1 (Transaction) *A transaction is a tuple $T = (P, <)$, with:*

- *P a set of data operations and one termination operation* Tm*, with* $\mathrm{Tm} \in \{\mathrm{Ab}, \mathrm{Cm}\}$*,*
- *$<$ a partial order on P, called the intra-transaction order, with $\bigwedge_{p \in P \setminus \{\mathrm{Tm}\}} p < \mathrm{Tm}$.*

A schedule represents the execution of multiple transactions. We define a schedule as a set of transactions with a partial order on the transaction's operations.

Definition 2.1.0.2 (Schedule) *A schedule is a tuple $S = (T, P, <)$, with:*

- *T a set of transactions,*
- *P the operations of the set of transactions T, i.e. $P = \bigcup_{(P', <') \in T} P'$*
- *$<$ a partial order on P, called the dynamic order, with $\bigcup_{(P', <') \in T} <' \subseteq <$.*

By definition, the dynamic order of a schedule always contains the intra-transaction orders of all transactions. Extra orders may be added to a schedule for the sake of concurrency control and recovery. A schedule where the dynamic order is a total order of the operation is called a *totally ordered schedule*.

2.2 Concurrency control

Now we have introduced the definitions of operations, transactions and schedules, we are ready to define correctness of schedules in the presence of concurrency. For the moment, we ignore recovery related problems and assume that all transactions in a schedule have committed.

As the basic notion of correctness, we consider a schedule where no concurrency is present. Such schedule is called a *serial* schedule [21]. In a serial schedule, the operations of a transaction are not interleaved by operations of other transactions.

Definition 2.2.0.3 (Serial schedule) *Let $S = (T, P, <)$ be a schedule. Schedule S is a serial schedule if there is a total order $<_{total}$ on T with:*

$$\forall T_i, T_j \in T: \qquad T_i <_{total} T_j \Rightarrow \bigwedge_{p \in P_{T_i}, q \in P_{T_j}} p < q$$

Correctness of schedules can be expressed in terms of equivalence to a serial schedule. A schedule that is equivalent to a serial schedule is called *serializable*.

Equivalence of schedules is expressed in terms of equivalence of the *semantics of schedules*. The semantics of schedules can be derived from the semantics of its operations. We describe the *actual semantics* of an operation p by a function on application states f_p.

Definition 2.2.0.4 (Actual operation semantics) *The actual operation semantics of operation p are a function on application states $f_p : AS \to AS$ that describes the transformations performed by operation p on the application states.*

The actual semantics of a totally ordered schedule are defined as the composition of the functions f_p of its operations.

Definition 2.2.0.5 (Actual semantics of a totally ordered schedule) *Let $S = (T, P, <)$ be a totally ordered schedule. If for each operation p, function $f_p : AS \rightarrow AS$ represents the actual semantics of operation p and $<$ defines the order $p_1 < \cdots < p_n$ on P, the actual semantics of S are defined as:*

$$M(S) \quad = \quad f_{p_n} \circ f_{p_{n-1}} \circ \cdots \circ f_{p_1}$$

The actual semantics of an arbitrary schedule S with a partial order on its operations can also be defined. However, the semantics are only defined if the semantics of all totally ordered schedules compatible with S are the same. This intuition is captured by the following definition.

Definition 2.2.0.6 (Actual semantics of a schedule) *Let $S = (T, P, <)$ be a schedule. If for all totally ordered schedules $S'' = (T, P, <'')$ and $S''' = (T, P, <''')$ with $< \subseteq <''$ and $< \subseteq <'''$ we have $M(S'') = M(S''')$, the actual semantics of S are defined as:*

$$M(S) = M(S')$$

... where $S' = (T, P, <')$ an arbitrary totally ordered schedule with $< \subseteq <'$.

This definition is used to define the notion of final state equivalence on schedules. Final state equivalence only considers database variables. For notational convenience, we will write the set of application variables as a vector. We write $[V]_W$ to denote a vector that is identical to V with only the rows of the variables in W. In a similar way, we define the projection of a relation R on vectors as:

$$[R]_W \quad = \quad \{([V]_W, [V']_W) | (V, V') \in R\} \tag{2.1}$$

We define final state equivalence and final state serializability as follows.

Definition 2.2.0.7 (Final state equivalence) *Let $S = (T, P, <)$ and $S' = (T', P', <')$ be two schedules. Let f_p define the actual semantics of each operation p. Schedules S and S' are final state equivalent, if:*

- $T = T'$
- $P = P'$
- $[M(S)]_{DB} = [M(S')]_{DB}$

Definition 2.2.0.8 (Final state serializable (FSR)) [1] *A schedule is final state serializable, if it is final state equivalent to a serial schedule.*

[1] As a notational convention, serializability definitions are followed by abbreviation xSR. The abbreviation references the set of schedules that satisfies the serializability criterion. In this case, the set FSR represents the set of final state serializable schedules.

Final state serializability only requires equivalence to a serial schedule with respect to the (final) database state. This means that states returned to the user through the transaction parameters are not considered. Consequently, a user may observe non-serializable results. Therefore, the stronger criteria of view equivalence and view serializability are defined over database variables *and* transaction parameters.

Definition 2.2.0.9 (View equivalence) *Let $S = (T, P, <)$ and $S = (T', P', <')$ be two schedules. Let f_p define the actual semantics for each operation p. Schedules S and S' are view equivalent, if:*

- $T = T'$
- $P = P'$
- $[M(S)]_{DB \cup TP} = [M(S')]_{DB \cup TP}$

Definition 2.2.0.10 (View serializable (*VSR*)) *A schedule S is view serializable, if it is view equivalent to a serial schedule.*

Schedules that are view serializable return the same values to the user and the final database state is the same as for some serial schedule. Therefore, no difference can be observed between a serial schedule and a view serializable schedule. Consequently, view serializability is a satisfactory correctness criterion for many applications. Unfortunately, it cannot be implemented efficiently [64].

Therefore, the subclass of *conflict serializable schedules* has been suggested for concurrency control [63]. Conflict serializability is based on the conflict relation among operations [80]. The conflict relation is defined as follows.

Definition 2.2.0.11 (Conflict relation) *Let f_p define the actual semantics for each operation p. The conflict relation CON on $P \times P$ is defined:*

$$CON(p, q) \quad \Leftrightarrow \quad (f_p \circ f_q \neq f_q \circ f_p)$$

Two operations p and q are said to *conflict* if $CON(p, q)$ and to *commute* otherwise.

The definition implies that the order of conflicting operations is relevant for the semantics of a schedule. Therefore, in the remainder of this thesis, we will assume that in schedules the conflicting operations are ordered. Two schedules are considered conflict equivalent, if the order of their conflicting operations is the same.

Definition 2.2.0.12 (Conflict equivalence) *Let $S = (T, P, <)$ and $S' = (T, P, <')$ be identical schedules, except for their orders $<$ and $<'$. Schedules S and S' are conflict equivalent if:*

$$\forall p, q \in P : CON(p, q) \Rightarrow (p < q = p <' q)$$

A schedule that is conflict equivalent to a serial schedule is a conflict serializable schedule.

Definition 2.2.0.13 (Conflict serializable (*CSR*)) *Schedule S is conflict serializable, if it is conflict equivalent to a serial schedule.*

The class of conflict serializable schedules is a proper subclass of view serializable schedules. This important result is stated by theorem 2.2.0.16. The proof uses the following lemma.

Lemma 2.2.0.14 (Conflict equivalence implies semantic equivalence) *Let $S = (T, P, <)$ and $S' = (T, P, <')$ be identical schedules except for their orders $<$ and $<'$. If S and S' are conflict equivalent, then $M(S) = M(S')$.*

Proof 2.2.0.15 *(Analogous to [53, page 766])*
By definition 2.2.0.6 we can assume that $<$ and $<'$ are total orders without loss of generality, and thus we can assume some composition order for $M(S)$ and $M(S')$ of the actual semantics functions of the operations in P.

We prove the lemma, by demonstrating that the composition order of $M(S)$ can be rewritten to the composition order of $M(S')$ by interchanging the semantics functions in the composition, without changing $M(S)$. This way, we demonstrate that $M(S) = M(S')$.

To demonstrate that this can always be done if S and S' are conflict equivalent and the conflicting operations are ordered, consider the derived composition order at an arbitrary *moment during the interchange procedure:*

$$\cdots \circ f_{p_i} \circ f_{p_{i+1}} \circ f_{p_{i+2}} \circ \cdots \circ f_{p_{i+r-1}} \circ f_{p_{i+r}} \circ \cdots \tag{2.2}$$

We assume that for all $1 \leq j < r : f_{p_i}$ and $f_{p_{i+j}}$ are ordered the same as in $M(S')$.

Let $f_{p_{i+r}}$ be the first function that is ordered differently with respect to f_{p_i}. Hence, in $M(S')$ we have $f_{p_{i+r}}$ ordered before f_{p_i}. We can correct the composition order by moving $f_{p_{i+r}}$ before f_{p_i} by iteratively interchanging $f_{p_{i+j}}$ with $f_{p_{i+r}}$. By definition 2.2.0.11 the semantics are preserved if $\neg CON(p_{i+j}, p_{i+r})$.

Now, assume that we have some $f_{p_{i+k}}$ with $1 \leq k < r$ and CON(p_{i+k},p_{i+r}). Because S and S' are conflict equivalent, we have that $M(S') = \cdots \circ f_{p_{i+k}} \circ \cdots \circ f_{p_{i+r}} \circ \cdots \circ f_{p_i} \circ \cdots$. But then, $f_{p_{i+k}}$ was the first function that was ordered differently in (2.2), and not $f_{p_{i+r}}$. Thus, such $f_{p_{i+k}}$ cannot exist and $f_{p_{i+r}}$ can be moved before p_i without changing the semantics.

The interchange procedure can be repeated, until all functions are ordered the same as in $M(S')$. This way, we demonstrated that $M(S) = M(S')$.

Theorem 2.2.0.16 *$CSR \subset VSR$*

Proof 2.2.0.17

$S \in CSR$

$\Rightarrow$ *(definition 2.2.0.13)*

$\exists S' : S'$ is a serial schedule and S and S' are conflict equivalent

$\Rightarrow$ *(lemma 2.2.0.14)*

$\exists S' : S'$ is a serial schedule and $M(S) = M(S')$

$\Rightarrow$ *(monotonicity of $[R]_W$)*

$\exists S' : S'$ is a serial schedule and $[M(S)]_{DB \cup TP} = [M(S')]_{DB \cup TP}$

$\Rightarrow$ *(definition 2.2.0.10)*

$S \in VSR$

The proof of lemma 2.2.0.14 implies a procedural definition of conflict serializability, where unordered operations are ordered in an arbitrary way and non-conflicting operations are interchanged to demonstrate equivalence to a serial schedule. This motivates an alternative definition of conflict serializability where this procedure is made explicit. This alternative definition for conflict serializability is useful, because it allows for convenient generalizations of correctness definitions in the context of failures and the tree model in section 2.3 and 2.4.

Alternative definition 2.2.0.18 (Conflict serializable) *A schedule $S = (T, P, <)$ is conflict serializable if it can be transformed into a serial schedule by the following rules:*

- commute rule*: for an adjacent*[2] *ordered pair of operations $p < q$ reverse the order, if $\neg CON(p, q)$ and not $p <' q$ for some $(P', <') \in T$.*
- order rule*: an unordered pair p and q is ordered either $p < q$ or $q < p$.*

2.2.1 Protocols

We consider two types of concurrency control protocols in this thesis[3]. One is serialization graph testing, that accepts the complete class of *CSR* but is of quadratic complexity on the number of transactions. The other is based on two-phase locking. Two-phase locking is of linear complexity, but accepts only a subclass of *CSR*.

Serialization graph testing is defined on a serialization graph.

Definition 2.2.1.1 (Serialization graph) *Let $S = (T, P, <)$ be a schedule. Its serialization graph $SG(S)$ is a directed graph (N, V) with:*

- $N = T$
- $V = \{(T', T'') \mid T', T'' \in T \wedge T' \neq T'' \wedge p \in P' \wedge q \in P'' \wedge p < q \wedge CON(p, q)\}$

The serialization graph represents the conflict order among the transactions of a schedule. If this order is acyclic the schedule is conflict equivalent to a serial schedule [63].

Theorem 2.2.1.2 (Conflict serializability) *$S \in CSR \Leftrightarrow SG(S)$ is acyclic*

Proof 2.2.1.3 *[63]*

As mentioned above, serialization graph testing is of quadratic complexity. This complexity is generally considered too high for a concurrency control. Therefore, locking schedulers are often preferred.

[2] Two operations p and q are considered *adjacent* in the order $<$, if there is no operation k such that $p < k < q$ or $q < k < p$.

[3] Another well known type of concurrency control is time stamp ordering.

Locking schedulers synchronize access to data objects using locks, which can be set on and removed from data objects on behalf of transactions. Typically, before each operation p is executed, a lock must be acquired on all objects accessed by operation p and the locks are released only after completing operation p. With each type of operation, a lock type is associated. We refer the lock type for operation p simply as *lock p*. A lock p can only be acquired if no conflicting lock q is held by another transaction on the same data object. Locks p and q are considered *conflicting* or *incompatible*, if $CON(p, q)$, and *commuting* or *compatible* otherwise.

Based on this principle, a family of locking protocols can be identified that accepts only conflict serializable schedules. It has been demonstrated that for conflict serializability it is sufficient if each transaction locks its data objects according to a two-phase locking schema [35]. We define two-phase locking by means of the following rules.

Protocol 2.2.1.4 (Two-phase locking (2PL))

- Lock acquisition rule*: before operation p is executed, a lock p is acquired on all accessed objects.*
- Lock conflict rule*: a lock p is granted only if no lock q is held by a different transaction on the same data object with* $CON(q, p)$.
- Lock release rule*: once a lock p has been released, no lock q of the same transaction is acquired. All locks are released as soon as possible.*

We denote the class of schedules accepted by protocol family *Prot* as $Gen(Prot)$. Two-phase locking protocols accept only a proper subclass of *CSR*. This is stated by the following theorem.

Theorem 2.2.1.5 $Gen(2PL) \subset CSR$

Proof 2.2.1.6 *[63]*

2.3 Recovery

So far, we only considered correctness without the presence of failures. In this section we generalize to the case with failures. Three types of failures are generally distinguished: transaction failures, system failures and media failures. In this thesis, we only consider transaction and system failures, but no media failures. For a discussion on media failures we refer to [59]. We model the failure of a transaction by an abort operation and the failure of a system with an abort operation for all active transactions at the moment the system failed.

According to the atomicity requirement of ACID, for each transaction either all of its results should appear in the database or none. This can be achieved by removing all effects of transactions that have not yet committed.

The removal of effects of transactions can be performed by means of compensating operations [41, 37].

Definition 2.3.0.7 (Compensating operation) *Let p and q be two operations with actual semantics f_p and f_q respectively. Operation q is the compensating operation of operation p if:*

$$[f_p \circ f_q]_{DB} = [f_{Id}]_{DB}$$

... with f_{Id} the identity function on application states.

We denote the compensating operation of operation p with p^-. We will also refer to operation p as a *forward operation* and p^- as a *backward recovery* operation.

Upon abort, it is assumed that the system performs a recovery procedure. The abort operation can therefore be considered as an abbreviation of the recovery procedure.

Recovery is performed by submitting the compensating operations of the forward operations for the aborted transactions. This means, that upon failure, a schedule is *expanded* by the appropriate compensating operations. Such schedule is defined an expanded schedule and is defined below.

As a notational convention we assume Cm_i (Ab_i) to be the commit (abort) operation of transaction T_i and p_i an operation of T_i, unless otherwise noted.

Definition 2.3.0.8 (Expanded schedule) *Let $S = (T, P, <)$ be a schedule. The expansion of S is a schedule $X(S) = (T', P', <')$ with:*

- $T' = T$
- $P' =$
  - $\{p_i \mid p_i \in P_{T_i} \wedge p_i \notin \{\mathrm{Ab}_i, \mathrm{Cm}_i\}\} \cup$
  - $\{p_i^- \mid p_i \in P_{T_i} \wedge \mathrm{Cm}_i \notin P_{T_i} \wedge p_i \notin \{\mathrm{Ab}_i, \mathrm{Cm}_i\}\}$
- $<' = < \cup$
  - $\{(p_j, p_i^-) \mid p_i \in P_{T_i} \wedge p_j \in P_{T_j} \wedge (p_j, \mathrm{Ab}_i) \in <\}$
  - $\{(p_i^-, p_j) \mid p_i \in P_{T_i} \wedge p_j \in P_{T_j} \wedge (\mathrm{Ab}_i, p_j) \in <\}$
  - $\{(p_i^-, p_j-) \mid p_i \in P_{T_i} \wedge p_j \in P_{T_j} \wedge (\mathrm{Ab}_i, \mathrm{Ab}_j) \in <\}$

Correctness of recovery is defined on expanded schedules. We define correctness in a procedural way, similar to the procedural definition of *CSR*. In addition to the commute rule, we add the compensation rule that removes two operations p_i and p_i^- if p_i^- represents the compensating operation for p_i. The class of schedules that can be reduced to a serial schedule is defined reducible.

Definition 2.3.0.9 (Reducible (*RED*)) [4] *Let $S = (T, P, <)$ be a schedule and $X(S) = (T', P', <')$ its expanded schedule. Schedule S is reducible if $X(S)$ can be transformed into a serial schedule by the following rules:*

[4] For reducibility we take the same notational convention as for serializability. This means, that the set *RED* represents the set of schedules that are reducible.

- commute rule*: for an adjacent ordered pair of operations $p < q$ reverse the order, if $\neg CON(p, q)$ and not $p <_k q$ for some $(P_k, <_k) \in T$.*
- compensate rule*: remove any adjacent ordered pair $p <' p^-$, where p^- is the compensating operation of p.*
- order rule*: an unordered pair p and q is ordered either $p <' q$ or $q <' p$.*

It is assumed that failures can occur at any time during execution. This means, that each *prefix* of a schedule must be reducible. We verify reducibility of prefixes of schedules, by assuming a group abort of all active transactions.

2.3.1 Protocols

Reducibility of a schedule can be verified in two steps. First, the expansion of the schedule is constructed and reducibility is verified. Second, the reduced schedule is verified for serializability. Reducibility is verified by removing adjacent operation pairs p, p^- from the conflict graph of the expanded schedule. The conflict graph is defined as follows.

Definition 2.3.1.1 (Conflict graph (CG)) *The conflict graph $CG(S)$ of a schedule $S = (T, P, <)$ is a directed graph (N, V) with:*

- $N = P$
- $V = \{(p, q) \mid p < q \wedge CON(p, q)\}$

The removal of pairs p, p^- follows the definition of reducible schedules by applying the commute, compensate and order rules. The commute rule is implicitly contained in the conflict graph: if there is no path between two operations p and p^- they can be moved together by means of the commute rule.

Lemma 2.3.1.2 (Adjacency) *Two operations p and q can be moved together in S with the commute rule if and only if there is no path between p and q in $CG(S)$.*

Proof 2.3.1.3 *[76]*

By Lemma 2.3.1.2 the following procedure can be used to remove compensating operations from the conflict graph of the expanded schedule $X(S)$ [76].

Procedure 2.3.1.4 (Conflict graph reduction)

1. *Construct $G = CG(XS(S))$.*
2. *Find a pair of nodes p and p^- in G such that there is no path between them.*
3. *If such a pair does not exist then*
   (a) *If G contains some compensating operation, declare the schedule S non-reducible and exit, else*

(b) If G contains no compensating operation, then declare S reducible and exit.

4. *Remove p and p^- from G along with all edges incidental to these nodes.*
5. *Go to step 2*

Unfortunately, the verification procedure may cause too much overhead as a concurrency control. This is mainly due to the fact that paths must be verified for all pairs p and p^- in the graph.

To avoid the verification of paths for reducibility, the class of backward safe schedules (*BSF*) has been introduced [76]. Schedules in *BSF* have a commit and abort order among transactions depending on the conflict order of forward operations and compensating operations. The class is defined as follows.

Definition 2.3.1.5 (Backward safe (*BSF*)) *Schedule $S = (T, P, <)$ is backward safe if:*

$\forall T_i, T_j \in T, p_i \in P_{T_i}, p_j \in P_{T_j}$ *:*

- $T_i \neq T_j \wedge$
- $p_i < p_j \wedge$
- $(\mathrm{Ab}_i, p_j) \notin < \wedge$
- $CON(p_j, p_i^-) \Rightarrow$
  - $(\mathrm{Cm}_j \in P_{T_j} \Rightarrow \mathrm{Cm}_i \in P_{T_i} \wedge \mathrm{Cm}_i < \mathrm{Cm}_j) \wedge$
  - $(\mathrm{Ab}_i \in P_{T_i} \Rightarrow \mathrm{Ab}_j \in P_{T_j} \wedge (\mathrm{Ab}_i, \mathrm{Ab}_j) \notin <)$

The class imposes a commit order $\mathrm{Cm}_i < \mathrm{Cm}_j$, if an operation p_j of T_j is executed after p_i and p_j conflicts with p_i^-. This way reducibility is guaranteed, because if operation p_i^- cannot be moved to p_i due to its conflict with p_j, the operation p_j can be removed first. This latter intuition is captured by the last bullet of the definition.

The intersection class *BSF* $\cap$ *CSR* is a proper subclass of *RED*, which is stated by theorem 2.3.1.6.

Theorem 2.3.1.6 *$BSF \cap CSR \subset RED$*

Proof 2.3.1.7 *[76]*

Theorem 2.3.1.6 allows for the development of an efficient concurrency control. To guarantee that a schedule is a member of *BSF*, a termination graph is maintained according to the commit and abort dependencies in line with definition 2.3.1.5. To guarantee that a schedule is a member of *CSR*, we apply the two-phase locking protocol (see definition 2.2.1.4). The combined protocol was suggested in [76].

Protocol 2.3.1.8 (Backward safe hybrid protocol) *Consider a termination graph with transactions as nodes and directed edges between the nodes.*

- Lock acquisition rule: *before operation p_j is executed, a lock p_j is acquired on all accessed objects,*
- Lock conflict rule: *a lock p_j is granted only if no lock p_i is held with $CON(p_i, p_j)$.*
- Lock release rule: *after a lock p_j is released, no lock p_i of the same transaction is acquired. All locks are released as soon as possible.*
- Termination graph extension rule: *include transaction T_j of p_j in the termination graph, and add an edge $T_i \leftarrow T_j$ for non-aborted transaction T_i with some $p_i \in T_i$ and $CON(p_j, p_i^-)$.*
- Commit rule: *if no transaction T_i precedes T_j in the termination graph, grant the commit Cm_j and remove T_j from the termination graph; else abort T_j*
- Abort rule: *abort T_j together with all transactions T_i that follow T_j in the termination graph. Remove T_j and these transactions from the termination graph.*

2.4 The tree model

This section introduces the tree model for transactions. The tree model is a generalization of the conventional flat model. Whereas in the flat model operations are only ordered by the intra-transaction order, in the tree model operations are also hierarchically ordered. Non-leaf operations can be considered implemented by their children and thus are defined at a higher level of abstraction.

The motivation for the tree model is twofold. First, it provides a transaction theory for nested implementations of applications. Second, it can be used to improve concurrency by exploiting the semantics of the higher level operations and improve recovery by considering each nesting as a recovery scope.

A *transaction tree* is defined as follows [10, 9, 85].

Definition 2.4.0.9 (Transaction tree) *A transaction tree is a tuple $T = (P, <_a, <_i)$ with:*

- *T a set of data operations and transactional operations,*
- *$<_a$ a hierarchical order on P, called the abstraction order,*
- *$<_i$ a partial order on P, called the intra-transaction order,*
- $\forall k \in P : \mathrm{Ch}_{(P,<_a)}(k) \neq \emptyset \Rightarrow$
  - $\mathrm{Ch}_{(P,<_a)}(k)$ *is a set of data operations and one transactional operation* Tm_k, *with* $\mathrm{Tm}_k \in \{\mathrm{Ab}_k, \mathrm{Cm}_k\}$,
  - $\bigwedge_{p \in \mathrm{Ch}_{(P,<_a)}(k) \setminus \mathrm{Tm}_k} p <_i \mathrm{Tm}_k$
  - $\forall p, q, p', q' \in P : p <_i q \wedge p <_a p' \wedge q <_a q' \Rightarrow p' <_i q'$

In the definition, $\mathrm{Ch}_{(P,<_a)}(k)$ denotes the direct descendents of operation k in P with respect to $<_a$, i.e.:

$$\mathrm{Ch}_{(P,<_a)}(k) = \{p \mid p \in P \wedge k <_a p \wedge \neg\exists q \in P : k <_a q <_a p\} \tag{2.3}$$

As a notational convenience, we will abbreviate $\mathrm{Ch}_{(P,<_a)}(k)$ to $\mathrm{Ch}(k)$ if P and $<_a$ are clear from the context.

The intra-transaction order is defined over all operations of P, such that an order among two operations p and q means that all their descendents are ordered in the same way.[5]

A conventional transaction $T = (P, <)$ can be expressed as a transaction tree $T' = (P', <'_a, <'_i)$ with an additional root operation r as:

$$\begin{aligned} P' &= P \cup \{r\} \\ <'_a &= \{(r, p) \mid p \in P\} \\ <'_i &= < \end{aligned}$$

2.4.1 Concurrency control

Analogous to the conventional flat model, we introduce the definitions of a schedule in the tree model.

Definition 2.4.1.1 (Tree schedule) *A tree schedule is a tuple $S = (T, P, <_a, <_d)$, with:*

- *T a set of transaction trees*
- *P the operations of the set of transaction trees, with $P = \bigcup_{(P',<'_a,<'_i)\in T} P'$*
- *$<_a$ a hierarchical order on P, called the abstraction order, with $<_a = \bigcup_{(P',<'_a,<'_i)\in T} <'_a$*
- *$<_d$ a partial order on P, called the dynamic order, with:*
  - *$(\bigcup_{(P',<'_a,<'_i)\in T} <'_i) \subseteq <_d$*
  - *$\forall p_i, p_j, p'_i, p'_j \in P : p_i <_d p_j \wedge p_i <_a p'_i \wedge p_j <_a p'_j \Rightarrow p'_i <_d p'_j$*

The main point for introducing transaction trees is that the semantics of higher level operations can be exploited for the notion of equivalence to a serial history. The gain here is that two higher level operations may not conflict even though their children may conflict. This results in a larger class of schedules than conflict serializable schedules. This class of *tree conflict serializable schedules* (*TCSR*) is defined by a procedure of commute, compose and order rules. A schedule is defined *TCSR* if the procedure terminates with a *serial tree schedule*. This is a tree schedule of totally ordered root operations.

Definition 2.4.1.2 (Tree conflict serializable (*TCSR*)) *Let $S = (T, P, <_a, <_d)$ be a tree schedule. Tree schedule S is tree conflict serializable if it can be transformed into a serial tree schedule by the following rules:*

[5] This is referred to as tree-consistent node ordering [85].

- commute rule: *for an adjacent pair of ordered leaf operations* $p <_d q$ *reverse the order, if* $\neg CON(p, q)$ *and not* $p <'_i q$ *for some* $(P', <'_a, <'_i) \in T$.
- compose rule: *remove operations* $\mathrm{Ch}(k)$ *for some operation* k*, if* $\mathrm{Ch}(k)$ *are leaf operations and are non-interleaved by other leaf operations.*
- order rule: *an unordered pair of leaf operations* p *and* q *is ordered either* $p <_d q$ *or* $q <_d p$.

Protocols

The definition of *TCSR* is not constructive and therefore not suitable for developing a concurrency control. A more conservative class of *TCSR* that can be verified efficiently is called *multi-level serializability* [11, 83]. In this class it is assumed that transaction trees are layered, i.e. all leaf operations of each transaction tree have the same distance to the root operation.

Definition 2.4.1.3 (Layered schedule) *A layered schedule is a tree schedule* $S = (T, P, <_a, <_d)$ *where the leaf operations of all transaction trees have the same distance to the root operation with respect order* $<_a$.

The reason why a layered schedule can be verified efficiently, is that each level can be considered a conventional flat schedule that, to a large extent, can be scheduled in a modular way. Each such flat schedule in a layered schedule is defined a *level-to-level schedule*. A layered schedule with n levels is referred to as an *n-level schedule*.

Definition 2.4.1.4 (Level-to-Level Schedule) *Let* $S = (T, P, <_a, <_d)$ *be an n-level schedule with layers* $L_0, \ldots, L_n$ *(in top-down order). The level-to-level schedule* $S_k = (T_k, P_k, <_k)$ *from level* k *to level* $k + 1$ *with* $0 \leq k < n$ *is a schedule with:*

- T_k *the set of transactions that represent the operations in* P *with distance* k *to the root w.r.t.* $<_a$.
- P_k *the set of operations of* P *with distance* $k + 1$ *to the root w.r.t.* $<_a$,
- $<_k$ *the order* $<_d$ *on the operations* P_k.

It has been demonstrated that conflict serializability for each level-to-level schedule is not sufficient for *TCSR* [82]. An additional property of level-to-level schedules is required, where for each pair of conflicting operations, there must be at least one conflicting pair among their children. This is referred to as *conflict faithfulness*.

Definition 2.4.1.5 (Conflict Faithfulness) *A layered schedule* $S = (T, P, <_a, <_d)$ *is conflict faithful if for each conflicting pair* p *and* q *at the same level, there are two conflicting children* $p' \in \mathrm{Ch}(p)$ *and* $q' \in \mathrm{Ch}(q)$.

Conflict faithfulness and conflict serializability for each level in layered schedules is sufficient for *TCSR*. This is expressed by theorem 2.4.1.6.

Theorem 2.4.1.6 (Multi-level serializable (*MCSR*)) *An n-level schedule S is tree conflict serializable if it is conflict faithful and for all $k : 0 \leq k < n$: level-to-level schedule $S_k \in CSR$.*

Proof 2.4.1.7 *[85]*

Theorem 2.4.1.6 allows for a highly modular concurrency control that is based on two-phase locking for each level-to-level schedule.

2.4.2 Recovery

The recovery theory of section 2.3 can be extended to the tree model. The interesting aspect of the tree model in the context of recovery is that the hierarchical structure of transaction trees can be exploited for more refined recovery. Each non-leaf operation is considered a separate recovery scope, such that an abort or commit of the non-leaf operation only applies to the children of this non-leaf operation. Each non-leaf operation can then be considered a sub-transaction.

A schedule of transaction trees with aborted non-leaf operations can be expanded similar to the expansion of conventional schedules. The expansion of tree schedules is defined as follows:[6]

Definition 2.4.2.1 (Expanded tree schedule) *Let $S = (T, P, <_a, <_d)$ be a tree schedule. The expanded tree schedule of S is a tree schedule $X(S) = (T', P', <'_a, <'_d)$ with:*

- $T' = T$
- $P' =$
  - $\{p_i \mid p_k \in P \wedge p_i \in \mathrm{Ch}(p_k) \wedge p_i \notin \{\mathrm{Ab}_k, \mathrm{Cm}_k\}\} \cup \{\mathrm{Cm}_k\} \cup$
  - $\{p_i^- \mid p_k \in P \wedge p_i \in \mathrm{Ch}(p_k) \wedge \mathrm{Ab}_k \in \mathrm{Ch}(p_k) \wedge p_i \neq \mathrm{Ab}_k\}$
- $<'_a = <_a \cup \{(p_k, p_i^-) \mid p_k, p_i \in P \wedge p_k <_a p_i\}$
- $<'_d = <_d \cup$
  - $\{(p_j, p_i^-) \mid p_j, p_k \in P \wedge p_i \in \mathrm{Ch}(p_k) \wedge p_j <_d \mathrm{Ab}_k\}$
  - $\{(p_i^-, p_j) \mid p_j, p_k \in P \wedge p_i \in \mathrm{Ch}(p_k) \wedge \mathrm{Ab}_k <_d p_j\}$
  - $\{(p_i^-, p_j^-) \mid p_k, p_l \in P \wedge p_i \in \mathrm{Ch}(p_k) \wedge p_j \in \mathrm{Ch}(p_l) \wedge \mathrm{Ab}_k <_d \mathrm{Ab}_l\}$

The difference between the conventional definition of expanded schedule and expanded tree schedule is that the abstraction order is extended, such that compensating operations are added as children of aborted or active non-leaf operations.

Correctness for the expanded tree schedule is defined by combining tree conflict serializability and reducibility.

Definition 2.4.2.2 (Tree reducible (*TRED*)) *Let $S = (T, P, <_a, <_d)$ be a tree schedule and $X(S) = (T', P', <'_a, <'_i)$ its expanded tree schedule. Tree schedule S is tree reducible if $X(S)$ can be transformed into a serial tree schedule by applying the following rules:*

[6] We write Tm_k to denote the termination operation of operation k, with $\mathrm{Tm}_k \in \{\mathrm{Cm}_k, \mathrm{Ab}_k\}$

- commute rule: *for an adjacent pair of ordered leaf operations* $p <'_d q$ *reverse the order, if* $\neg CON(p, q)$ *and not* $p <''_i q$ *for some* $(P'', <''_a, <''_i) \in T'$.
- compose rule: *remove operations* $\mathrm{Ch}(k)$ *for some operation* k*, if* $\mathrm{Ch}(k)$ *are leaf operations and are non-interleaved by other leaf operations.*
- compensate rule: *remove any adjacent leaf pair* $p <_d p^-$*, where* p^- *is the compensating operation of* p.
- order rule: *an unordered leaf pair* p *and* q *is ordered either* $p <'_d q$ *or* $q <'_d p$.

Chapter 3

Semantics-based concurrency control

3.1 Introduction

In conventional serializability theory it is assumed that each transaction must execute in full isolation. This has important consequences for the allowable degree of concurrency, because the execution must result in the same observed states as a serial execution of transactions, i.e. the execution must be *view serializable*.

However, for many applications full isolation is not required: these applications can tolerate some degree of interference during execution, while they still produce their intended effect on the database. The fact that some interference can be tolerated can have significant impact on concurrency, because certain non-view serializable interleaving of operations can be accepted. The following example gives a schedule that is not view serializable, but can still be considered correct.

Example 3.1.0.3 (Correct non-view serializable schedule) *Consider a transaction T_1 that takes the address of a customer and delivers a parcel to the customer's address. In addition, consider a concurrent transaction T_2 that updates the address. Transaction T_1 is implemented by two operations* intake *and* deliver*. Operation* intake *takes address n from the user through transaction parameter n and writes the address to the database variable a. Operation* deliver *reads database variable a, and writes the delivery address to variable d. Transaction T_2 is implemented by a single operation update that writes an address to variable a.*

Now, consider the following schedule:

T_1 : *intake* *deliver*

T_2 : *update*

The schedule represents an execution in which transaction T_1 does not execute in full isolation: operation deliver *reads the address produced by operation update instead of the address produced by operation intake. This is not acceptable, if T_1 requires that the delivery address d is the same as the address supplied by the user. However, if the delivery address*

need not be the same as the address taken from the user, the schedule is acceptable even though operation update *writes a different address in variable a. This would be the case if delivery only requires that the address a is valid when it starts.*

The example illustrates that for certain applications isolated execution is not required. The fact that isolated execution is not required may improve concurrency. This fact is particularly important for long transactions, because isolated execution of long transactions may have considerable impact on concurrency.

The example also illustrates, that the required degree of isolation of transactions is dependent on the semantics of the application. The semantics for T_1, where the delivery address must be the same as the user supplied address, did not tolerate interference, whereas the relaxed semantics of T_1, where delivery can be done to any valid address, allowed for interference by T_2.

In this chapter we investigate how application semantics can be exploited to improve concurrency by reducing isolation. We aim at a general formal model that captures semantics of applications and we pursue practical concurrency controls.

We are not the first to investigate this. An overview of the most important approaches is provided in the next section.

3.1.1 Related work

Considerable work has been done to improve concurrency while exploiting the semantics of applications. Essentially, the work can be split into two directions: one direction that maintains view serializability as correctness criterion, while the other direction abandons this correctness criterion and instead relies on interleaving specifications of transactions.

The first direction generalizes the classical work on concurrency control. The classical approach adopts conflict serializability as correctness criterion, because it is a subclass of view serializable schedules that is suitable for efficient concurrency controls. The class of conflict serializable schedules is extended in two steps.

The first step considers the semantics of the operations. Instead of considering only read and write semantics for operations, the actual function of the operation is considered in the context of the application. With these extra semantics, certain previously conflicting operations can be identified as non-conflicting. This results in a weaker conflict relation, such that the class of conflict serializable schedules is extended [80, 55].

The second step extends the first to the case where the combined semantics of a set of operations are considered. This was initially motivated by the architecture of transactional systems. In [82] this approach was taken for layered systems. Here, transactions are considered as operations that invoke operations at lower levels in the system. In [5] this work was further generalized to non-layered, composite systems. It has been demonstrated that the higher-level semantics of sets of operations can improve concurrency significantly.

These works demonstrated that by exploiting semantics of operations and sets of operations transactions can be allowed to execute concurrently. However, the approaches all preserve view serializability and thus only schedules are accepted that preserve full isolation. As a result, the example from the introduction is not by supported by any of these

approaches.

The second direction does not consider view serializability as basis for correctness and allows for non-isolated execution. Instead, transactions are decomposed into atomic steps. Steps generally consist of a subset of operations of the transaction that execute in a serializable way. Correctness is defined by the application designer by means of an allowable interleaving of the steps of concurrent transactions.

Initial work on interleaving specifications can be found in [37]. Here, the database is partitioned into database variables that require serializability and those that do not require serializability. Based on this distinction, transactions are grouped, such that the transactions in the same compatibility set can be arbitrarily interleaved without violating serializability, whereas transactions in different groups cannot be interleaved at all. A special case of this approach is the SAGA model [38]. In this model all transactions can interleave in an arbitrary way. To preserve correctness, it is required that each step observes and preserves database consistency. This requirement severely limits the decomposition possibilities of transactions.

In [56] a more refined model of specifying allowable interleaving of transactions is proposed. For each transaction, the operations are grouped according to a hierarchical structure, such that at each level of the hierarchy a set of breakpoints between the groups is implied. Transactions are also hierarchically grouped. The lowest level at which two transactions T_i and T_j are grouped, determines at what layer the operations of both transactions can be interleaved. Despite the fact that the specification model is more flexible than [37], the hierarchical approach has limited flexibility.

The model proposed in [36] generalizes the proposal of [56]. Here, each transaction is decomposed into steps and for each step it is specified what transactions are allowed to interleave. Although this is more flexible, only interleaving with respect to whole transactions can be specified.

The work [1] is the first to combine interleaving specifications with conventional serializability theory. For this, the operations of each transaction are grouped into atomic units. The atomic units are defined relative to other transactions and define how the atomic units are allowed to interleave. A schedule is considered *relatively serializable*, if the atomic units are serializable with respect to each other. It is demonstrated that the approach is a generalization of [36]. In [75] the approach of [1] is further refined, such that atomic units can be defined relative to operations instead of transactions.

Although the sketched approaches provide a refined model for specifying interleaving specifications, they leave the responsibility for correctness of the interleaving specification with the application designer. In addition, no guidelines are offered to the application designer what interleaving should be correct. The design process is therefore complex, error prone and thus rather impractical.

To overcome this design problem, some approaches have suggested the specification of isolation predicates between steps. One such approach is the ConTract model [66, 79, 67]. A ConTract can be considered a partial order of steps. With each step an entry and exit invariant is associated, similar to a pre- and post-condition of the step. Entry invariants may reference exit invariants of previously executed steps, such that a degree of isolation can be specified between steps. If a step S_i invalidates an entry or exit invariant of a step S_j it is considered ordered before or after S_j. As a correctness criterion, ConTracts should

execute in a serializable way with respect to this order.

A similar approach is taken in [2]. Here, a model is proposed where transactions are decomposed into steps and where the application designer explicitly states a pre-condition before every step that states the 'required consistency' of the database. Schedules are only correct if the consistency assertions are satisfied before each step. Unfortunately, the consistency assertions are not sufficient to guarantee that transactions observe consistent database states. For this a notion of orderability is introduced, similar to serializability.

The advantage of these approaches, is that the specification of the allowed interleaving is more intuitive, because it can be expressed in terms of the semantics of the transactions themselves, and not relative to other transactions. However, this specification appears insufficient for correctness, because it is not sufficient for observing consistent database states. Therefore, again, a notion of serializability is introduced. The relation between the semantics of the application and the notion of serializability remains unclear.

The first work to establish a formal link between the semantics of transactions and the notion of serializability is [17, 14, 13]. The approach takes as input a formal specification of the semantics of transactions in terms of a pre- and post-condition. A schedule is defined correct, if it satisfies the compositional result of a sequence of transactions, and database consistency is restored. Importantly, the compositional result is defined in terms of the semantics of transactions and not in terms of database states. As a result, semantic correctness is weaker than view serializability. It only requires that the specifications of the transactions are satisfied, but does not require that transactions execute in full isolation. For the concurrency control, the formal specification of the semantics of a transaction are used to establish the minimal constraints between the steps of a transaction that are required for each transaction to meet its specification. As such, the minimal isolation requirements between the steps can be formally derived. At run-time a locking-based 'assertional' concurrency control controls the interleaving of steps, such that semantic correctness is preserved.

Possibly one of the most important contributions of the work is the definition of semantic correctness as a correctness criterion that is weaker than view serializability, proposed in [13]. Semantic correctness allows for non-isolated execution of transactions, such as our example from the introduction, while preserving some notion of serializability. However, much of the potential gain is lost in the proposed concurrency control. For example, the assertional concurrency control does not allow concurrent execution of transactions that invalidate each other's results. Second, the approach has a practical limitation on the decomposition of steps, because the concurrency control does not exploit the combined semantics of sets of steps. This way, a reasonable granularity of steps must be maintained, in order to exploit some higher-level semantics such as in [82].

3.1.2 Approach

The discussion of related work shows that the limitations of view serializability can be overcome by introducing explicit interleaving specifications. This however burdens the application designer with heavy responsibility for preserving correctness in the presence of concurrency. Even more important, it provides no guidelines for what interleaving of steps is correct. As an initial attempt, explicit isolation predicates were introduced, that guarantee that at least the specifications of transactions are met. However, these predicates alone prove insufficient for correctness definitions for schedules. The main problem here is

that no notion of serializability is used in these approaches. Not surprisingly, the notion of serializability was reintroduced in recent approaches. Eventually, in [13] a formal link was established between isolation predicates between steps and the notion of serializability.

The work of [13] comes forth from the tradition of interleaving specifications, and not surprisingly, a new concurrency control was proposed that is based on allowed interleaving of steps. The problem with the approach and the concurrency control in particular is that it ignores the existing serializability theory, with all its practically relevant results.

Therefore, in this chapter we investigate how conventional serializability theory can be applied in the context of the approaches of interleaving specifications, and in particular the work of [13]. Such relation is relevant, because it enables the reuse of the results of serializability theory. Even more important, it allows for the application of the results to existing commercial systems that are mostly based on serializability theory with minimal effort.

Our serializability theory is based on an *application specific interpretation* of semantics of operations, schedules and transactions. Instead of considering the actual semantics of operations, we only consider the *relevant* semantics of operations *in the context of their application*. The relevant semantics of operations are generally weaker than the actual semantics of an operation. For example, the function of operation intake from the example in the introduction writes the *value* of the input parameter n to database variable a. However, in the context of the application, the relevant semantics of operation *intake* are much weaker: it is only required that *some* valid address is written to database variable a. This makes, that the address after operation intake may be modified by a concurrent transaction T_2 without invalidating the *relevant* result of operation intake.
We introduce a definition of semantics that can capture this subtlety. Based on this definition, we introduce a definition for the semantics of operations, and extend this definition to schedules, defined as the semantic composition of its operations. Such definition is then used to introduce novel correctness criteria that are semantic interpretations of the conventional notions of final state serializability, view serializability and conflict serializability. We refer to these new correctness criteria as *semantic* final state serializability, *semantic* view serializability and *semantic* conflict serializability.
The definition of semantic conflict serializability is of high practical relevance, because many concurrency controls are based on conflict serializability. Semantic conflict serializability is defined based on the semantic interpretation of the conventional conflict relation among operations: the *semantic* conflict relation. For the concurrency control, we propose conventional techniques such as serialization graph testing and two-phase locking. The improvement in concurrency therefore comes directly from the weaker definition of the conflict relation.

In addition, we investigate the possibility to exploit higher-level semantics. Initially, we define semantics of transactions in a similar way as semantics for operations. Based on such definition, the definition of the semantics of a serial schedule can be expressed in a similar way as the semantics of a schedule. The semantics of a serial schedule of transactions can then be used to introduce even weaker notions of correctness than the introduced semantic serializability notions, because the semantics of transactions can be exploited. This approach of exploiting semantics of transactions can be further generalized to the tree model, such that semantics of sets of operations can also be exploited. This approach is discussed in chapter 4.

3.1.3 Outline

We start our discussion in section 3.2 by motivating the need for a more general definition of semantics than semantics based on functions (see chapter 2). Then, we introduce such a definition of semantics for application state transitions. The definition is used to define the semantics of operations. The semantics of schedules are defined as the composition of the semantics of its operations.

In section 3.3 the new definition of semantics is used to introduce novel correctness criteria for schedules. In particular, we introduce the criteria *semantic final state serializability*, *semantic view serializability* and *semantic conflict serializability*. The latter correctness criterion is similar to conventional conflict serializability, except for the semantic interpretation of the conflict relation. We demonstrate how weaker application semantics result in higher degrees of concurrency.

In section 3.4 we demonstrate how concurrency can be further improved by exploiting the semantics of transactions. We introduce a definition for correct transaction semantics and introduce the definition of *sufficient operation semantics* in the context of transaction semantics. This definition is useful, because it can be used to define weaker operation semantics, which results in higher degrees of concurrency. We introduce novel correctness criteria based on the semantics of transactions.

We conclude in section 3.5.

3.2 Semantics of operations and schedules

In chapter 2 we introduced a semantics based on functions for operations in the context of a schedule. These semantics are defined for each operation by a function on application states. We implicitly assumed that these semantics of the operations are *sufficient* for each transaction to achieve its intended result. However, the function of the operation may however not be *necessary* for the application to achieve its intended result.

As an illustration, reconsider example 3.1.0.3 from the introduction. The operation *intake* takes some address from the user and stores it in database variable a. The resulting state of address a is therefore the same as the state of the input parameter n. However, according to the weaker semantics of the application, it appears that the *relevant* semantics of operation intake are weaker. It is not required that the resulting state of the address is the same as the address supplied by the user. Rather, it is sufficient, if the address a represents *some* valid address. This explains why a concurrent transaction is allowed to change the state of the address during execution of transaction T_1. We therefore conclude, that the relevant semantics of operation intake require that the resulting address a is valid, rather than that it is required that the state of address a equals the state of input parameter n.

This means, that the semantics of operation intake are sufficient for transaction T_1 to complete successfully, but are not necessary. Instead, weaker semantics of the operation are sufficient for T_1 that allow a result consisting of a *set of states* on variable a, i.e. the set of states where variable a represents a valid address.

The reasoning above motivates us to generalize the definition of operation semantics from *functions* to binary *relations* on application states. We start with a generic definition of semantics of application state transitions.

Definition 3.2.0.1 (Application state transition semantics) *The semantics of an application state transition are defined as a binary relation on the set of application states $R : AS \times AS$, where each application state is represented by a vector corresponding to the application variables. We reference the components of the vector pair as:*

$$((x1_{pre}, x2_{pre}, \ldots, xn_{pre}), (x1_{post}, x2_{post}, \ldots, xn_{post}))$$

... with $x1, x2, \ldots, xn$ the application variables AV.

In the sequel of this thesis, we will refer to an application state transition, simply as *transition* and we may refer to application state transition semantics simply as *semantics*.

We identify the binary relation R on pairs of application states with its characterizing predicate function F, such that:

$$((x1_{pre}, x2_{pre}, \ldots, xn_{pre}), (x1_{post}, x2_{post}, \ldots, xn_{post})) \in R$$
$$\Leftrightarrow$$
$$F(x1_{pre}, x2_{pre}, \ldots, xn_{pre}, x1_{post}, x2_{post}, \ldots, xn_{post})$$

This way, we can write semantics either by means of the binary relation or the characterizing predicate function. We illustrate the definition by means of the following example.

Example 3.2.0.2 (Semantics) *Let AV be the set of application variables with $AV = \{x, y, z\}$. The semantics of the transition where variable x is incremented by one can be described by predicate function:*

$$x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} \wedge z_{post} = z_{pre} \tag{3.1}$$

As the set of appliation variables is generally large, we may abbreviate those conjuncts for variables that remain invariant by $\mathrm{Inv}(V)$. Predicate Inv is defined as:

$$\mathrm{Inv}(V) =_{def} \bigwedge_{x \in AV \setminus V} x_{post} = x_{pre}$$

Thus, for the transition of the example we can write:

$$x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\}).$$

We illustrate this definition by example 3.1.0.3 from the introduction.

Example 3.2.0.3 (Semantics predicate) *The semantics of the operations in the introduction example can be specified as:*

$$R_{intake} = (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\}))$$

The premisse expresses the fact that at the beginning of operation intake the variable n represents a valid address. The conclusion expresses the fact that variable a is written with the value of n.

$$R_{deliver} = (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge \mathrm{Inv}(\{d\}))$$

The premisse states that variable a must represent a valid address. The conclusion states that the delivery address is written with the same as the address a.

$$R_{update} = (valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))$$

Operation update *writes some valid address to variable a.*

We have motivated our definition for semantics by the claim that it allows us to introduce weaker semantics for operations than the actual semantics of operations. Intuitively, semantics R' for an operation are weaker than semantics R, if R' allows for at least the same state transitions as R. We define this formally as follows.

Definition 3.2.0.4 (Weaker/stronger semantics) *Let R and R' be two semantics. Semantics R' are weaker than semantics R (semantics R are stronger than R') if $R \subset R'$.*

Example 3.2.0.5 (Weaker/stronger semantics) *Consider the semantics R_{intake} for operation intake as given in example 3.2.0.2:*

$$R_{intake} = (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\}))$$

The semantics R_{intake} specify that after operation intake, variable a has the same value as the initial value of variable n.

If for a certain application it is only required that after operation intake variable a represents some valid address, but not necessarily $a = n$, we may assign the following semantics to operation intake as well:

$$R'_{intake} = (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))$$

We have that $R_{intake} \subset R'_{intake}$, i.e. R'_{intake} is a weaker relation on state transition pairs than R_{intake}.

Although theoretically any semantics can be assigned to operations, in practice operations will perform state transitions according to their *actual semantics*.[1]. Any semantics assigned to operations must therefore be weaker than the actual semantics. Semantics with this property are called *justified*.[2]

Definition 3.2.0.6 (Justified semantics) *Let R be the actual semantics of operation p and let R' be some arbitrary semantics. R' are justified semantics for p if: $R \subseteq R'$.*

In the examples 3.2.0.5 and 3.2.0.3, the semantics R_{intake} represent the actual semantics. The semantics R'_{intake} are justified, because $R_{intake} \subseteq R'_{intake}$. In the remainder of this thesis, we assume justified semantics for operations.

Now we have defined the semantics of operations, we can define the semantics of schedules. The semantics of schedules are generally not specified by the user, because a schedule is generally not known at design-time. Therefore, we introduce a definition for the composition of semantics. We start with sequential composition of semantics, which is defined as the relational composition.

Definition 3.2.0.7 (Composition of semantics) *The composition of semantics R and R', denoted $R \oplus R'$ is defined:*

$$\begin{aligned} R \oplus R' =_{def} \\ &\{((x1_{pre}, \ldots, xm_{pre}), (x1_{post}, \ldots, xm_{post})) \mid \exists (x1_{mid}, \ldots, xm_{mid}) : \\ &((x1_{pre}, \ldots, xm_{pre}), (x1_{mid}, \ldots, xm_{mid})) \in R \wedge \\ &((x1_{mid}, \ldots, xm_{mid}), (x1_{post}, \ldots, xm_{post})) \in R'\} \end{aligned}$$

[1] See definition 2.2.0.4

[2] For the definition, we redefine the actual semantics function as a relation.

It follows from the identification of binary relations with functions (see page 45) that we can write $F \oplus F'$ for $R \oplus R'$, with:

$$
\begin{array}{l}
F \oplus F' = \\
\quad \exists(x1_{mid}, \ldots, xm_{mid}) : \\
\quad F[(x1_{post}, \ldots, xm_{post}) \leftarrow (x1_{mid}, \ldots, xm_{mid})] \wedge \\
\quad F'[(x1_{pre}, \ldots, xm_{pre}) \leftarrow (x1_{mid}, \ldots, xm_{mid})]
\end{array}
$$

... where $P[(x1, \ldots, xm) \leftarrow (y1, \ldots, ym)]$ denotes that each variable xi in predicate P is replaced by variable yi.

We illustrate the definition by means of the delivery example.

Example 3.2.0.8 (Composition of semantics) *Recall the semantics of the operations intake and deliver from example 3.2.0.3:*

$$
\begin{array}{rcl}
R_{intake} & = & (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\})) \\
R_{deliver} & = & (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge \mathrm{Inv}(\{d\}))
\end{array}
$$

The semantics of the sequential composition are defined as:

$$
\begin{array}{l}
R_{intake} \oplus R_{deliver} \\
= \exists(n, a, d, \ldots) : \\
\quad (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\}))[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)] \wedge \\
\quad (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge a_{pre} = a_{post} \wedge \mathrm{Inv}(\{d\}))[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)] \\
= \exists(n, a, d, \ldots) : \\
\quad (valid(n_{pre}) \Rightarrow a = n_{pre} \wedge \mathrm{Inv}(\{a\})[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)]) \wedge \\
\quad (valid(a) \Rightarrow d_{post} = a \wedge a = a_{post} \wedge \mathrm{Inv}(\{d\})[(n_{pre}, a_{pre}, d_{pre}, \ldots) < -(n, a, d, \ldots)]) \\
= (valid(n_{pre}) \Rightarrow n_{pre} = n_{post} = d_{post} = a_{post} \wedge \mathrm{Inv}(\{a, d\}))
\end{array}
$$

The definition of sequential composition can be generalized to sets of totally ordered semantics.

Definition 3.2.0.9 (Totally ordered semantics) *Let RR be a set of semantics and $<_R$ a total order on RR, with $R_1 <_R \ldots <_R R_n$. The total order of semantics are defined as:*

$$(RR, <_R) =_{def} R_1 \oplus \ldots \oplus R_n$$

The definition can be applied to totally ordered schedules in the context of some semantics assignment to its operations. This is illustrated by the following example.

Example 3.2.0.10 (Totally ordered semantics) *Reconsider example 3.1.0.3 from the introduction, with schedule:*

T_1	*intake*		*deliver*
T_2		*update*	

. . . and operation semantics:

$$\begin{aligned} R_{intake} = &\quad (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\})) \\ R_{deliver} = &\quad (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge \mathrm{Inv}(\{d\})) \\ R_{update} = &\quad (valid(a_{post}) \wedge \mathrm{Inv}(\{a\})) \end{aligned}$$

The semantics of the totally ordered schedule are derived as follows:

$$\begin{aligned} & R_{intake} \oplus R_{update} \oplus R_{deliver} \\ = \; & (\exists(n, a, d, \ldots) : \\ & (valid(n_{pre}) \Rightarrow a_{post} = n_{pre} \wedge \mathrm{Inv}(\{a\}))[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)] \wedge \\ & (valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)]) \\ & \oplus R_{deliver} \\ = \; & (\exists(n, a, d, \ldots) : \\ & (valid(n_{pre}) \Rightarrow a = n_{pre} \wedge \mathrm{Inv}(\{a\})[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)] \wedge \\ & (valid(a_{post}) \wedge \mathrm{Inv}(\{a\})[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)])) \\ & \oplus R_{deliver} \\ = \; & ((valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))) \oplus R_{deliver} \\ = \; & \exists(n, a, d, \ldots) : \\ & (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)] \wedge \\ & (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge \mathrm{Inv}(\{d\}))[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)] \\ = \; & \exists(n, a, d, \ldots) : \\ & (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \mathrm{Inv}(\{a\}))[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)] \wedge \\ & (valid(a_{pre}) \Rightarrow d_{post} = a_{pre} \wedge \mathrm{Inv}(\{d\}))[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)] \\ = \; & \exists(n, a, d, \ldots) : \\ & (valid(n_{pre}) \Rightarrow valid(a) \wedge \mathrm{Inv}(\{a\})[(n_{post}, a_{post}, d_{post}, \ldots) \leftarrow (n, a, d, \ldots)]) \wedge \\ & (valid(a) \Rightarrow d_{post} = a \wedge \mathrm{Inv}(\{d\})[(n_{pre}, a_{pre}, d_{pre}, \ldots) \leftarrow (n, a, d, \ldots)]) \\ = \; & (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge d_{post} = a_{post} \wedge \mathrm{Inv}(\{a, d\})) \end{aligned}$$

In general however, the operations in a schedule are not totally ordered, but partially ordered. The semantics of a partial order of operations can be defined as the disjunction of the semantics of all total orders of the operations that satisfy the partial order.

Definition 3.2.0.11 (Partially ordered semantics) *Let RR be a set of semantics and $<_r$ a partial order on RR. Let $<_R$ be the set of total orders on RR that satisfies $\forall <_{R_i} \in <_R$: $(<_r \subseteq <_{R_i})$. The partially ordered semantics are defined as:*

$$(RR, <_r) =_{def} \bigcup_{<_{R_i} \in <_R} (RR, <_{R_i}).$$

This is an important definition in the context of schedules, because it allows us to reason about the partially ordered semantics of the operations in a schedule. In particular, if we have a schedule $S = (T, P, <)$ and the semantics for each operation $p \in P$ are defined by R_p. Then, the *composed semantics of the schedule* are defined by:

$$(\bigcup_{p \in P} R_p, <)$$

As a result, we can now reason about the semantics of schedules and relate these semantics to appropriate correctness criteria. This is discussed in the following section.

3.3 Semantic serializability

In this section we define correctness notions for semantic serializability. We follow a similar approach as conventional serializability theory by considering schedules serializable, if they are equivalent to a serial schedule.

As the basis for the equivalence relation on schedules, we consider the composed semantics of a schedule. Initially, we adapt the conventional definitions of final state serializability and view serializability to *semantic* final state serializability and *semantic* view serializability. Then, in section 3.3.2 we introduce the definition for semantic conflict serializability.

3.3.1 Semantic final state and semantic view serializability

The generalized definition of semantics for operations can be directly applied to the conventional definitions of final state serializability and view serializability. In the definitions we simply replace the semantics function $M(S)$ with the composed semantics of the schedule.

First, we introduce *semantic final state equivalence*. Similar to final state equivalence, semantic final state equivalence is only defined with respect to the database variables.

Definition 3.3.1.1 (Semantic final state equivalence) *Let $S = (T, P, <)$ and $S' = (T', P', <')$ be two schedules. Schedules S and S' are semantic final state equivalent in the context of justified semantics R_p for each operation $p \in P$, if:*

- $T = T'$
- $P = P'$
- $[(\bigcup_{p \in P} R_p, <)]_{DB} = [(\bigcup_{p \in P} R_p, <')]_{DB}$

Based on the definition of semantic final state equivalence, we can define semantic final state serializability as follows.

Definition 3.3.1.2 (Semantic final state serializable (*SFSR*)) *A schedule is semantic final state serializable if it is semantic final state equivalent to a serial schedule.*

The correctness criterion *SFSR* shares an important disadvantage with its conventional criterion *FSR* in that it defines serializability only with respect to database variables, but not with respect to results returned to the user. This is generally unacceptable.

Therefore, we introduce the class of *semantic view serializable* schedules that requires that users only observe results from serializable executions. We define semantic view serializability by means of semantic view equivalence of schedules. Semantic view equivalence of schedules is a stronger equivalence relation than semantic final result equivalence, because we interpret the semantics not only with respect to database variables but also with respect to the transaction parameters (TP) that interact with the users.

Semantic view equivalence and semantic view serializability are defined as follows.

Definition 3.3.1.3 (Semantic view equivalence) *Let $S = (T, P, <)$ and $S' = (T', P', <'$) be two schedules. Schedules S and S' are semantic view equivalent in the context of justified semantics R_p for each operation $p \in P$, if:*

- $T = T'$
- $P = P'$
- $[(\bigcup_{p \in P} R_p, <)]_{DB \cup TP} = [(\bigcup_{p \in P} R_p, <')]_{DB \cup TP}$

Definition 3.3.1.4 (Semantic view serializable ($SVSR$)) *A schedule is semantic view serializable if it is semantic view equivalent to a serial schedule.*

It may be clear from the definitions that the class of semantic view serializable schedules is a subclass of semantic final result serializable schedules, because the results on the transaction parameters are included in $SVSR$.

We illustrate the definition of semantic final result serializability and semantic view serializability by the following example.

Example 3.3.1.5 (SFSR and not SVSR) *Consider transactions T_1 and T_2. Transaction T_1 consists of operations* incx *and* incy*, and transaction T_3 consist of one operation* audit*. The semantics of the operations are defined by:*

$$\begin{aligned} R_{incx} &= (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\})) \\ R_{incy} &= (y_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{y\})) \\ R_{audit} &= (p_{post} = x_{pre} \wedge q_{post} = y_{pre} \wedge \mathrm{Inv}(\{p, q\})) \end{aligned}$$

... with $x, y \in DB$ and $p, q \in TP$. Now, consider schedule S:

T_1 : *incx* *incy*

T_2 : *audit*

We demonstrate that $S \in SFSR$ and $S \notin SVSR$. For this, we derive the semantics of schedule S. Then, we derive the semantics of the serial schedule S' where operation audit *is ordered after* incy*. Based on the composed semantics of schedule S and S', we conclude that S is semantic final state serializable and not semantic view serializable.*
The semantics of schedule S are derived as follows :

$$\begin{aligned} & Rincx \oplus Raudit \oplus Rincy \\ \Leftrightarrow\ & (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\})) \oplus (p_{post} = x_{pre} \wedge q_{post} = y_{pre} \wedge \mathrm{Inv}(\{p, q\})) \oplus Rincy \end{aligned}$$

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...) :$
$((x_{post} = x_{pre}+1 \wedge \text{Inv}(\{x\}))[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$(p_{post} = x_{pre} \wedge q_{post} = y_{pre} \wedge \text{Inv}(\{p, q\}))[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)])$
$\oplus$ *Rincy*

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...) :$
$(x_{mid} = x_{pre} + 1 \wedge \text{Inv}(\{x\})[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$p_{post} = x_{mid} \wedge q_{post} = y_{mid} \wedge \text{Inv}(\{p, q\})[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)])$
$\oplus$ *Rincy*

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} \wedge \text{Inv}(\{x, p, q\})) \oplus$ *Rincy*

$\Leftrightarrow$ $\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...) :$
$(x_{post} = x_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} \wedge \text{Inv}(\{x, y, p, q\}))$
$[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$(y_{post} = y_{pre} + 1 \wedge \text{Inv}(\{y\}))[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)]$

$\Leftrightarrow$ $\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...) :$
$(x_{mid} = x_{pre} + 1 \wedge p_{mid} = x_{pre} + 1 \wedge q_{mid} = y_{pre} \wedge \text{Inv}(\{x, y, p, q\}))$
$[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$(y_{post} = y_{mid} + 1 \wedge \text{Inv}(\{y\}))[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)]$

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} \wedge \text{Inv}(\{x, y, p, q\}))$

The semantics of serial schedule S' are derived as follows:

Rincx $\oplus$ *Rincy* $\oplus$ *Raudit*

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge \text{Inv}(\{x\})) \oplus (y_{post} = y_{pre} + 1 \wedge \text{Inv}(\{y\})) \oplus$ *Raudit*

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, \ldots) :$
$(x_{post} = x_{pre}+1 \wedge \text{Inv}(\{x\}))[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$(y_{post} = y_{pre} + 1 \wedge \text{Inv}(\{y\}))[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)])$
$\oplus$ *Raudit*

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, \ldots) :$
$(x_{mid} = x_{pre} + 1 \wedge \text{Inv}(\{x\})[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$y_{post} = y_{mid} + 1 \wedge \text{Inv}(\{y\})[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow (x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)])$
$\oplus$ *Raudit*

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge \text{Inv}(\{x, y\})) \oplus$ *Raudit*

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, \ldots) :$
$(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge \text{Inv}(\{x, y\}))[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$(p_{post} = x_{pre} \wedge q_{post} = y_{pre} \wedge \text{Inv}(\{p, q\}))[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)]$

$\Leftrightarrow$ $(\exists(x_{mid}, y_{mid}, p_{mid}, q_{mid}, \ldots) :$
$(x_{mid} = x_{pre} + 1 \wedge y_{mid} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y\})[(x_{post}, y_{post}, p_{post}, q_{post}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)] \wedge$
$p_{post} = x_{mid} \wedge q_{post} = y_{mid} \wedge \mathrm{Inv}(\{p, q\})[(x_{pre}, y_{pre}, p_{pre}, q_{pre}, ...) \leftarrow$
$(x_{mid}, y_{mid}, p_{mid}, q_{mid}, ...)]$

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y, p, q\}))$

Based on the semantics of schedules S and S' we can now verify whether schedule S is semantic final result equivalent to serial schedule S' and whether schedule S is semantic view equivalent to serial schedule S'. We have $T = T'$ and $P = P'$ and in addition:

$[(\bigcup_{p \in P} R_p, <)]_{DB} = [(\bigcup_{p \in P} R_p, <')]_{DB}$

$\Leftrightarrow$ $[(\exists p_{pre}, p_{post}, q_{pre}, q_{post} : (x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} \wedge \mathrm{Inv}(\{x, y, p, q\})))]_{DB} =$
$[(\exists p_{pre}, p_{post}, q_{pre}, q_{post} : (x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y, p, q\})))]_{DB}$

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y\})) =$
$(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y\}))$

$\Leftrightarrow$ *TRUE*

Hence, we can conclude that schedules S and S' are semantic final state equivalent, and thus schedule $S \in SFSR$. In a similar way we verify whether $S \in SVSR$. We have $T = T'$, $P = P'$ and:

$[(\bigcup_{p \in PS} R_p, <)]_{DB \cup TP} = [(\bigcup_{p \in PS} R_p, <')]_{DB \cup TP}$

$\Leftrightarrow$ $(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} \wedge \mathrm{Inv}(\{x, y, p, q\})) =$
$(x_{post} = x_{pre} + 1 \wedge y_{post} = y_{pre} + 1 \wedge p_{post} = x_{pre} + 1 \wedge q_{post} = y_{pre} + 1 \wedge \mathrm{Inv}(\{x, y, p, q\}))$

$\Leftrightarrow$ *FALSE*

... and therefore we conclude that $S \notin SVSR$. The reason is that the semantics of schedule S and S' are different with respect to transaction parameter q.

3.3.2 Semantic conflict serializability (*SCSR*)

After defining semantic final state serializability and semantic view serializability, we now apply the relevant semantics to the definition of conflict serializability, resulting in the correctness criterion of *semantic conflict serializability*. The primary difference between conventional conflict serializability and semantic conflict serializability is the definition of the conflict relation. We introduce the *semantic conflict relation* as a binary relation on semantics.

Definition 3.3.2.1 (Semantic conflict relation) *The semantic conflict relation SCON on a binary relation of semantics is defined:*

$SCON(R, R') \quad \Leftrightarrow \quad (R \oplus R' \neq R' \oplus R)$

The definition implies that the order of the operations p and q with conflicting semantics R_p and R_q is relevant for the composed semantics of a schedule. Therefore, we will assume that in schedules operations p and q with conflicting semantics are ordered.
We illustrate the definition of *SCON* in the following example.

Example 3.3.2.2 (SCON) *Consider an operation* inc *that increments variable x and a multiplier operation* mul *that doubles the value of variable x, with semantics:*

$$\begin{aligned} R_{inc} &= (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\})) \\ R_{mul} &= (x_{post} = 2x_{pre} \wedge \mathrm{Inv}(\{x\})) \end{aligned}$$

We have :

$SCON(R_{inc}, R_{mul})$

$\Leftrightarrow$ $R_{inc} \oplus R_{mul} \neq R_{mul} \oplus R_{inc}$

$\Leftrightarrow$ $(\exists(x,\ldots) : (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\}))[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge (x_{post} = 2x_{pre} \wedge \mathrm{Inv}(\{x\}))[(x_{pre},\ldots) \leftarrow (x,\ldots)]) \neq$
$(\exists(x,\ldots) : (x_{post} = 2x_{pre} \wedge \mathrm{Inv}(\{x\}))[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\}))[(x_{pre},\ldots) \leftarrow (x,\ldots)])$

$\Leftrightarrow$ $(\exists(x,\ldots) : (x = x_{pre} + 1 \wedge \mathrm{Inv}(\{x\})[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge$
$x_{post} = 2x \wedge \mathrm{Inv}(\{x\})[(x_{pre},\ldots) \leftarrow (x,\ldots)]) \neq$
$(\exists(x,\ldots) : (x = 2x_{pre} \wedge \mathrm{Inv}(\{x\})[(x_{post},\ldots) \leftarrow (x,\ldots)]$
$\wedge\, x_{post} = x + 1 \wedge \mathrm{Inv}(\{x\})[(x_{pre},\ldots) \leftarrow (x,\ldots)])$

$\Leftrightarrow$ $(x_{post} = 2x_{pre} + 2 \wedge \mathrm{Inv}(\{x\})) \neq (x_{post} = 2x_{pre} + 1 \wedge \mathrm{Inv}(\{x\}))$

$\Leftrightarrow$ *TRUE*

Apparently, $SCON(R_{inc}, R_{mul}) \Leftrightarrow$ TRUE and thus both operations should be considered to conflict under this interpretation of their semantics.

The interesting aspect of the semantic conflict relation however, is that weaker semantics for operations can be considered. Then, this also results in a weaker semantic conflict relation. This is illustrated by the following example, where we consider weaker semantics for the increment operation.

Example 3.3.2.3 (Weaker SCON) *Reconsider the increment and multiplier operations from example 3.3.2.2, but now consider the following weaker semantics for operation inc:*

$$R'_{inc} = (x_{post} > x_{pre} \wedge \mathrm{Inv}(\{x\}))$$

Instead of stating that the final value of x is one more than its initial value, we make the weaker statement that the final value of x is larger than its initial value. Again, we calculate SCON, but this time for the semantics R'_{inc} and R_{mul}.

$SCON(R'_{inc}, R_{mul})$

$\Leftrightarrow$ $R'_{inc} \oplus R_{mul} \neq R_{mul} \oplus R'_{inc}$

$\Leftrightarrow$ $(\exists(x,\ldots) : (x_{post} > x_{pre} \wedge \text{Inv}(\{x\}))[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge$
$(x_{post} = 2x_{pre} \wedge \text{Inv}(\{x\}))[(x_{pre},\ldots) \leftarrow (x,\ldots)]) \neq$
$(\exists(x,\ldots) : (x_{post} = 2x_{pre} \wedge \text{Inv}(\{x\}))[(x_{post},\ldots) \leftarrow (x,\ldots)]$
$\wedge (x_{post} > x_{pre} \wedge \text{Inv}(\{x\}))[(x_{pre},\ldots) \leftarrow (x,\ldots)])$

$\Leftrightarrow$ $(\exists(x,\ldots) : (x > x_{pre} \wedge \text{Inv}(\{x\})[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge$
$x_{post} = 2x \wedge \text{Inv}(\{x\})[(x_{pre},\ldots) \leftarrow (x,\ldots)]) \neq$
$(\exists(x,\ldots) : (x = 2x_{pre} \wedge \text{Inv}(\{x\})[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge$
$x_{post} > x \wedge \text{Inv}(\{x\})[(x_{pre},\ldots) \leftarrow (x,\ldots)])$

$\Leftrightarrow$ $(x_{post} > 2x_{pre} \wedge \text{Inv}(\{x\})) \neq (x_{post} > 2x_{pre} \wedge \text{Inv}(\{x\}))$

$\Leftrightarrow$ *FALSE*

This time, we conclude $SCON(R'_{inc}, R_{mul}) \Leftrightarrow FALSE$. *Apparently, with the weaker semantics* R'_{inc} *the operations* inc *and* mul *are not considered to conflict, whereas with the stronger semantics* R_{inc} *for* inc *the operations do conflict.*

Now we have defined a semantic conflict, we can define semantic conflict equivalence analogously to the conventional definition of conflict equivalence.

Definition 3.3.2.4 (Semantic conflict equivalence) *Let* $S = (T, P, <)$ *and* $S' = (T, P, <'$ *) be identical schedules, except for their orders. Let* R_p *be a justified semantics for each operation* $p \in P$. *Schedules* S *and* S' *are semantic conflict equivalent with respect to the semantics* R_p *for each operation* p *if:*

$$\forall p, q \in P : SCON(R_p, R_q) \Rightarrow (p < q = p <' q)$$

In a similar way as the conventional definition, we define the class of semantic conflict serializable schedules.

Definition 3.3.2.5 (Semantic conflict serializable (*SCSR*)) *A schedule is semantic conflict serializable if it is semantic conflict equivalent to a serial schedule.*

The important property of semantic conflict serializability with respect to the conventional definition of conflict serializability is that semantic conflict serializability is defined *in the context of a semantics assignment to operations.* The result we achieved is that concurrency is improved if weaker semantics are considered for the operations. This follows directly from the fact that weaker semantics imply a weaker semantic conflict relation and a weaker semantic conflict relation implies a weaker notion of semantic conflict equivalence. There is however a limit to weakening the operation semantics. We will discuss this in section 3.4.

This leaves us to prove that *SCSR* is a subset of *SVSR*. We prove in a similar way as Theorem 2.2.0.16, but with the following additional lemma.

Lemma 3.3.2.6 (Conflict ordering implies totally ordered semantics) *Let* $(R, <)$ *be a partially ordered semantics. Let* $<'$ *be a total order on* R *with* $< \subseteq <'$. *Then:*

$$((\bigwedge_{R_p, R_q \in R} (SCON(R_p, R_q) \Rightarrow R_p < R_q \vee R_q < R_p)) \Rightarrow (R, <) = (R, <')$$

Proof 3.3.2.7 *We demonstrate that we can iteratively remove the orders from $<'$ on non-semantic conflicting semantics without violating the lemma.*

We start with the case where $<$ is identical to $<'$ without the order $R_p <' R_q$ and with $\neg SCON(R_p, R_q)$. Now, assume that $(R, <) \neq (R, <')$.

By definition 3.2.0.11 we have $(R, <) = (R, <') \cup (R, <^*)$ with $<^*$ a total order with $<^* = < \cup \{(R_q, R_p)\}$. The semantics R_p and R_q must be adjacent in $<^*$, because R_p and R_q are only ordered differently in $<'$ and $<^*$. By assumption, we have $\neg SCON(R_p, R_q)$. But then by definition 3.3.2.1 we have $(R, <') = (R, <^*)$ and thus $(R, <) = (R, <')$.

This reasoning can be iteratively applied until only the orders on the semantic conflicting semantics remain.

Lemma 3.3.2.8 (Semantic conflict equivalence implies semantic equivalence) *Let $(R, <)$ and $(R, <')$ be partially ordered semantics. Let in $(R, <)$ and $(R, <')$ the semantically conflicting pairs be ordered by $<$ and $<'$. if the semantically conflicting pairs in R are ordered the same by $<$ and $<'$ then$(R, <) = (R, <')$*

Proof 3.3.2.9 *(Analogous to the proof of lemma 2.2.0.14)*
By lemma 3.3.2.6 we can assume that $<$ and $<'$ are total orders without loss of generality, and thus we can assume that $(R, <)$ and $(R, <')$ represent some sequential composition of R.

We prove the lemma, by demonstrating that the sequential composition of $(R, <)$ can be rewritten to the sequential composition of $(R, <')$ by interchanging the elements in the composition, without changing the composed semantics. It follows that $(R, <) = (R, <')$.

To demonstrate that this can always be done if the semantically conflicting pairs in R are ordered the same by $<$ and $<'$, consider the composition order at an arbitrary *moment during the procedure of rewriting $(R, <)$:*

$$\ldots \oplus R_i \oplus R_{i+1} \oplus R_{i+2} \oplus \ldots \oplus R_{i+r-1} \oplus R_{i+r} \oplus \ldots \tag{3.2}$$

We assume that for all $j : 1 \leq j < r$, R_i and R_{i+j} are ordered the same by $<'$.

Let R_{i+r} be the first semantics that are ordered differently with respect to R_i. Hence, in $(R, <')$ we have R_{i+r} ordered before R_i. We can correct the composition order by moving R_{i+r} before R_i by iteratively interchanging R_{i+j} with R_{i+r}. By definition 3.3.2.1 the composed semantics are preserved if $\neg SCON(R_{i+j}, R_{i+r})$.

Now, assume that we have some R_{i+k} with $1 \leq k < r$ and $SCON(R_{i+k}, R_{i+r})$. Because the semantically conflicting pairs are ordered the same, we have that $(R, <') = \ldots \oplus R_{i+k} \oplus \ldots \oplus R_{i+r} \oplus \ldots \oplus R_i \oplus \ldots$. But then, R_{i+k} were the first semantics that were ordered differently in (3.2), and not R_{i+r}. Thus, such R_{i+k} cannot exist and R_{i+r} can be moved before R_i without changing the semantics.

The interchange procedure can be repeated, until all semantics are ordered the same as in $(R, <')$. Thus, we demonstrated that $(R, <) = (R, <')$.

Theorem 3.3.2.10 *$SCSR \subset SVSR$*

Proof 3.3.2.11 *For any schedule $S = (T, P, <)$ with semantics R_p assigned to each operation p:*

$S \in SCSR$

$\Rightarrow$ *definition 3.3.2.4*

$\exists S' : S'$ *is a serial schedule and* $\forall p, q \in P : SCON(R_p, R_q) \Rightarrow ((p < q) = (p <' q))$

$\Rightarrow$ *lemma 3.3.2.6*

$\exists S' : S'$ *is a serial schedule and* $(\bigcup_{p \in P} R_p, <) = (\bigcup_{p \in P} R_p, <')$

$\Rightarrow$ *monotonicity of* $[R]_W$

$\exists S' : S'$ *is a serial schedule and* $[(\bigcup_{p \in P} R_p, <)]_{DB \cup TP} = [(\bigcup_{p \in P} R_p, <')]_{DB \cup TP}$

$\Rightarrow$ *definition 3.3.1.4*

$S \in SVSR$

3.3.3 Concurrency control for *SCSR*

We know from conventional theory that conflict serializability can be verified by testing a serialization graph for cycles [35]. As we only changed the definition of the conflict relation, semantic conflict serializability can also be verified by testing a serialization graph. We have to introduce a *semantic serialization graph* similar to a conventional serialization graph, where edges are inserted according to the semantic conflict order instead of the conventional conflict order. The schedule is semantic conflict serializable if and only if the graph is acyclic. A proof for this is analogous to the proof of theorem 2.2.1.2.

The problem with serialization graph testing is that it suffers from quadratic complexity and therefore may cause too much overhead for certain applications. Locking protocols have much less overhead and are generally preferred over serialization graph testing. In particular, two-phase locking is often used.

Two-phase locking can be directly applied to guarantee semantic conflict serializability. For each operation a special lock type is introduced for each type of operation and two locks are considered conflicting if they lock the same object and the corresponding operations semantically conflict. Two-phase locking further requires that when each operation is executed, the corresponding locks on the accessed objects have been acquired and after a lock has been released, no new lock is acquired by the transaction (see definition 2.2.1.4 of two-phase locking).

3.4 Transaction semantic serializability

In the previous sections we discussed how application semantics can be exploited to improve concurrency by defining weaker semantics for operations than their actual semantics. In this section we extend this approach and discuss how application semantics can be exploited at the level of transactions. We demonstrate that with the transaction semantics we can define weaker correctness criteria than defined in the previous section. In addition, we discuss how the transaction semantics can be exploited to improve concurrency.

3.4.1 Semantics of transactions

A transaction can be considered as a complex application state transition. Therefore, we assign semantics to transactions in a similar way as we assign semantics to operations. The main difference however is that the transaction must be defined for any initial state that satisfies the database consistency constraint and the pre-condition of the transaction parameters of T. In addition, the transaction must preserve database consistency. Only in these cases, we consider the semantics for a transaction *correct*.

Definition 3.4.1.1 (Correct transaction semantics) *Let $T = (P, <)$ be a transaction, I the database consistency constraint and B the transaction specific pre-condition on the transaction parameters of T and R a semantics for T. Semantics R are correct for transaction T, if:*

$$((x1_{pre}, \ldots, xm_{pre}), (x1_{post}, \ldots, xm_{post})) \in R :$$
$$(I((x1_{pre}, \ldots, xm_{pre})) \wedge B((x1_{pre}, \ldots, xm_{pre})) \Rightarrow I((x1_{post}, \ldots, xm_{post})))$$

In the sequel of this thesis we will assume that the semantics for transactions are correct.

In the next section, we will discuss how transaction semantics can be exploited to define weaker correctness criteria.

3.4.2 Transaction semantic serializability

Transaction semantics can be used to develop weaker definitions of correctness. We use transaction semantics to define schedules correct, if the composed semantics of a schedule are *no weaker* than the semantics of a sequence of transactions. Because of the implication, we do not adopt a notion of equivalence. Rather we define serializability directly.

We define *transaction semantic final state serializability* as the equivalent of semantic final state serializability with transaction semantics. Again, we interpret the semantics only with respect to the database variables.

Definition 3.4.2.1 (Transaction semantic final state serializability (*TSFSR*)) *Let $S = (T, P, <)$ be a schedule and let R_p be a justified semantics for each operation $p \in P$ and let R_{T_i} be correct semantics for transaction $T_i \in T$. Schedule S is transaction semantic final state serializable if there is a total order $T_1 \ldots T_n$ of T such that:*

$$[(\bigcup_{p \in P} R_p, <)]_{DB} \subseteq [R_{T_1} \oplus \ldots \oplus R_{T_n}]_{DB}$$

In a similar way, we define *transaction semantic view serializability* as semantic view serializability with respect to the composed semantics of transactions. We only consider the database variables DB and transaction parameters TP.

Definition 3.4.2.2 (Transaction semantic view serializability (*TSVSR*)) *Let $S = (T, P, <)$ be a schedule and let R_p be a justified semantics for each operation $p \in P$ and*

let R_{T_i} be a correct semantics for transaction $T_i \in T$. Schedule S is transaction semantic view serializable if there is a total order $T_1 \ldots T_n$ of T such that:

$$[(\bigcup_{p \in P} R_p, <)]_{DB \cup TP} \subseteq [R_{T_1} \oplus \ldots \oplus R_{T_n}]_{DB \cup TP}$$

Although theoretically interesting, the definitions of *TSFSR* and *TSVSR* are of limited practical use. Therefore, we investigate how the transaction semantics can be used in the context of semantic conflict serializability.

Semantic conflict serializability was defined by means of the semantic conflict relation. We discussed in section 3.3.2 that higher degrees of concurrency can be achieved, if weaker semantics are assigned to the operations. However, we did not discuss how weak the semantics assigned to operations can be. This is exactly how we can exploit the transaction semantics: to provide a limit on the weaking of operation semantics. In fact, we require that the assigned semantics of operations must be *sufficient* to guarantee that the result of the transaction is achieved. This is guaranteed, if the composed semantics of the operations imply the semantics of their transaction. This is captured by the following definition.

Definition 3.4.2.3 (Sufficient operation semantics) *Let $T = (P, <)$ be a transaction and let R_p be a semantics for each operation $p \in P$ and let R_T be a correct semantics for transaction T. The assignment of operation semantics is sufficient if:*

$$(\bigcup_{p \in P} R_p, <) \subseteq R_T$$

We illustrate the definition of sufficient operation semantics by the following example.

Example 3.4.2.4 (Sufficient operation semantics) *Reconsider transaction T_1 from the initial example 3.1.0.3 of this chapter. Assume the following semantics for transaction T_1:*

$$R_{T_1} = (valid(n_{pre}) \Rightarrow valid(d_{post}) \wedge d_{post} = a_{post} \wedge \mathrm{Inv}(\{d, a\}))$$

Further, assume that the operation semantics are defined by:

$$\begin{aligned} R_{intake} &= (valid(n_{pre}) \Rightarrow a_{post} = n_{post} \wedge \mathrm{Inv}(\{a\})) \\ R_{deliver} &= (d_{post} = a_{post} \wedge \mathrm{Inv}(\{d\})) \end{aligned}$$

To verify whether the operation semantics are sufficient for R_{T_1}, we apply definition 3.4.2.3:

$$(\{R_{intake}, R_{deliver}\}, \{(R_{intake}, R_{deliver})\}) \subseteq R_{T_1}$$

$$\Leftrightarrow R_{intake} \oplus R_{deliver} \subseteq R_{T_1}$$

$$\Leftrightarrow (\exists(n, d, a, \ldots) : (valid(n_{pre}) \Rightarrow a_{post} = n_{post} \wedge \mathrm{Inv}(\{a\}))[(n_{post}, d_{post}, a_{post}, \ldots) \leftarrow (n, d, a, \ldots)] \wedge (d_{post} = a_{post} \wedge \mathrm{Inv}(\{d\}))[(n_{pre}, d_{pre}, a_{pre}, \ldots) \leftarrow (n, d, a, \ldots)]) \subseteq R_{T_1}$$

$$\Leftrightarrow (\exists(n, d, a, \ldots) : (valid(n_{pre}) \Rightarrow a = n \wedge \mathrm{Inv}(\{a\})[(n_{post}, d_{post}, a_{post}, \ldots) \leftarrow (n, d, a, \ldots)] \wedge d_{post} = a_{post} \wedge \mathrm{Inv}(\{d\})[(n_{pre}, d_{pre}, a_{pre}, \ldots) \leftarrow (n, d, a, \ldots)]) \subseteq R_{T_1}$$

$\Leftrightarrow$ $(valid(n_{pre}) \Rightarrow n_{post} = d_{post} = a_{post} \wedge \text{Inv}(\{a, d\})) \subseteq (valid(n_{pre}) \Rightarrow valid(d_{post}) \wedge d_{post} = a_{post} \wedge \text{Inv}(\{d, a\}))$

$\Leftrightarrow$ *TRUE*

We conclude that the operation semantics R_{intake} *and* $R_{deliver}$ *are sufficient for* R_{T_1}*. Next, we demonstrate that the semantics of the operations can be weakened. For this, we consider the weaker semantics* R'_{intake} *for operation* intake*:*

$R'_{intake} = (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \text{Inv}(\{a\}))$

The assigned semantics for operation intake have been weakened, because its result no longer specifies that $a = n$*, i.e. we have:* $R_{intake} \subset R'_{intake}$*.*

Again, we verify whether the operations are sufficient for R_{T_1}*:*

$(\{R'_{intake}, R_{deliver}\}, \{(R_{intake}, R_{deliver})\}) \subseteq R_{T_1}$

$\Leftrightarrow$ $R'_{intake} \oplus R_{deliver} \subseteq R_{T_1}$

$\Leftrightarrow$ $(\exists(n, d, a, \ldots) :$
$(valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge \text{Inv}(\{a\}))[(n_{post}, d_{post}, a_{post}, \ldots) \leftarrow (n, d, a, \ldots)] \wedge$
$(valid(a_{post}) \wedge d_{post} = a_{post} \wedge \text{Inv}(\{d\}))[(n_{pre}, d_{pre}, a_{pre}, \ldots) \leftarrow (n, d, a, \ldots)]) \subseteq R_{T_1}$

$\Leftrightarrow$ $(\exists(n, d, a, \ldots) :$
$(valid(n_{pre}) \Rightarrow valid(a) \wedge \text{Inv}(\{a\})[(n_{post}, d_{post}, a_{post}, \ldots) \leftarrow (n, d, a, \ldots)] \wedge$
$d_{post} = a_{post} \wedge \text{Inv}(\{d\})[(n_{pre}, d_{pre}, a_{pre}, \ldots) \leftarrow (n, d, a, \ldots)]) \subseteq R_{T_1}$

$\Leftrightarrow$ $(valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge d_{post} = a_{post} \wedge \text{Inv}(\{a, d\})) \subseteq (valid(n_{pre}) \Rightarrow valid(d_{post}) \wedge d_{post} = a_{post} \wedge \text{Inv}(\{d, a\}))$

$\Leftrightarrow$ *TRUE*

We conclude that even with the weaker semantics R'_{intake} *for operation* intake*, the operations semantics are still sufficient. Because* R'_{intake} *is weaker than* R_{intake} *we may have higher degrees of concurrency than when considering* R_{intake} *for operation* intake*.*

With the definition of *sufficient* operation semantics and the definition of *justified* operation semantics, we have defined criteria for the weakest and strongest semantics for operations: the semantics of operations must be no stronger than their actual semantics, but the semantics must be no weaker than their sufficient semantics.

Formally, for each transaction $T = (P, <)$ and semantics R_T for T and actual semantics R_p for operation $p \in P$, and some assigned semantics R'_p for operation p, we have:

$$\bigwedge_{p \in P} (R_p \subseteq R'_p) \text{ and } (\bigcup_{p \in P} R'_p, <) \subseteq R_T$$

In general, the application designer will choose operation semantics no stronger than the sufficient operation semantics, because the weaker the semantics, the higher the degree of concurrency that is achieved. This way, the transaction semantics can be directly exploited to improve concurrency in the context of semantic conflict serializability and there is no need to introduce a new correctness definition. We complete the chapter by the following theorem that states that sufficient operation semantics imply $SCSR \subset TSVSR$.

Theorem 3.4.2.5 *All operations have sufficient semantics in the context of their transactions* $\Rightarrow SCSR \subset TSVSR$

Proof 3.4.2.6 *Let* $S = (T, P, <)$ *be an arbitrary schedule. Let* R_p *be the semantics assigned to each operation* $p \in P$*. Let* R_{T_i} *be the semantics assigned to transaction* $T_i \in T$*.*

We have:

$S \in SCSR$

$\Rightarrow$ *definition 3.3.2.5*

$\exists S' : S'$ *is a serial schedule and* $\forall p, q \in P : SCON(R_p, R_q) \Rightarrow ((p < Sq) = (p < S'q))$

$\Rightarrow$ *lemma 3.3.2.8*

$\exists S' : S'$ *is a serial schedule and* $(\bigcup_{p \in P} R_p, <) = (\bigcup_{p \in P} R_p, <')$

$\Rightarrow$ *definition 3.4.2.3*

$\exists S' : S'$ *is a serial schedule and* $(\bigcup_{p \in P} R_p, <) \subseteq (R_{T_1} \oplus \ldots \oplus R_{T_n})$

$\Rightarrow$ *monotonicity of* $[R]_W$

$\exists S' : S'$ *is a serial schedule and* $[(\bigcup_{p \in P} R_p, <)]_{DB \cup TP} \subseteq [(R_{T_1} \oplus \ldots \oplus R_{T_n})]_{DB \cup TP}$

$\Rightarrow$ *definition 3.4.2.2*

$S \in TSVSR$

3.5 Conclusions

Conventional serializability theory assumes that each transaction must execute in isolation. However, for many applications transactions need not execute in full isolation. So, for these applications the theory is too constraining, resulting in unnecessary loss of concurrency.

Therefore, many other approaches have been suggested in literature to relax isolation by exploiting the semantics of applications. However, in most approaches the definition of correctness is only related informally to the semantics of the applications. A notable exception is the work of [13], which is the first to establish a formal link between correctness of schedules and the semantics of applications. The work however does not exploit established serializability theory. This is unfortunate, because most practical concurrency controls are based on this theory. We fill this gap in this chapter.

We have introduced a formal definition for the semantics of operations and based on this definition, we defined the semantics of a schedule. We defined the semantics of a schedule by means of the composition of operation semantics. The formal definition of the semantics of a schedule allows us to introduce notions of equivalence over schedules that are based on the semantics of the operations. We introduced semantic final state equivalence and semantic view equivalence as the semantic counterparts of the conventional notions of final state equivalence and view equivalence. With these semantic equivalence definitions we introduced corresponding serializability classes.

Next, we exploited the semantics of operations to define the practically relevant class of semantic conflict serializability. This class is the semantic equivalent of conventional conflict serializability, except that the conventional conflict relation is interpreted over the semantics of operations. This semantic conflict relation is weaker than the conventional conflict relation and thus the class of semantic conflict serializable schedules is larger than the conventional conflict serializable schedules. The great appeal of this class, is that conventional concurrency controls can be reused, such as serialization graph testing and locking. In particular, implementations that are based on two-phase locking can be reused, simply by redefining the conflict relation.

In the last section of this chapter we demonstrated how transaction semantics can be exploited to improve concurrency. We introduced a notion of correctness for the assigned semantics to transactions. Based on the semantics of transactions we introduced the novel correctness criteria of transaction semantic final state serializability and transaction semantic view serializability. For semantic conflict serializability we did not introduce a new class. Instead, we introduced the notion of sufficient operation semantics in the context of the semantics of their transaction. This notion provides guidelines to what extent the semantics of operations can be weakened. This is important, because weaker operation semantics result in higher degrees of concurrency in the context of semantic conflict serializability.

In the next chapter we discuss how more application semantics can be exploited by means of the tree model.

Chapter 4

Semantics-based concurrency control for the tree model

4.1 Introduction

In chapter 3 we introduced the new correctness criteria *SFSR*, *SVSR* and *SCSR* based on the semantics of operations. In the second part of the chapter we introduced larger classes by exploiting transaction semantics. Here, we extended the classes *SFSR* and *SVSR* to *TSFSR* and *TSVSR*. The classes *TSFSR* and *TSVSR* defined schedules correct if they imply the semantic result of some serial schedule with respect to their transaction semantics. In contrast to *SFSR* and *SVSR*, we did not extend the class *SCSR*. Instead, we introduced a definition of *sufficient* operation semantics. The definition can be used to find the weakest operation semantics that are still sufficient to implement the semantics of the transaction. This is important, because weaker operation semantics result in higher degrees of concurrency.

In this chapter we extend this approach and aim at a larger class than *SCSR* by exploiting the semantics of *subsets of operations* of transactions, in addition to the semantics of transactions. These extra semantics improve concurrency if subsets of operations are identified with weaker semantics than the semantics of the composed individual operations.

To exploit semantics of subsets of operations, we assign semantics to the tree model.[1] The tree model is suitable for our purpose, because it allows a straightforward application of our semantics in the model and efficient concurrency control protocols are readily available.

The particular aspect of our semantic approach to the tree model is the treatment of dependencies among the operation semantics at different levels. We will describe explicitly how the semantics of non-leaf operations relate to both lower and higher-level operations in a transaction tree. In addition, we demonstrate how existing concurrency control techniques can be applied.

[1] See section 2.4

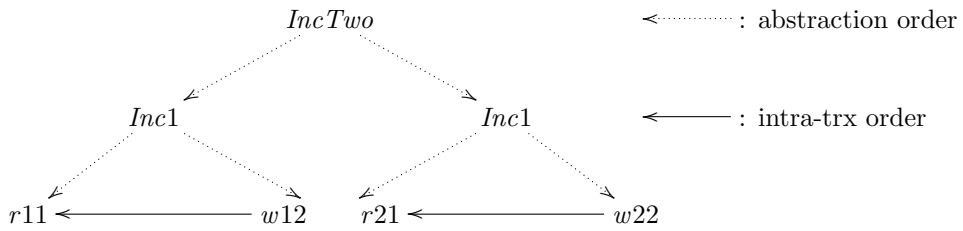

Figure 4.1: Transaction tree T from example 4.2.0.2

4.1.1 Outline

In section 4.2 we demonstrate how the semantics of operations relate in transaction trees. We introduce the notion of sufficient semantics for operations in a transaction tree. Then, we introduce the correctness criterion of semantic tree conflict serializability ($STCSR$). In section 4.3 we propose concurrency controls for semantic tree conflict serializability. We propose subclasses of $STCSR$ that can be scheduled efficiently by means of serialization graph testing and locking protocols. In section 4.3.3 we compare the newly introduced classes.

4.2 Semantics of transaction trees

In chapter 2 we defined transaction trees as a partially ordered and hierarchically structured set of operations.[2] The hierarchical relation between the operations is based on the semantic relationship between the operations. The semantic relationship is similar to the relationship of semantics between the transaction and its operations: for each non-leaf operation the child operations should have sufficient semantics to preserve the semantics of their parent. Consequently, we define sufficient semantics in a transaction tree as follows.

Definition 4.2.0.1 (Sufficient tree operation semantics) *Let $T = (P, <_a, <_i)$ be a transaction tree, with operation r as its root. Let R_p be a semantics for each operation $p \in P$, and R_T a semantics for transaction tree T. The semantics R_p are sufficient, if $R_r \subseteq R_T$ and for all non-leaf operations k:*

$$\left(\bigcup_{p \in \mathrm{Ch}(k)} R_p \right) \subseteq R_k$$

The definition is an application of definition 3.4.2.3, where a non-leaf operation is considered a transaction and its children the operations of the transaction. In the sequel of this thesis we will assume sufficient operation semantics for transaction trees unless noted otherwise.

We illustrate the sufficient operation semantics by means of a classical example of a transaction that increments a counter.

[2] See definition 2.4.0.9.

Example 4.2.0.2 (Sufficient operation semantics) *Let $T = (P, <_a, <_i)$ be a transaction tree that increments a variable x by two, with:*

- $P = \{inctwo, inc1, inc2, r11, w12, r21, w22\}$
- $<_a = \{(inctwo, inc1), (inctwo, inc2), (inc1, r11), (inc1, w12), (inc2, r21), (inc2, w22)\}$
- $<_i = \{(r11, w12), (r21, w22)\}$

Transaction tree T is depicted graphically in figure 4.1.

We consider the following semantics for the transaction tree and its operations:

$$\begin{aligned} R_T &= (x_{post} = x_{pre} + 2 \wedge \mathrm{Inv}(\{x, v, w\})) \\ R_{inc1} &= (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, v\})) \\ R_{r11} &= (v_{post} = x_{pre} \wedge \mathrm{Inv}(\{v\})) \\ R_{w12} &= (x_{post} = v_{pre} + 1 \wedge \mathrm{Inv}(\{x\})) \\ R_{inc2} &= (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, w\})) \\ R_{r21} &= (w_{post} = x_{pre} \wedge \mathrm{Inv}(\{w\})) \\ R_{w22} &= (x_{post} = w_{pre} + 1 \wedge \mathrm{Inv}(\{x\})) \end{aligned}$$

The semantics of the tree are sufficient, because the semantics of each non-leaf operation are implied by the semantics of its children. This is demonstrated as follows.

The semantics R_{r11} and R_{w12} are sufficient for R_{inc1}, because:

$$\begin{aligned} & R_{r11} \oplus R_{w12} \subseteq R_{inc1} \\ \Leftrightarrow\ & (\exists(v, \ldots) : (v_{post} = x_{pre} \wedge \mathrm{Inv}(\{v\}))[(v_{post}, \ldots) \leftarrow (v, \ldots)] \wedge \\ & (x_{post} = v_{pre} + 1 \wedge \mathrm{Inv}(\{x\}))[(v_{pre}, \ldots) \leftarrow (v, \ldots)]) \subseteq (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, v\})) \\ \Leftrightarrow\ & (\exists(v, \ldots) : (v = x_{pre} \wedge \mathrm{Inv}(\{v\})[(v_{post}, \ldots) \leftarrow (v, \ldots)] \wedge \\ & x_{post} = v + 1 \wedge \mathrm{Inv}(\{x\})[(v_{pre}, \ldots) \leftarrow (v, \ldots)]) \subseteq (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, v\})) \\ \Leftrightarrow\ & (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{v, x\})) \subseteq (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, v\})) \\ \Leftrightarrow\ & \mathit{TRUE} \end{aligned}$$

In a similar way, it can be demonstrated that the semantics R_{r21} and R_{w22} are sufficient for R_{inc2}.

The semantics R_{inc1} and R_{inc2} are sufficient for R_{inctwo}. As we have $\neg SCON(R_{inc1}, R_{inc2})$ we can consider an arbitrary order for R_{inc_1} and R_{inc2}. We consider $R_{inc1} < R_{inc2}$:

$$\begin{aligned} & R_{inc1} \oplus R_{inc2} \subseteq R_{inctwo} \\ \Leftrightarrow\ & (\exists(x, \ldots) : (x_{post} = x_{pre} + 1 \wedge \mathrm{Inv}(\{x, v\}))[(x_{post}, \ldots) \leftarrow (x, \ldots)] \wedge \\ & (x_{post} = x_{pre}{+}1 \wedge \mathrm{Inv}(\{x, w\}))[(x_{pre}, \ldots) \leftarrow (x, \ldots)]) \subseteq (x_{post} = x_{pre}{+}2 \wedge \mathrm{Inv}(\{x, v, w\})) \end{aligned}$$

$\Leftrightarrow$ $(\exists(x,\ldots) : (x = x_{pre} + 1 \wedge \mathrm{Inv}(\{x,v\})[(x_{post},\ldots) \leftarrow (x,\ldots)] \wedge$
$x_{post} = x + 1 \wedge \mathrm{Inv}(\{x,w\})[(x_{pre},\ldots) \leftarrow (x,\ldots)]) \subseteq (x_{post} = x_{pre} + 2 \wedge \mathrm{Inv}(\{x,v,w\}))$

$\Leftrightarrow$ $(x_{post} = x_{pre} + 2 \wedge \mathrm{Inv}(\{x,v,w\})) \subseteq (x_{post} = x_{pre} + 2 \wedge \mathrm{Inv}(\{x,v,w\}))$

$\Leftrightarrow$ *TRUE*

Now we have defined sufficiency of semantics for operations of transaction trees, we can introduce a new class of serializable schedules, by adapting *TCSR* (see definition 2.4.1.2) with respect to the semantic conflict relation. We assume that the operation semantics of all transaction trees are sufficient.

Definition 4.2.0.3 (Semantic tree conflict serializable (*STCSR*)) *Let $S = (T, P, <_a, <_d)$ be a tree schedule and R_p the semantics for each operation $p \in P$. Tree schedule S is semantic tree conflict serializable if the semantics of the transaction trees in T are sufficient and schedule S can be transformed into a serial tree schedule by the following rules:*

- commute rule: *for an adjacent pair of ordered leaf operations $p <_d q$ reverse the order, if $\neg SCON(R_p, R_q)$ and not $p <' q$ for some $(P', <') \in T$,*
- compose rule: *remove operations* $\mathrm{Ch}(k)$ *for some operation k, if* $\mathrm{Ch}(k)$ *are leaf operations and are non-interleaved by other leaf operations,*
- order rule: *an unordered pair of leaf operations p and q is ordered either $p <_d q$ or $q <_d p$.*

The difference of *STCSR* with respect to *TCSR* is that the semantic conflict relation is considered instead of the conventional conflict relation. In addition, the semantics of each transaction tree must be sufficient.

We illustrate the class *STCSR* by means of the following example.

Example 4.2.0.4 (STCSR) *Consider transaction trees T and T' that are defined in a similar way as transaction T from example 4.2.0.2. In addition, consider the following schedule S:*

T	$r11$	$w12$					$r21$	$w22$			
T'			$r11'$	$w12'$					$r21'$	$w22'$	

We demonstrate that this schedule is STCSR by applying definition 4.2.0.3.

From example 4.2.0.2 we know that the operation semantics of T and T' are sufficient. In addition, the commute, compose and order rules can be applied to S to transform S into a serial tree schedule as follows:

T	$r11$	$w12$					$r21$	$w22$			
T'			$r11'$	$w12'$					$r21'$	$w22'$	

$\Rightarrow$ *compose r11 and w12 into inc1*

T	$inc1$						$r21$	$w22$			
T'			$r11'$	$w12'$					$r21'$	$w22'$	

$\Rightarrow$ *compose r11' and w12' into inc1'*

T $inc1$ $r21$ $w22$

T' $inc1'$ $r21'$ $w22'$

$\Rightarrow$ *compose r21 and w22 into inc2*

T $inc1$ $inc2$

T' $inc1'$ $r21'$ $w22'$

$\Rightarrow$ *compose r21' and w22' into inc2'*

T $inc1$ $inc2$

T' $inc1'$ $inc2'$

$\Rightarrow$ *commute inc1' and inc2*

T $inc1$ $inc2$

T' $inc1'$ $inc2'$

$\Rightarrow$ *compose inc1 and inc2 into inctwo*

T $inctwo$

T' $inc1'$ $inc2'$

$\Rightarrow$ *compose inc1' and inc2' into inctwo'*

T $inctwo$

T' $inctwo'$

The class of semantic tree conflict serializable schedules is larger than semantic conflict serializable schedules. This is stated by Theorem 4.2.0.5.

Theorem 4.2.0.5 *$SCSR \subset STCSR$*

Proof 4.2.0.6 *Similar to CSR, the class SCSR can be defined in a constructive way by considering the semantic conflict relation instead of the conventional conflict relation. The fact that $SCSR \subseteq STCSR$ then follows directly from the constructive definition 2.2.0.18 of SCSR, because the commute and order rules of SCSR are also included in STCSR and thus we have $SCSR \subseteq STCSR$. The fact that SCSR is a proper subset of STCSR follows from example 4.2.0.4.*

4.3 Concurrency control for *STCSR*

The class *STCSR* is difficult to implement due to the fact that the composition order in a transaction tree is only partially defined by the abstraction order: no order is imposed between sibling operations. This order is however important, because it determines the class of schedules that are accepted. This is illustrated by the following example.

Example 4.3.0.7 (Relevant abstraction order) *Consider transaction T and T' from example 4.2.0.4 and a schedule S:*

T $r11$ $w12$ $r21$ $w22$

T' $r11'$ $w12'$ $r21'$ $w22'$

If transaction T must be completely composed before T' is composed, the schedule is not acceptable. This is due to the fact that the operations of T can only be composed by moving

out the operations $r11'$ and $w12'$ by means of the commute rule. However, both operations conflict with operations $inc1$ and $inc2$, so that this is not possible.

This example therefore shows that the composition order is relevant for the class of schedules to be accepted.

The conventional approach to impose an order of composition is to relate the composition order to the abstraction order within transaction trees [82, 5, 85]. However, the abstraction order is defined within individual transaction trees only, and thus the composition order is independent of the dynamics of a schedule. Such dynamic composition order may however be beneficial for concurrency.

This is illustrated by the following example.

Example 4.3.0.8 (Dynamic composition order) *Consider a transaction tree T that increments a variable x twice by means of two pairs of read and write operations. To improve concurrency, the semantics of the operations of T are captured by abstract operations. The abstract operations are based on the operations of the prefixes of T. The structure of transaction tree T is graphically depicted in figure 4.3.*

The operations have the following semantics:

$$\begin{aligned}
R_T = R_{inctwo} &= (x_{post} = x_{pre} + 2 \wedge \text{Inv}(\{x, v\})) \\
R_{w2} &= (x_{post} = v_{pre} + 1 \wedge \text{Inv}(\{x\})) \\
R_{incr1} &= (x_{post} = x_{pre} + 1 \wedge v_{post} = x_{post} \wedge \text{Inv}(\{x, v\})) \\
R_{r12} &= (v_{post} = x_{pre} \wedge \text{Inv}(\{v\})) \\
R_{inc11} &= (x_{post} = x_{pre} + 1 \wedge \text{Inv}(\{x, v\})) \\
R_{w112} &= (x_{post} = v_{pre} + 1 \wedge \text{Inv}(\{x\})) \\
R_{r111} &= (v_{post} = x_{pre} \wedge \text{Inv}(\{v\}))
\end{aligned}$$

Consider transaction T' that is identical to T, except that it accesses variable v' instead of v. Let schedule S be defined:

T	$r111$	$w112$			$r12$	$w2$		
T'			$r111'$	$w112'$			$r12'$	$w2'$

If the abstraction order of the transactions is considered for the order of composition, the schedule is not acceptable: the first two operations of T and T' are composed simultaneously, which results in the following schedule:

T	$inc11$		$r12$	$w2$		
T'		$inc11'$			$r12'$	$w2'$

After the first composition, the second composition into operations $inc11$ and $inc21$ cannot be performed, because the composition of $inc11$ requires that $inc11'$ is moved before $inc11$, while the composition of $inc11'$ requires that $inc11$ is moved before $inc11'$. The schedule is acceptable however, if operation $incr1'$ is composed only after the complete composition of T. This is demonstrated by the following procedure of commute and compose rules:

T	$r111$	$w112$			$r12$	$w2$		
T'			$r111'$	$w112'$			$r12'$	$w2'$

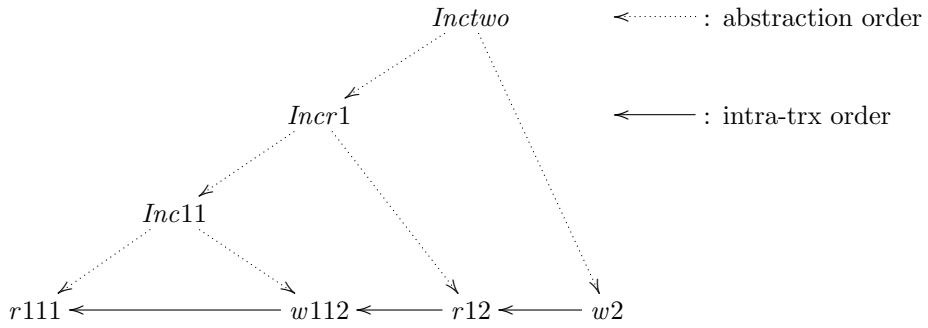

Figure 4.2: Left-most transaction tree of example 4.3.0.8

$\Rightarrow$ *compose r111, w112 into inc11 and r111', w112' into inc11'*

T	$inc11$		$r12$	$w2$	
T'		$inc11'$		$r12'$	$w2'$

$\Rightarrow$ *commute inc11 and inc11'*

T		$inc11$	$r12$	$w2$	
T'	$inc11'$			$r12'$	$w2'$

$\Rightarrow$ *compose inc11, r12 into incr1*

T		$incr1$	$w2$	
T'	$inc11'$		$r12'$	$w2'$

$\Rightarrow$ *compose incr1, w2 into inctwo*

T		$inctwo$	
T'	$inc11'$	$r12'$	$w2'$

$\Rightarrow$ *commute inc11' and inctwo*

T	$inctwo$		
T'		$inc11'$	$r12'$	$w2'$

$\Rightarrow$ *compose inc21, r12' into incr1'*

T	$inctwo$	
T'		$incr1'$	$w2'$

$\Rightarrow$ *compose incr1', w2' into inctwo'*

T	$inctwo$	
T'		$inctwo'$

The schedule is STCSR, because the procedure completes with a single operation for each transaction in the schedule.

To include dynamic composition orders in schedules, we introduce an explicit composition order within the definition of schedules. Such schedules are referred to as *ordered tree schedules*.

Definition 4.3.0.9 (Ordered tree schedule) *An ordered tree schedule is a tuple* $S = (T, P, <_m, <_d)$*, with:*

- T *a set of transaction trees,*
- P *the operations of the set of transaction trees, i.e.* $P = \bigcup_{(P', <'_a, <'_i) \in T} P'$
- $<_m$ *a hierarchical order on* P*, called the composition order, with* $\bigwedge_{(P', <'_a, <'_i) \in T} (<'_a \subseteq <_m)$*,*
- $<_d$ *a partial order on* P*, called the dynamic order, with:*
  - $\bigwedge_{(P', <'_a, <'_i) \in T} (<'_i \subseteq <_d)$*,*
  - $\forall p_i, p_j, p'_i, p'_j \in P : (p_i <_d p_j \wedge p_i <_m p'_i \wedge p_j <_m p'_j) \Rightarrow p'_i <_d p'_j$*.*

With the definition of the ordered tree schedules, we introduce the new class of ordered semantic tree conflict serializable schedules (*OSTCSR*). The class *OSTCSR* is similar to the definition of *STCSR*, except for the composition order $<_m$.

Definition 4.3.0.10 (Ordered semantic tree conflict serializable (*OSTCSR*)) *Let* $S = (T, P, <_m, <_d)$ *be an ordered tree schedule. Ordered tree schedule* S *is ordered semantic tree conflict serializable if the semantics of the transaction trees are sufficient and schedule* S *can be transformed into a serial tree schedule by the following rules:*

- commute rule: *for an adjacent pair of ordered leaf operations* $p <_d q$ *reverse the order, if* $\neg SCON(R_p, R_q)$ *and not* $p <'_i q$ *for some* $(P', <'_a, <'_i) \in T$*,*
- compose rule: *for all operations* k*, remove the operations* $\mathrm{Ch}(k)$*, if* $\mathrm{Ch}(k)$ *are leaf operations according to order* $<_m$ *and are non-interleaved by other leaf operations,*
- order rule: *an unordered pair* p *and* q *of leaf operations according to* $<_d$*, is ordered either* $p <_d q$ *or* $q <_d p$*.*

We demonstrate that *OSTCSR* is a proper subclass of *STCSR*.

Theorem 4.3.0.11 $OSTCSR \subset STCSR$

Proof 4.3.0.12 *The only difference in definition between OSTCSR and STCSR is the compose rule. Thus, the theorem is only false if composition is allowed by OSTCSR but not by STCSR.*

There are two differences in the definitions. First, the composition of OSTCSR requires simultaneous composition of multiple operations at once. Second, composition is only allowed if the removed operations are leaf operations according to $<_m$*.*

The first difference cannot violate $OSTCSR \subseteq STCSR$*, because if a set of operations can be composed at once, they can also be composed in some order.*

The second difference cannot violate $OSTCSR \subseteq STCSR$ *either, because by definition of OSTCSR we have* $\wedge_{(P', <'_a, <'_i) \in T} <'_a \subseteq <_m$ *and thus the composition order according to OSTCSR can always be followed by STCSR.*

The class OSTCSR is a proper subset of STCSR, because example 4.3.0.8 is STCSR but not OSTCSR if $\wedge_{(P', <'_a, <'_i) \in T} <'_a = <_m$*.*

4.3.1 A locking protocol for *OSTCSR*

For scheduling *OSTCSR*, we take a similar approach as two-phase locking for layered systems according to [82]. As we do not have an actual layered system, we derive the notion of a layered system from the composition order to establish what locks for what operations should be compared in the protocol.

The *level of an operation* in an ordered tree schedule is defined as the of an operation to the root operation of the schedule according to the order $<_m$. We say that two operations p and q *share a leve*l, if their level is the same, or the level of operation p is less than the level of operation q and operation p has no children.

With this definition of levels of operations, we can apply layered two-phase locking, with the exception that locks are considered conflicting if operations p and q *share* a level (instead of having the *same* level) and levels are defined with respect to the composition order. We call this locking protocol the *ordered semantic tree two-phase locking protocol* ($OST2PL$).

Protocol 4.3.1.1 (Ordered semantic tree two-phase locking protocol ($OST2PL$)) *Let R_p represent sufficient semantics for each operation $p \in P$ for its transaction tree.*

- Lock acquisition rule: *before operation p is executed, a lock R_p is acquired on all accessed objects,*
- Lock conflict rule: *a lock R_p is only granted if no lock R_q is held by a different parent operation than the parent of p on the same data object with $SCON(q, p)$ and operations p and q share a level with respect to the composition order,*
- Lock release rule: *after lock R_p is released no new lock R_q is acquired if p and q have the same parent or either is the parent of the other with respect to the abstraction order of the transaction of p and q.*

Theorem 4.3.1.2 states that the $OST2PL$ protocol preserves *OSTCSR*.

Theorem 4.3.1.2 *$Gen(OST2PL) \subset OSTCSR$*

Proof 4.3.1.3 *(By contradiction) Consider a schedule $S \in Gen(OST2PL)$, but $S \neq OSTCSR$. This means, that there is a moment in the procedure of commute, compose and order rules that the compose rule cannot be applied while the locking protocol was applied.*

We consider an arbitrary step in the construction of the serial tree schedule, with corresponding tree schedule S_k, such that tree schedule S_k is the first schedule that cannot be composed by the compose rule. We assume that S_k represents the schedule after applying the compose rule k times. The operations in the schedule all share level k, because their parents have not yet been composed. This means, that the operations at level k were all scheduled according to two-phase locking with respect to their parents. By theorem 2.2.1.5 we know that two-phase locking preserves serializability with respect to the parents, and thus the schedule can be rewritten by the commute rule to a serial schedule with respect to the parent operations. This contradicts the fact that the compose rule cannot be applied.

As the locks are only considered among operations that share a level, it may be possible that the locking protocol implies a different serialization order at two different levels. Such different serialization order is inconsistent with the procedure of OSTCSR and must not

occur. To see whether the locking protocol exhibits this property, we assume a conflict order $p_i < p_j$ *in schedule* S_k*, and a conflict order* $p_{jk} < p_{il}$ *at level schedule* S_{k-1}*. The fact that we have* $p_{jk} < p_{il}$ *in* S_{k-1} *means that the lock on* p_{jk} *was released before the lock on* p_{il} *was acquired. According to the locking protocol, this means that the lock on* p_j *was acquired* before *the lock on* p_i *was released. However, this contradicts the order* $p_i < p_j$ *and thus, the locking protocol preserves the serialization order over the levels. We conclude that OSTCSR is preserved, i.e.* $Gen(OST2PL) \subseteq OSTCSR$*.*

The fact that $Gen(OST2PL)$ *is a proper subset of OSTCSR directly follows from theorem 2.2.1.5.*

4.3.2 Prefix semantic tree conflict serializability

The class *OSTCSR* was introduced to remove the non-determinism from the procedure of *STCSR*. We removed the choice on the order of composition by means of the composition order. With the definition of *OSTCSR*, we already noted that the composition order may be specified with respect to some (dynamic) property of a schedule. In this section we consider such a special subclass of *OSTCSR*, where the composition order is based on the dynamic order of the operations in the schedule. We demonstrate that for this subclass a fairly simple locking protocol exists.

We define the subclass by including the dynamic order of a schedule into the composition order. In addition, we assume that the transaction trees in the schedule are *left-most abstracted*. This means, that for each prefix of the transaction, there is a unique abstract operation. Because of this property and the eager composition order, we call this subclass *prefix semantic tree conflict serializable*.

Definition 4.3.2.1 (Prefix semantic tree conflict serializable (*PSTCSR*))
Let $S = (T, P, <_m, <_d)$ *be an ordered tree schedule. Tree schedule* S *is prefix semantic tree conflict serializable, if:*

- $S \in OSTCSR$*,*
- *the transaction trees are left-most abstracted,*
- $<_d \subseteq <_m$*.*

Because we have $<_d \subseteq <_m$, the operations of each transaction tree are composed in the same order as their execution. As a result, for each schedule prefix the maximal semantics of the transaction prefixes are exploited.

A locking protocol for PSTCSR

The locking protocol we propose exploits the fact that each transaction prefix is composed to its corresponding prefix operation. The principle behind the locking protocol is as follows. A lock is held for each prefix operation of each active transaction tree. Because the lock is held, the prefix operations of all other active transaction trees will commute and thus all prefix operations can be interchanged by the commute rule. This way, any prefix operation can be moved to a newly scheduled operation p in the schedule. By definition of *PSTCSR*'s composition rule, the prefix operation and operation p can then be composed

to the corresponding new prefix operation. If this new prefix operation again can acquire a lock, we return to the situation that all prefix operations are locked and a new operation can be scheduled. This way, the locking protocol preserves *PSTCSR*.

Protocol 4.3.2.2 (Prefix locking protocol (*PrL*)) *Let R_p represent the semantics for operation p.*

- lock acquisition rule: *before operation p is executed, a lock R_k is acquired on all accessed objects by operation k , where k is the parent of operation f and p, and f is the current prefix operation of the transaction tree of p,*
- lock conflict rule: *a lock R_k is granted only if no lock R_g is held by a different transaction tree on the same data object with $SCON(R_g, R_k)$,*
- lock release rule: *after acquiring the lock R_k, the lock on R_f is released.*

According to rule one, a new operation p can only be scheduled, if the new prefix operation k does not semantically conflict with the current prefix operations of the active transactions. The lock conflict rule has become very basic without any reference to levels, and just states that two locks conflict if their corresponding operations semantically conflict. Rule three specifies that each new lock replaces the old lock.

Theorem 4.3.2.3 states that the prefix locking protocol preserves *PSTCSR*.

Theorem 4.3.2.3 *$Gen(PrL) \subset PSTCSR$*

Proof 4.3.2.4 *Consider an arbitrary prefix S_{k-1} of schedule S and prefix S_k that is the extension of S_{k-1} by operation p_j of transaction T_j. Now, assume that S_{k-1} and S_k are both $Gen(PrL)$ and S_{k-1} is PSTCSR but S_k is not PSTCSR.*

As prefix S_{k-1} is PSTCSR, it can be rewritten by the commute and composition rules to a schedule with prefix operations for each active transaction. Let operation p_f be the prefix operation of transaction T_j in S_{k-1}.

As S_k is not PSTCSR, it must not be possible to move operations p_f and p_j together in the schedule by the commute rule. This is only not possible if there is a semantic conflict order $R_{p_f} < R_{p_k} < \ldots < R_{p_j}$ for some operation p_k. As S_{k-1} was PSTCSR, operation p_k must be a prefix operation of some active transaction. However, because S_{k-1} was also $Gen(PrL)$, locks for R_{p_f} and R_{p_k} are held simultaneously, and thus R_{p_f} and R_{p_k} must semantically commute. Thus, such conflict order cannot exist and thus p_f can be moved to p_j by the commute rule. This contradicts our assumption.

As a consequence, schedule prefix S_k must be PSTCSR. The reasoning can be applied for arbitrary prefixes of S and thus we conclude $Gen(PrL) \subseteq PSTCSR$.

To see that $Gen(PrL)$ is a proper subclass of PSTCSR, consider the following example:

T_1	$wx1$		$wy1$	
T_2		$wx2$		$wy2$

... where operations wx access variable x and operations wy access variable y and all semantics of the operations semantically conflict. The operations wy can be moved to their corresponding prefix operations wx by the commute rule and thus the schedule is PSTCSR. However, the schedule is not $Gen(PrL)$, because R_{wx1} and R_{wx2} do not commute.

4.3.3 Classification of semantic tree conflict serializability classes

In this section we summarize the discussion above by comparing the classes of semantic tree serializable schedules and the discussed locking protocols.

First, we summarize the results from the previous sections. We already established $SCSR \subset STCSR$ by theorem 4.2.0.5. By definition 4.3.2.1 we have that $PSTCSR \subset OSTCSR$. Also, by theorem 4.3.0.11 we have that $OSTCSR \subset STCSR$. This leaves us to prove the relation of *PSTCSR* with *CSR*.

Theorem 4.3.3.1 states that *SCSR* is a proper subclass of *PSTCSR*.

Theorem 4.3.3.1 *$SCSR \subset PSTCSR$*

Proof 4.3.3.2 *Consider a schedule prefix S_{k-1} of schedule S that is SCSR and PSTCSR, and schedule prefix S_k that is SCSR but not PSTCSR. This must be due to the operation p_i that was added to S_{k-1}.*

Schedule prefix S_{k-1} was PSTCSR and thus can be rewritten to a schedule with the prefix operations of the active transactions. The fact that S_k is not PSTCSR means that operation p_i cannot be moved to the prefix operation of the same transaction. This means that we have a conflict order $R_{p_f} < R_{p_k} < \ldots < R_{p_i}$.

Next, we expand this schedule in an iterative way, such that each prefix operation in the schedule is replaced by its leaf operations, i.e. $p_{f_1} \ldots p_{f_n} \ldots p_{k_1} \ldots p_{k_m} \ldots p_i$. By definition 4.2.0.1, parent operations always have weaker semantics than the semantic result of their children. This means, that for the conflict order $R_{p_f} < R_{p_k}$ there must be two operations p_{f_j} and p_{k_l} that are conflict ordered $R_{p_{f_j}} < R_{p_{k_l}}$. In a similar way, p_{k_l} must be conflict ordered before p_i. But then, a conflict order exists between $R_{p_{f_j}}$ and R_{p_i}, which means that the schedule is not SCSR. Thus, we have a contradiction.

To see that SCSR is a proper subset of PSTCSR we refer to example 4.3.0.8, which is a schedule that is not SCSR, but is PSTSCR.

With the result of Theorem 4.3.3.1, we have:

$$SCSR \subset PSTCSR \subset OSTCSR \subset STCSR \tag{4.1}$$

4.4 Conclusions

In the previous chapter we demonstrated how the semantics of transactions can be exploited to increase concurrency. In this chapter we generalized this approach to the tree model. In addition to that, we introduced new serializability classes, ordered semantic tree conflict serializable and prefix semantic tree conflict serializable, for the tree model that can exploit the application semantics more effectively. We introduced locking protocols that can be scheduled efficiently.

These concurrency control protocols demonstrate that the new serializability classes are not only a theoretic achievement, but also have practical relevance.

Chapter 5

Semantic decomposition

In the previous chapter we adopted serializability as the correctness criterion for schedules. Serializability requires equivalence to a serial execution, where transactions write all of their results in a single database snapshot. For many applications serializability is however not required. Instead, it is often sufficient, if the result of a transaction is written to the database in multiple smaller steps. This weaker interpretation of the semantics of a transaction is exploited by some transaction models by decomposing a transaction into ACID steps. The correctness criterion for schedules can then be weakened from serializability with respect to transactions to serializability with respect to steps. As steps are smaller than transactions, concurrency is increased. Unfortunately, the decomposition of a transaction into ACID steps generally assumes a partitioning of the operations of the transaction. This seriously limits the applicability of these models. In this chapter we extend this approach in two directions. First, we demonstrate how application semantics can be exploited in such model. Second, we demonstrate how the assumption on the partitioning of operations can be relaxed.

First, to exploit application semantics, we introduce a refined definition of the semantics of a transaction, called the *decomposed* semantics of a transaction. The decomposed semantics of a transaction are defined as a *set of application state transitions* that *transform a consistent database state into another consistent database state*. We exploit the refined semantics for transactions by introducing a correctness criterion, called *decomposed semantic conflict serializability*, which is essentially semantic conflict serializability with respect to the set of transitions.

Second, we overcome the limitation of the decomposition of transactions. Whereas in the conventional step decomposition approach *all* semantics of a *subset* of operations are considered, we take the approach where for each transition only *part* of the semantics of *all* operations of the transaction are considered. We demonstrate that for such approach conventional concurrency control techniques can be used.

5.1 Introduction

In conventional transaction models, transactions are considered the unit of transition for the database. This means, that in a serial schedule the result of each transaction appears

in a single database state. For many applications however, it is not required that all results of a transaction appear in the same database state. Instead, it is often sufficient, if the transaction applies its result to the database in multiple smaller transitions.

This is illustrated in the following example.

Example 5.1.0.3 (Multiple transitions) *Consider a transaction T that moves from set s all elements with $ID = p_1$ to set r and moves from set s all elements with $ID = p_2$ to set r. The semantics of transaction T can be described by:*[1]

$$
\begin{aligned}
R_T = (& \\
s_{post} = & \quad \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge \\
r_{post} = & \quad \{e | e \in s_{pre} \wedge \neg(e.ID \neq p_1 \wedge e.ID \neq p_2)\} \wedge Inv(PV_T \cup \{s, r\})
\end{aligned}
$$

The specification states that when T completes, the set s contains the same items as in the initial state, except for those items with $ID = p_1$ and $ID = p_2$, and the set r contains exactly these items. Although the specification is sufficient, it is too strong if it is only required that the elements present in r are removed from s only at a certain point during execution.

The example demonstrates that certain application semantics cannot be effectively captured by a single application state transition. For these applications, serializability with respect to transactions is an overly restrictive correctness criterion, which results in unnecessarily low degrees of concurrency.

To overcome this limitation, several models have been proposed over the years. A survey of the most important models is given in the next section.

5.1.1 Related work

One of the most well known approaches that relaxes serializability is the SAGA model [38, 23]. In the SAGA model, transactions are decomposed into a sequence of ACID steps. A schedule is defined correct, if it is serializable with respect to the steps. Concurrency is improved, because steps are smaller units of serialization than transactions.

The SAGA model has two important limitations. The first limitation is that the steps are totally ordered. This constraint is relaxed to partially ordered steps in more recent models, such as WIDE [44], TSME [39, 40], METEOR [50], X-transactions [78], and virtual transactions [51].

The other limitation is more serious: the model assumes that the operations of a transaction can be partitioned, such that each subset of operations preserves database consistency. This assumption severely limits the decomposition of transactions. The assumption is particularly a problem for transactions that produce a result at the beginning of a transaction that is needed at the end of the transaction. An important category of such transactions include order processing transactions. Such transactions cannot be decomposed according to the SAGA model, or any of the previous models.

An early attempt to relax the assumption of independent units of work is the ConTract model [79]. The ConTract model assumes a partial order of steps, while invariants can

[1] Recall that PV is the set of program variables.

be defined between steps, such that results can be transferred between steps. The main disadvantage of the ConTract model is however that correctness is not well-defined.

Another approach for decomposition of transactions is interleaving specifications. Interleaving specifications provide a means to express what transactions may interleave other transactions and at what points in their execution. An early proposal in this direction is multi-level atomicity [56]. Here interleaving is specified by hierarchically grouping transactions and specifying breakpoints for interleaving points. A more recent approach is relative serializability [75] where steps, called atomic units, are defined relatively between two transactions. Although interleaving specifications are flexible, they require substantial specification effort, because interleaving is generally expressed relatively to other transactions. In addition, the responsibility for correctness of executions is left with the application designer. No guidelines are presented for what interleaving is correct.

More recently, a more formal approach to step-decomposition was proposed [17, 14, 15]. Transactions are decomposed into a sequence of steps similar to the SAGA model, but steps do not necessarily observe and preserve database consistency. Similar to the invariants in the ConTract model, assertions are defined between steps, such that results can be transferred between steps. In contrast to the ConTract model, these assertions are formally derived from the transaction semantics and thus it is possible to reason about correctness in a formal way. Although semantics of applications are exploited to guarantee that the specifications of each transaction are met, the proposed correctness criterion is still defined with respect to transactions as a whole.

A rather different approach is taken in [49], where not the transactions, but the database consistency constraint is considered for decomposition. It is assumed that a database consistency constraint is written in conjunctive normal form, and serializability is only required with respect to each conjunct of the database consistency constraint. In fact, the database is partitioned according to the conjuncts and serializability is only required with respect to each part, but not necessarily in the same order. This weaker correctness criterion is called predicate-wise serializability. Predicate-wise serializability guarantees that each transaction observes and preserves database consistency, but it does not guarantee that each transaction observes serializable results.

5.1.2 Approach

Despite the fact that many proposals appeared in literature to decompose transactions, none of these models is satisfactory. First, the interleaving specifications lack a formal link with the application semantics, and therefore correctness is ill-defined. Second, the models that decompose transactions into ACID steps have limited applicability, because the decomposition assumes a partitioning on the transaction's operations. Third, the models that impose assertions between steps still resort to some notion of serializability with respect to transactions.

The goal of this chapter is to propose a model that allows for decomposition of transactions that overcomes these shortcomings.

To exploit the application semantics, we introduce a more refined definition of the semantics of transactions, called the *decomposed semantics*. The decomposed semantics of a transaction are defined as a *set of transition semantics*. Each transition semantics performs part of work of a transaction on the database, while preserving database consistency. To

overcome the limitation on the decomposition, we do not assume a partitioning on the operations of a transaction with respect to the transitions. Instead, we consider each operation as a candidate to implement a transition and define for each operation the *relevant* semantics in the context of each transition. To exploit the weaker transaction semantics, we define a schedule correct if it implies the semantics of a sequential composition of transitions. We define this correctness criterion *transition semantic serializable*. Because the correctness criterion is based on a notion of serializability, we investigate whether conventional concurrency control techniques can be used for scheduling.

5.1.3 Outline

This chapter is organized as follows. In section 5.2 we introduce the refined definition of operation and transaction semantics, called the *decomposed transaction semantics*. In section 5.3 we introduce correctness criteria that exploit the decomposed transaction semantics, based on final state serializability, view serializability and conflict serializability. In section 5.4 we develop a concurrency control, based on graph testing and locking.

5.2 Semantic operation and transaction decomposition

As mentioned in the introduction, for many applications it may be sufficient if the results are written to the database in multiple steps, rather than that all results are written to a single database snapshot. To describe such semantics more precisely, we introduce the definition of decomposed semantics.

Definition 5.2.0.1 (Semantic decomposition) *Let R be the semantics of an application state transition. Let RR be a set of application state transition semantics and $<$ a partial order on RR. Then $(RR, <)$ is a semantic decomposition of R if:*

$$(RR, <) \subseteq R$$

Essentially, the definition of semantic decomposition is the reverse definition of semantic composition (see definition 3.2.0.11) with the exception that the composed semantics are assumed known. Semantic decomposition can be considered in the context of both operations and transactions. In this context we consider decomposed semantics for operations and transactions in a similar way as conventional semantics. We illustrate the definition of semantic decomposition in the following example.

Example 5.2.0.2 (Decomposed semantics) *Reconsider transaction T from example 5.1.0.3 that moves elements from a set s to set r with semantics:*

$$\begin{aligned} R_T = (& \\ s_{post} = & \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge \\ r_{post} = & \{e | e \in s_{pre} \wedge e.ID = p_1 \wedge e.ID = p_2)\} \wedge Inv(PV_T) \cup \{s, r\}) \end{aligned}$$

As it is not required that the elements are removed from s in the final database state, the application relevant semantics of T can be more accurately described by decomposed semantics. Let RD_T be a decomposed semantics for T, with:

$$RD_T = (\{R_1, R_2, R_3\}, \{(R_1, R_3), (R_2, R_3)\})$$

... where:

$$\begin{aligned} R_1 &= (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \text{Inv}(\{s, u\})) \\ R_2 &= (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \text{Inv}(\{s, v\})) \\ R_3 &= (r_{post} = u_{pre} \cup v_{pre} \wedge \text{Inv}(\{r\})) \end{aligned}$$

... with $u, v \in PV_T$.

To verify that RD_T *is a semantic decomposition of* R_T*, we apply definition 5.2.0.1. We have to prove that:*

$$(R_1 \oplus R_2 \oplus R_3 \cup R_2 \oplus R_1 \oplus R_3) \subseteq R_T$$

We will only prove that: $R_1 \oplus R_2 \oplus R_3 \subseteq R_T$*. The proof for* $R_2 \oplus R_1 \oplus R_3 \subseteq R_T$ *is similar. We have:*

$$\begin{aligned} & R_1 \oplus R_2 \oplus R_3 \subseteq R_T \\ \Leftrightarrow\ & (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \text{Inv}(\{s, u\})) \oplus \\ & (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \text{Inv}(\{s, v\})) \oplus R_3 \subseteq R_T \\ \Leftrightarrow\ & (\exists(s, u, v, \ldots) : \\ & (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \text{Inv}(\{s, u\})) \\ & [(s_{post}, u_{post}, v_{post}, \ldots) \leftarrow (s, u, v, \ldots)] \wedge \\ & (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \text{Inv}(\{s, v\})) \\ & [(s_{pre}, u_{pre}, v_{pre}, \ldots) \leftarrow (s, u, v, \ldots)]) \\ & \oplus R_3 \subseteq R_T \\ \Leftrightarrow\ & (\exists(s, u, v, \ldots) : \\ & (u = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s = s_{pre} - u \wedge \text{Inv}(\{s, u\})[(s_{post}, u_{post}, v_{post}, \ldots) \leftarrow \\ & (s, u, v, \ldots)] \wedge \\ & v_{post} = \{e | e \in s \wedge e.ID = p_2\} \wedge s_{post} = s - v_{post} \wedge \text{Inv}(\{s, v\})[(s_{pre}, u_{pre}, v_{pre}, \ldots) \leftarrow \\ & (s, u, v, \ldots)]) \\ & \oplus R_3 \subseteq R_T \\ \Leftrightarrow\ & (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge v_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID = p_2\} \wedge \\ & s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge \text{Inv}(\{s, u, v\})) \oplus R_3 \subseteq R_T \\ \Leftrightarrow\ & (\exists(s, u, v, r, ...) : \\ & (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge v_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID = p_2\} \wedge \\ & s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge \text{Inv}(\{s, u, v\})) \\ & [(s_{post}, u_{post}, v_{post}, r_{post}, ...) \leftarrow (s, u, v, r, ...)] \wedge \\ & (r_{post} = u_{pre} \cup v_{pre} \wedge \text{Inv}(\{r\}))[(s_{pre}, u_{pre}, v_{pre}, r_{pre}, \ldots) \leftarrow (s, u, v, r, \ldots)]) \subseteq R_T \\ \Leftrightarrow\ & (\exists(s, u, v, r, ...) : \\ & (u = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge v = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID = p_2\} \wedge \\ & s = \{e | e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge \text{Inv}(\{s, u, v\})[(s_{post}, u_{post}, v_{post}, r_{post}, ...) \leftarrow \\ & (s, u, v, r, ...)] \wedge \\ & r = u \cup v \wedge \text{Inv}(\{r\})[(s_{pre}, u_{pre}, v_{pre}, r_{pre}, \ldots) \leftarrow (s, u, v, r, \ldots)]) \subseteq R_T \end{aligned}$$

$\Leftrightarrow$ $(u_{post} = \{e|e \in s_{pre} \wedge e.ID = p_1\} \wedge v_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID = p_2\} \wedge$
$s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge r = \{e|e \in s_{pre} \wedge e.ID = p_1 \wedge e.ID = p_2\} \wedge \text{Inv}(\{s, u, v, r\}))$
$\subseteq (s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge r = \{e|e \in s_{pre} \wedge e.ID = p_1 \wedge e.ID = p_2)\} \wedge Inv(PV_T) \cup \{s, r\})$

$\Leftrightarrow$ *TRUE*

We conclude that RD_T is indeed a semantic decomposition of R_T.

5.3 Correctness for semantically decomposed transactions

Now we have defined decomposed semantics we can introduce novel correctness criteria based on the decomposed semantics of transactions. Essentially, we consider a schedule correct, if its composed semantics are no weaker than some sequential composition of the semantics of the *transitions* of the transaction, as opposed to a sequential composition of *transactions*. Such definition however only makes sense, if each transition performs its intended result and database consistency is preserved. Therefore, we introduce the following property for semantic decomposition.

Definition 5.3.0.3 (Consistent semantic decomposition) *Let $(RR, <)$ be a semantic decomposition of R. $(RR, <)$ is a consistent semantic decomposition, if:*

$$\bigwedge_{R' \in RR, ((x1_{pre}, \ldots, xm_{pre}), (x1_{post}, \ldots, xm_{post})) \in R'} I((x1_{pre}, \ldots, xm_{pre})) \Rightarrow I((x1_{post}, \ldots, xm_{post}))$$

With this additional property, we can introduce our novel correctness criteria. In section 5.3.1 we introduce novel correctness criteria similar to final state and view serializability and in section 5.3.2 we introduce a correctness criterion that is similar to conflict serializability.

5.3.1 Decomposed semantic final state and view serializability

We define correctness notions in a similar way as we did in chapter 3. We define decomposed semantic final state serializable (*DSFSR*) and decomposed semantic view serializable (DSVSR) as the equivalents of *TSVSR* and *TSFSR* in the context of decomposed transaction semantics.

Definition 5.3.1.1 (Decomposed semantic final state serializability (*DSFSR*)) *Let $S = (T_S, P_S, <_S)$ be a schedule and R_p a justified semantics for operation $p \in P_S$. Let $(R_T, <_T)$ be a consistent decomposed semantics for transaction $T \in T_S$. Let $RR = \bigcup_{T \in T_S} R_T$ and $<_r = \bigcup_{T \in T_S} <_T$. Schedule S is decomposed semantic final state serializable if there is a total order $<_R$ with $<_r \subseteq <_R$ and :*

$$[(\bigcup_{p \in P_S} Rp, <_S)]_{DB} \subseteq [(RR, <_R)]_{DB}$$

The definition of *DSFSR* is similar to the definition of *TSFSR* where the notion of transaction is replaced by the definition of transition. The definition of decomposed semantic *view* serializability is identical to *DSFSR*, except for the fact that transaction parameters are now included.

Definition 5.3.1.2 (Decomposed semantic view serializability (*DSVSR*)) *Let* $S = (T_S, P_S, <_S)$ *be a schedule and* R_p *a justified semantics for operation* $p \in P_S$*. Let* $(R_T, <_T)$ *be a consistent decomposed semantics for transaction* $T \in T_S$*. Let* $RR = \bigcup_{T \in T_S} R_T$ *and* $<_r = \bigcup_{T \in T_S} <_T$*. Schedule* S *is decomposed semantic view serializable if there is a total order* $<_R$ *with* $<_r \subseteq <_R$ *and :*

$$[(\bigcup_{p \in P_S} R_p, <_S)]_{DB \cup TP} \subseteq [(RR, <_R)]_{DB \cup TP}$$

In the next section we introduce a class of schedules similar to conflict serializability that can be scheduled efficiently.

5.3.2 Decomposed semantic conflict serializability

In this section we introduce a correctness criterion similar to conflict serializability that is based on semantic decomposition. The criterion is similar to transaction semantic conflict serializability as defined in chapter 3. There is however a complication here compared to the approach with respect to transactions. Whereas each operation is uniquely assigned to a transaction, in the context of decomposed transactions a single operation may implement to the semantics of *multiple* transitions of the transaction. This means, that the semantics of each operation should be considered *in the context* of each transition.

We solve this problem by semantically decomposing operations into multiple transitions, similar to the decomposition of transactions into transitions. This allows us to assign each transition of an operation to a transition of a transaction.

First of all, the decomposition of operations must be justified, i.e. the composition of the operation's semantics must not be stronger than its actual semantics.

Definition 5.3.2.1 (Justified decomposed operation semantics) *Let* R *be the actual semantics of operation* p *and let* $(R', <'_r)$ *be some partially ordered set of semantics for operation* p*.* $(R', <'_r)$ *is a justified decomposed semantics for operation* p *if:*

$$R \subseteq (R', <'_r).$$

In addition to this requirement, the decomposed operation semantics must also be sufficient for the decomposed transaction semantics. We formalize this as follows.

Definition 5.3.2.2 (Sufficient decomposed operation semantics) *Let* $T = (P_T, <_T$ *) be a transaction and* $(R_j, <_j)$ *be a decomposed semantics for operation* $p_j \in P_T$ *and let* $(R_T, <_{R_T})$ *be a consistent decomposed semantics for transaction* T*, such that* $R_{j_k} \in R_j$ *denotes the semantics for operation* p_j *in the context of* $R_{T_k} \in R_T$*. The decomposed semantics* $(R_j, < j)$ *of operations* $p_j \in P_T$ *are sufficient in the context of* $(R_T, <_{R_T})$ *if:*

$$\bigwedge_{R_{T_k} \in R_T} ((\bigcup_{p_j \in P} R_{j_k}, <_T) \subseteq R_{T_k})$$

We illustrate the definition of sufficient operation semantics with the following example.

Example 5.3.2.3 (Sufficient operation semantics) *Reconsider transaction T from example 5.2.0.2. We defined the following decomposed semantics:*

$$RD_T = (\{R_1, R_2, R_3\}, \{(R_1, R_3), (R_2, R_3)\})$$

... with:

$$\begin{aligned} R_1 &= (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \mathrm{Inv}(\{s, u\})) \\ R_2 &= (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \mathrm{Inv}(\{s, v\})) \\ R_3 &= (r_{post} = u_{pre} \cup v_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

Assume that transaction T is implemented as:

$$T = (\{remove_1, remove_2\}, \{(remove_1, remove_2)\})$$

... with operation semantics:

$R_{remove_1} = (s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1\} \wedge r_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge \mathrm{Inv}(\{s, r\}))$

$R_{remove_2} = (s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_2\} \wedge r_{post} = r_{pre} \cup \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge \mathrm{Inv}(\{s, r\}))$

We consider these operation semantics in the context of the semantics R_1, R_2 and R_3. Essentially, for R_1 we only require the first conjunct of R_{remove_1}. Similarly, for R_2 we only require the first conjunct of R_{remove_2}. For R_3 we require the second conjuncts of both R_{remove_1} and R_{remove_2}.

We introduce the decomposed operation semantics:

$$\begin{aligned} RD_{remove_1} &= (\{R_{11}, R_{21}, R_{31}\}, \{(R_{11}, R_{31}), (R_{21}, R_{31})\}) \\ RD_{remove_2} &= (\{R_{12}, R_{22}, R_{32}\}, \{(R_{12}, R_{32}), (R_{22}, R_{32})\}) \end{aligned}$$

... with:

$$\begin{aligned} R_{11} &= (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \mathrm{Inv}(\{s, u\})) \\ R_{21} &= (\mathrm{Inv}(\emptyset)) \\ R_{31} &= (r_{post} = u_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

$$\begin{aligned} R_{12} &= (\mathrm{Inv}(\emptyset)) \\ R_{22} &= (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \mathrm{Inv}(\{s, v\})) \\ R_{32} &= (r_{post} = r_{pre} \cup v_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

The decomposed operation semantics for R_1, R_2 and R_3 are sufficient, because:

$$R_{11} \oplus R_{12} \subseteq R_1$$

$\Leftrightarrow$ $(u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \text{Inv}(\{s, u\})) \oplus (\text{Inv}(\emptyset)) \subseteq$
$(u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \text{Inv}(\{s, u\}))$

$\Leftrightarrow$ *TRUE*

$R_{21} \oplus R_{22} \subseteq R_2$

$\Leftrightarrow$ $(\text{Inv}(\emptyset)) \oplus (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \text{Inv}(\{s, v\})) \subseteq$
$(v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \text{Inv}(\{s, v\}))$

$\Leftrightarrow$ *TRUE*

$R_{31} \oplus R_{32} \subseteq R_3$

$\Leftrightarrow$ $(r_{post} = u_{pre} \wedge \text{Inv}(\{r\})) \oplus (r_{post} = r_{pre} \cup v_{pre} \wedge \text{Inv}(\{r\})) \subseteq R_3$

$\Leftrightarrow$ $(\exists(r, \ldots) : (r_{post} = u_{pre} \wedge \text{Inv}(\{r\}))[(r_{post}, \ldots) \leftarrow (r, \ldots)] \wedge (r_{post} = r_{pre} \cup v_{pre} \wedge$
$\text{Inv}(\{r\}))[(r_{pre}, \ldots) \leftarrow (r, \ldots)]) \subseteq R_3$

$\Leftrightarrow$ $(\exists(r, \ldots) : (r = u_{pre} \wedge \text{Inv}(\{r\})[(r_{post}, \ldots) \leftarrow (r, \ldots)] \wedge$
$r_{post} = r \cup v_{pre} \wedge \text{Inv}(\{r\})[(r_{pre}, \ldots) \leftarrow (r, \ldots)]) \subseteq R_3$

$\Leftrightarrow$ $(r_{post} = u_{pre} \cup v_{pre} \wedge \text{Inv}(\{r\})) \subseteq (r_{post} = u_{pre} \cup v_{pre} \wedge \text{Inv}(\{r\}))$

$\Leftrightarrow$ *TRUE*

The great appeal of the decomposition of operations is that we now consider only part of the semantics of an operation in the context of some transition. As partial operation semantics are generally weaker than the actual semantics of operations, it may be the case that two operations are considered conflicting in the context of a non-decomposed transaction, but may be considered non-conflicting in the context of a single transition of a semantically decomposed transaction. This is illustrated by the following example.

Example 5.3.2.4 (Non-conflicting in context of decomposition) *Consider two transactions T and T' that are defined in an identical way as transaction T in example 5.3.2.3, except that for T' the operations and variables r, u and v are primed. We consider the same semantics for the operations, with:*

$R_{remove_1} = (s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_1\} \wedge r_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge \text{Inv}(\{s, r\}))$
$R_{remove'_2} = (s_{post} = \{e | e \in s_{pre} \wedge e.ID \neq p_2\} \wedge r'_{post} = r'_{pre} \cup \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge \text{Inv}(\{s, r'\}))$

In the context of the semantics R_{remove_1} and $R_{remove'_2}$ we have that the operations semantically conflict, because:

$SCON(R_{remove_1}, R_{remove'_2})$

$\Leftrightarrow$ $(R_{remove_1} \oplus R_{remove'_2} \neq R_{remove'_2} \oplus R_{remove_1})$

$\Leftrightarrow$ $((s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1\} \wedge r_{post} = \{e|e \in s_{pre} \wedge e.ID = p_1\} \wedge \text{Inv}(\{s, r\})) \oplus$
$(s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_2\} \wedge r'_{post} = r'_{pre} \cup \{e|e \in s_{pre} \wedge e.ID = p_2\} \wedge$
$\text{Inv}(\{s, r'\}))) \neq$
$((s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_2\} \wedge r'_{post} = r'_{pre} \cup \{e|e \in s_{pre} \wedge e.ID = p_2\} \wedge$
$\text{Inv}(\{s, r'\})) \oplus$
$(s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1\} \wedge r_{post} = \{e|e \in s_{pre} \wedge e.ID = p_1\} \wedge \text{Inv}(\{s, r\})))$

$\Leftrightarrow$ $(s_{post} = \{e|e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID \neq p_2\} \wedge r_{post} = \{e|e \in s_{pre} \wedge e.ID =$
$p_1\} \wedge r'_{post} = r'_{pre} \cup \{e|e \in s_{pre} \wedge e.ID \neq p_1 \wedge e.ID = p_2\} \wedge \text{Inv}(\{s, r, r'\})) \neq (s_{post} =$
$\{e|e \in s_{pre} \wedge e.ID \neq$
$p_1 \wedge e.ID \neq p_2\} \wedge r_{post} = \{e|e \in s_{pre} \wedge e.ID = p_1 \wedge e.ID \neq p_2\} \wedge r'_{post} = r'_{pre} \cup \{e|e \in$
$s_{pre} \wedge e.ID = p_2\} \wedge \text{Inv}(\{s, r, r'\}))$

$\Leftrightarrow$ *TRUE*

The operations semantically conflict, because while the result on s is the same in either order, the results on r and r' are different in either order.

Now, consider the weaker semantics for the operations in the context of transition R_3. The decomposed semantics of operation $remove_1$ and $remove'_2$ in the context of this transition are defined:

$$\begin{aligned} R_{31} &= (r_{post} = u_{pre} \wedge \text{Inv}(\{r\})) \\ R'_{32} &= (r'_{post} = r'_{pre} \cup v'_{pre} \wedge \text{Inv}(\{r'\})) \end{aligned}$$

Now we have:

$SCON(R_{31}, R'_{32})$

$\Leftrightarrow$ $(R_{31} \oplus R'_{32}) \neq (R'_{32} \oplus R_{31})$

$\Leftrightarrow$ $((r_{post} = u_{pre} \wedge \text{Inv}(\{r\})) \oplus (r'_{post} = r'_{pre} \cup v'_{pre} \wedge \text{Inv}(\{r'\}))) \neq$
$((r'_{post} = r'_{pre} \cup v'_{pre} \wedge \text{Inv}(\{r'\})) \oplus (r_{post} = u_{pre} \wedge \text{Inv}(\{r\})))$

$\Leftrightarrow$ $(r_{post} = u_{pre} \wedge r'_{post} = r'_{pre} \cup v'_{pre} \wedge \text{Inv}(\{r, r'\})) \neq$
$(r_{post} = u_{pre} \wedge r'_{post} = r'_{pre} \cup v'_{pre} \wedge \text{Inv}(\{r, r'\}))$

$\Leftrightarrow$ *FALSE*

Thus, even though the operations $remove_1$ and $remove'_2$ semantic conflict in the context of semantics R_{remove_1} and $R_{remove'_2}$, their semantics in the context of transition R_3 do not conflict.

With the decomposed semantics, we can now introduce a correctness criterion that is based on semantic conflict serializability with respect to the individual transitions of transactions.

Definition 5.3.2.5 (Decomposed semantic conflict serializable (*DSCSR*)) *Let $S = (T_S, P_S, <_S)$ be a schedule. Let $(R_p, <_p)$ be a decomposed semantics of operation $p \in P_S$. Let $(R_T, <_T)$ be a decomposed semantics of transaction $T \in T_S$. Let $R = \bigcup_{T \in T_S} R_T$ and*

$R_P = \bigcup_{p \in PS} R_p$. Schedule S is decomposed semantic conflict serializable with respect to the decompositions if there is a total order $<_R$ on R with $\bigwedge_{T \in T_S}(<_T \subseteq <_R)$ and:

$$\bigwedge_{R_{p_i}, R_{q_j} \in R_P} (R_i <_T R_j \wedge SCON(R_{p_i}, R_{q_j}) \Rightarrow p <_S q)$$

The definition states that there must be a total order of transitions of the transactions in the schedule, such that whenever two operations p and q semantic conflict in the context of two transitions R_i and R_j and R_i is ordered before R_j by the total order, then p must be ordered before q in S.

We illustrate the definition of *DSCSR* by the following example.

Example 5.3.2.6 (DSCSR) *Reconsider transaction T and T' from example 5.3.2.4. We have for T (and T' in a similar way):*

$$T = (\{remove_1, remove_2\}, \{(remove_1, remove_2)\})$$

$$RD_T = (\{R_1, R_2, R_3\}, \{(R_1, R_3), (R_2, R_3)\})$$

... with:

$$\begin{aligned} R_1 &= (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \mathrm{Inv}(\{s, u\})) \\ R_2 &= (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \mathrm{Inv}(\{s, v\})) \\ R_3 &= (r_{post} = u_{pre} \cup v_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

We proposed the following semantic decomposition for the operations $remove_1$ and $remove_2$:

$$\begin{aligned} RD_{remove_1} &= (\{R_{11}, R_{21}, R_{31}\}, \{(R_{11}, R_{31}), (R_{21}, R_{31})\}) \\ RD_{remove_2} &= (\{R_{12}, R_{22}, R_{32}\}, \{(R_{12}, R_{32}), (R_{22}, R_{32})\}) \end{aligned}$$

... with:

$$\begin{aligned} R_{11} &= (u_{post} = \{e | e \in s_{pre} \wedge e.ID = p_1\} \wedge s_{post} = s_{pre} - u_{post} \wedge \mathrm{Inv}(\{s, u\})) \\ R_{21} &= (\mathrm{Inv}(\emptyset)) \\ R_{31} &= (r_{post} = u_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

$$\begin{aligned} R_{12} &= (\mathrm{Inv}(\emptyset)) \\ R_{22} &= (v_{post} = \{e | e \in s_{pre} \wedge e.ID = p_2\} \wedge s_{post} = s_{pre} - v_{post} \wedge \mathrm{Inv}(\{s, v\})) \\ R_{32} &= (r_{post} = r_{pre} \cup v_{pre} \wedge \mathrm{Inv}(\{r\})) \end{aligned}$$

Now, consider schedule S:

T	$remove_1$		$remove_2$	
T'		$remove'_1$		$remove'_2$

From example 5.3.2.4 we know that $SCON(R_{remove_1}, R_{remove'_2})$. In the same way, we can demonstrate that $SCON(R_{remove'_1}, R_{remove_2})$. Consequently, the schedule is not SCSR (see

chapter 3). To verify whether the schedule is DSCSR, we consider the conflicts for the decomposed semantics. It can be verified that in this context, the only conflicting pairs include:

$(R_{11}, R_{31}), (R_{11}, R_{22}), (R_{11}, R'_{11}), (R_{11}, R'_{22}),$
$(R_{22}, R_{32}), (R_{22}, R'_{11}), (R_{22}, R'_{22}),$
$(R_{31}, R_{32}),$
$(R'_{11}, R'_{31}), (R'_{11}, R'_{22}),$
(R'_{22}, R'_{32})

Next, we demonstrate that $S \in DSCSR$ with the total order:

$R_1 <_T R'_1 <_T R_2 <_T R'_2 <_T R_3 <_T R'_3$.

The fact that $R_1 <_T R'_1 <_T R_2 <_T R'_2$ is satisfied by the schedule should be no surprise. This leaves us to demonstrate that the orders on R_3 and R'_3 are also satisfied.

Schedule $S \notin DSCSR$ if one of the following holds:

$$\begin{aligned} R'_1 < R_3 \wedge SCON(R'_{11}, R_{31}) &\Rightarrow remove'_1 < remove_1 \\ R'_2 < R_3 \wedge SCON(R'_{21}, R_{31}) &\Rightarrow remove'_1 < remove_1 \\ R_1 < R'_3 \wedge SCON(R_{12}, R'_{31}) &\Rightarrow remove_2 < remove'_1 \\ R_2 < R'_3 \wedge SCON(R_{22}, R'_{31}) &\Rightarrow remove_2 < remove'_1 \end{aligned}$$

However, the semantic conflicts are all FALSE, and thus $S \in DSCSR$.

This leaves us to prove that any schedule that is in *DSCSR* has the semantics of a sequential composition of the transitions, i.e. $DSCSR \subseteq DSVSR$. This is stated by theorem 5.3.2.7.

Theorem 5.3.2.7 *All operations have justified and sufficient decomposed semantics in the context of their decomposed transaction semantics $\Rightarrow$ $DSCSR \subseteq DSVSR$.*

Proof 5.3.2.8
Let $S = (T_S, P_S, <_S)$ be an arbitrary schedule.
Let $(R_{T_i}, <_{RT_i})$ be the decomposed semantics assigned to transaction $T_i \in T_S$.
Let R_{p_i} be the semantics assigned to operation $p_i \in P_S$.
Let $(R_i, <_i)$ be the decomposed semantics assigned to operation p_i.
Let R_{i_k} be the semantics of operation p_i in the context of transition $R_{T_{i_k}} \in R_{T_i}$.
Let $<_{R_T}$ be a total order on R_T implied by DSCSR.

In addition, consider the following abbreviations:

$$\begin{aligned} R_P &= \bigcup_{p_i \in P_S} Ri \\ R_T &= \bigcup_{T_i \in T_S} R_{T_i} \\ (R_P, <_{rT}) &= \{(R_{i_{j_k}}, R_{r_{s_t}}) | R_{i_{j_k}}, R_{r_{s_t}} \in R_P \wedge R_{T_{i_k}} <_{R_T} R_{T_{r_t}} \wedge R_{i_{j_k}} <_{T_i} R_{r_{s_t}}\} \\ (R_P, <_{rP}) &= \{(R_{i_{j_k}}, R_{r_{s_t}}) | R_{i_{j_k}}, R_{r_{s_t}} \in R_P \wedge p_{i_j} <_S p_{r_s}\} \end{aligned}$$

We have:

$S \in DSVSR$

$\Leftarrow [(\bigcup_{p_i \in P_S} Ri, <_S)]_{DB \cup TP} \subseteq [(R_T, <_{R_T})]_{DB \cup TP}$

$\Leftarrow$ *(monotonicity of* $[R]_W$*)*

$\bigcup_{p_i \in P_S} Ri, <_S) \subseteq (R_T, <_{R_T})$

$\Leftarrow$ *(justified decomposed operation semantics)*

$(R_P, <_{rP}) \subseteq (R_T, <_{R_T})$

$\Leftarrow$ *Lemma 3.3.2.8*

$(R_P, <_{rT}) \subseteq (R_T, <_{R_T}) \wedge S \in DSCSR$

$\Leftarrow$ *(sufficient decomposed operation semantics)*

$S \in DSCSR$

In the next section we introduce concurrency controls for *DSCSR*.

5.4 Concurrency control for *DSCSR*

We consider two types of concurrency control for *DSCSR*. First we discuss serialization graph testing and then we discuss a locking protocol.

5.4.1 Serialization graph testing

Testing for *DSCSR* can be done in a similar way as conventional conflict serializability by introducing a serialization graph that is tested for cycles. There are however two differences. One difference is that the nodes in the graph correspond to transitions instead of transactions. Another difference is that the order among the transitions depends on the semantic conflicts of the decomposed semantics of the operations in the context of the corresponding transitions. We define the decomposed semantic serialization graph as follows.

Definition 5.4.1.1 (Decomposed semantic serialization graph (*DSSG*)) *Let* $S = (T_S, P_S, <_S)$ *be a schedule and* $(R_T, <_T)$ *be a decomposed semantics for each transaction* $T \in T_S$ *and* $(R_p, < p)$ *be a decomposed semantics of each operation* $p \in P$*. Its decomposed semantic serialization graph* $DSSG(S)$ *is a directed graph* (N, V)*, with:*

$$
\begin{aligned}
N &= \bigcup_{T \in T_S} R_T \\
V &= \{(R_i, R_j) | \exists R_{p_i}, R_{q_j} \in \bigcup_{p \in P} R_p : p_i <_S q_j \wedge SCON(R_{p_i}, R_{q_j}) \wedge p_i <_S q_j\}
\end{aligned}
$$

The serialization graph $DSSG(S)$ can be used to test S for *DSCSR*. This is expressed in the serializability theorem below.

Theorem 5.4.1.2 (Decomposed semantic conflict serializability)
$S \in DSCSR \Leftrightarrow DSSG(S)$ *is acyclic*

Proof 5.4.1.3 *(only if) (By contradiction) Assume* $S \in DSCSR$ *but* $DSSG(S)$ *is cyclic. By definition of DSCSR, a total order* $<_R$ *exists on the transaction transitions, such that those pairs of operations are ordered according to this order for which there are conflicting decomposed semantics. If* $DSSG(S)$ *is cyclic, there must be a set of operations, such that they are conflict ordered in a cycle. This order must be consistent with* $<_R$*, which contradicts the fact that* $<_R$ *is a total order. Therefore,* $DSSG(S)$ *cannot contain a cycle.*

(if) (By contradiction) Assume $DSSG(S)$ *is acyclic but* $S \notin DSCSR$*. Let* $<_R$ *be a total order on the transaction transitions that is consistent with the order in* $DSSG(S)$*. If* $S \notin DSCSR$*, then there must be decomposed operation semantics* R_{p_i} *and* R_{q_j} *for operations* p *and* q*, such that* $R_i < R_j \wedge SCON(R_{p_i}, R_{q_j}) \wedge q <_S p$*. However, by definition of* $DSSG(S)$*,* $SCON(R_{p_i}, R_{q_j}) \wedge q <_S p$ *would result in* $(R_j, R_i) \in V$*, which contradicts that* $(R_i, R_j) \in <_R$*.*

The theorem can be used to verify whether a schedule $S \in DSCSR$ by testing $DSSG(S)$ for cycles.

5.4.2 A locking protocol

Testing for cycles in a serialization graph has a complexity that is quadratic in the number of transitions in the schedule. This complexity may be too high for certain applications. Therefore, we propose a locking protocol for *DSCSR*.

We adopt two-phase locking and apply the protocol with respect to each transaction transition. The difference with the conventional locking protocol, is that locks are not acquired for each operation, but instead locks are acquired for the decomposed semantics of each operation.

Under this interpretation, we adapt definition 2.2.1.4 of the two-phase locking protocol, such that locks are defined in the context of the semantic decomposition. We establish the following locking protocol.

Protocol 5.4.2.1 (Decomposed two-phase locking (*D2PL*)) *Let* $(R_T, <_T)$ *be the decomposed semantics for transaction* T *and let* $(R_p, < p)$ *be the decomposed semantics for each operation* p*, where* R_{p_i} *represents the semantics of operation* p *in the context of transition* $R_{T_i} \in R_T$*.*

- Lock acquisition rule: *before operation* p *is executed, a lock* R_{p_i} *is acquired on all objects accessed by* p *for all* $R_{p_i} \in R_p$*.*
- Lock conflict rule: *a lock* R_{p_i} *is granted only if no lock* R_{q_j} *is held by a different transaction with* $SCON(R_{q_j}, R_{p_i})$ *on the same object.*
- Lock release rule: *after a lock* R_{p_i} *is released, no lock* R_{q_i} *is acquired for the same transition. All locks are released as soon as possible.*

Effectively, the protocol applies two-phase locking with respect to each transaction transition and the corresponding decomposed operation semantics. Note that the lock conflict rule is defined with respect to transactions, because it can be assumed that a transaction in

isolation guarantees DSCSR. Because two-phase locking guarantees serializability, *DSCSR* is guaranteed under the locking protocol. The proof is analogous to the conventional two-phase locking case.

5.5 Conclusions

The definition of semantics of a transaction in terms of a single application state transition is too restrictive for many applications. Therefore, we introduce the notion of *decomposed semantics*. With decomposed semantics the semantics of a transaction can be expressed in terms of a partially ordered set of semantics. A transition represents part of the execution of a transaction that performs a state transition on the database while preserving database consistency.

Based on the decomposed semantics of transactions, we introduce novel correctness criteria in line with conventional serializability theory. In particular, we defined decomposed semantic conflict serializability (*DSCSR*). Schedules are decomposed semantic conflict serializable only if their semantics are equivalent to the semantics of a serial execution of transitions. The desirable property of *DSCSR* is that it can be scheduled using conventional concurrency control techniques.

Chapter 6

Semantic forward reducibility

Atomicity requires that either all or none of the results of a transaction appear in the database. Atomicity is generally guaranteed by backward recovery, where all results of the transaction are removed from the database. However, backward recovery is undesirable for long transactions, because much work is lost. Alternatively, backward recovery may also not be possible, if some results cannot be backward recovered. In such cases, forward recovery is often considered as an alternative recovery strategy where atomicity is guaranteed by completing a transaction.

In this chapter we develop a model that supports both backward as well as forward recovery to guarantee atomicity. It follows from our model that there is an inherent trade-off between the knowledge about forward recovery and the degree of concurrency achieved. To maximize concurrency while forward recovery is desired, we require only knowledge about one possible forward recovery alternative. Alternative forward recovery strategies are supported by allowing applications to resume execution after partial rollbacks.

6.1 Introduction

Atomicity of transactions is commonly guaranteed by applying backward recovery, where, upon failure, all effects of a transaction are removed from the database. Although this strategy guarantees atomicity, backward recovery may be undesirable for transactions that produce expensive results, because these results are lost upon failure. Alternatively, for certain applications, backward recovery may not even be possible. This is the case if results are produced that cannot be undone, e.g. if physical objects are modified beyond repair.

In order to support these requirements, many advanced transaction models have been proposed in the literature. We review the most important models below.

6.1.1 Related work

The traditional way of backward recovery in database systems is by means of before-image restore. Here, values of database variables are recorded on persistent storage before they are overwritten. Upon recovery, these old values are written back to the database, such that all

effects of a transaction can be erased. To guarantee atomicity, it is further necessary to erase the effects of transactions that have observed results of aborted transactions. Schedules that satisfy this property are referred to as *recoverable* [45, 63, 21].

Recoverable schedules however have the undesirable property that transactions become vulnerable to failures of other transactions, because they are allowed to observe uncommitted results. A common solution to this problem is to adopt *cascadeless* schedules, where transactions are not allowed to observe uncommitted results. Unfortunately, this results in considerable loss of concurrency. To make this worse, often the more conservative class of *strict* schedules is adopted that not only prohibits transactions to observe uncommitted results, but also prohibits overwriting uncommitted results. The reason for this is that recovery can be implemented more efficiently. However, strict schedules suffer from even less concurrency than cascadeless schedules. In particular, the popular implementation of strict and serializable schedules is strict two phase locking, where locks are held until the end of transactions. It is not difficult to see, that this has severe implications for concurrency.

An important advance in improving concurrency while maintaining recoverability has been made by recognizing that recovery can also be performed by semantically undoing effects of a transaction. Semantic recovery [41, 37, 38] is based on the fact that operations have compensating operations that semantically undo their effects. Then, if the compensating operations are executed in reverse order of the operations in the schedule, the effects are removed from the database. For recoverability it is then sufficient to ensure that for each transaction the necessary compensating operations can be executed. This strategy improves concurrency over the before-image model, because the semantics of compensating operations can be exploited to demonstrate commutativity with other operations in the schedule.

Only recently, a well founded theory has been established that supports this idea, referenced as the unifying theory of concurrency and recovery [69, 6, 76]. It was demonstrated that a larger class of schedules can be considered serializable and recoverable than the conventional serializable and recoverable class by exploiting the semantics of compensating operations and the fact that the pair compensatable operation - compensating operation has null semantics. Even it was demonstrated that relatively large subclasses exist that can be scheduled in a reasonably efficient way.

Concurrency can be improved even more, if the atomicity requirement is interpreted in the context of certain applications. In some cases, the semantic undo does not have to restore the exact before database state, but a state that is *semantically equivalent* in the context of the application can be considered correct as well [37]. This weaker notion of atomicity is referred to as *semantic atomicity*.

One of the first models to overcome the limitations of backward recovery is the flexible transaction model [34]. This model was inspired by the application domain of multidatabases. In this domain it is sometimes not possible to perform backward recovery, such that the model must accommodate for mixed transactions that include compensatable and non-compensatable sub-transactions. To guarantee atomicity, the commit of non-compensatable operations is delayed until the end of the transaction.

This approach however only works under two assumptions. First, all non-compensatable operations must be able to participate in an atomic commit protocol to guarantee atomicity. Second, the commit of non-compensatable operations must be delayed until the end of the transaction. To guarantee recoverability this means that all operations that follow non-compensatable operations must participate in the atomic commit protocol. Both are very

limiting assumptions.

A more general approach to forward recovery was introduced in [57, 58]. It was acknowledged that the commit of non-compensatable operations does not have to be delayed, as long as the transaction can be guaranteed to complete. The model includes three types of operations: compensatable, pivot and retriable operations. Retriable operations are operations that are guaranteed to succeed after a finite number of retries. Pivot operations are operations that are neither compensatable nor retriable. Compensating operations are considered to be retriable. To guarantee atomic execution of transactions, each transaction must consist of at most one pivot sub-transaction. Then atomicity is guaranteed, if for each transaction first all compensatable operations, then the pivot sub-transaction and then the retriable sub-transactions are committed. Multiple pivot sub-transactions can be allowed, as long as they all participate in the same atomic commit protocol [52].

Although the approaches of [57, 58] are considerably more flexible, multiple pivot operations are still required to participate in an atomic commit protocol. In [89] a flexible transaction model was proposed that overcomes this restriction. Whereas the approach in [57, 58] assumes that there is exactly *one* postfix of a transaction after a pivot operation that must consist of retriable operations, the flexible transaction model only requires that there is *some* postfix of a transaction after each pivot operation. Before this postfix is executed, multiple alternative postfixes may be tried of non-retriable, but compensatable operations. As the retriable postfix is only executed if really necessary, this postfix can be considered the *forward recovery* strategy of the transaction. The notion of *semi-atomicity* is introduced by [89] to reference the fact that atomicity is guaranteed while considering the alternative implementations of transactions. A protocol is proposed that guarantees semi-atomicity.

In [88] the correctness with respect to semi-atomicity is extended with F-serializability in the context of multi-database systems. A schedule is F-serializable if it is globally serializable and compensation-interference free. The compensation-interference free requirement means that for each compensatable sub-transaction it should always be possible to execute the compensating operation. This is guaranteed, if sub-transactions are only interleaved between an operation p and its compensating operation p^- if its access set is disjoint with the write set of the operation p. In addition, necessary and sufficient rules are identified to avoid cascading aborts and compensations for reasons of serializability. These rules imply that concurrent flexible transactions do not conflict with any possible future operation.

The approach of [88] avoids cascading aborts, which is a desirable property in general. However, it comes at the cost of reduced concurrency because conflicting operations are delayed until it is ensured that the compensating operations will not be executed. Also, the approach of [88] delays operations that conflict with any possible future operations of a transaction. This requires knowledge of all possible future operations, which may not be available.

The work of [70] improves upon [88] in several aspects. First, it improves concurrency by including the semantics of compensating operations. For this, the unifying theory of concurrency and recovery is adopted. Second, concurrency is improved by relaxing the requirement of cascadeless schedules to cascading aborts. The main gain here is that fewer operations have to be delayed. Third, the requirement is dropped that all future operations need to be known in advance. Here only operations that are actually scheduled are considered for correctness, whereas in [88] all possible future operations need to be considered as well.

This avoids unnecessary blocking of resources. Unfortunately, the strategy works only if a single transactional process at a time is *completing*, i.e. cannot be backward recovered anymore. This seriously compromises concurrency.

6.1.2 Approach

The work of [88] has demonstrated that it is possible to guarantee atomicity for transactions with multiple non-compensatable operations if *all* operations of each transaction are announced to the scheduler. In turn, [70] has demonstrated that *no* operations need to be announced if each pivot operation can be followed by a suffix of retriable operations. The latter approach is appealing because the scheduler does not require knowledge about the forward recovery strategy. However, the lack of knowledge may result in considerable loss of concurrency, because only one active transaction at any point in time is allowed to have executed a non-compensatable operation. The loss of concurrency particularly applies to applications with long transactions.[1]

Therefore, we develop a model where knowledge about forward recovery is exploited to enlarge concurrency. To maximize concurrency in the presence of forward recovery, we guarantee only one forward recovery strategy. We support alternative forward recovery strategies by allowing applications to resume execution after partial backward recovery. However, none of these alternative executions by the application are guaranteed.

Partial backward recovery is supported by extending the conventional model with a *rollback operation*. A rollback operation implies the backward recovery up to the last non-compensatable operation of the transaction.

The scope of the backward recovery can be further customized by means of *durability operations*. Durability operations are special types of non-compensatable operations which only purpose it is to provide persistency of intermediate results and thus protect intermediate results for backward recovery. We note that durability operations are different from conventional savepoint operations, because savepoint operations generally do not provide persistency of results.

Further, we enlarge concurrency, by allowing for cascaded aborts. Generally, cascaded aborts are not supported by transaction models, because much work may be lost in case of an abort. In our case, cascaded aborts can be allowed, because application designers can limit the amount of work lost by means of durability operations.

We present our model as an extension to the unifying theory of concurrency and recovery [76].

6.1.3 Outline

The remainder of this chapter is organized as follows. In section 6.2 we introduce our recovery model. This includes an extension to the conventional execution model with new transactional operations and the distinction between non-compensatable operations and compensatable operations. Then, in section 6.3 we introduce the definition of correctness of schedules in the context of the extended execution model. In section 6.4 we develop concurrency controls for this correctness criterion. We demonstrate the applicability of

[1] In [70] it is argued that this should not be a problem, because pivot operations are often executed at the end of a long transaction. We argue that this is not true for all applications.

the approach to the tree model in section 6.5 and the applicability of the approach to the semantic decomposition in section 6.6. We conclude in section 6.7.

6.2 Recovery model

In this section, we introduce our recovery model. First, we discuss the extensions to the execution model that include the distinction between compensatable and non-compensatable operations and new transactional operations. Second, we formalize the backward and forward recovery in terms of a forward expanded schedule. Third, we introduce a definition for sufficient operation semantics in the context of the recovery model.

6.2.1 Extended execution model

In the unifying theory of concurrency and recovery it is assumed that all operations are compensatable. We consider this assumption too strong. Instead, we assume that the set of database operations *PDB* is partitioned into compensatable operations and non-compensatable operations. Let P be a set of operations, then we denote the subset of compensatable operations in P with $Cmp(P)$ and the subset of non-compensatable operations with $NCmp(P)$.

In addition to this, we introduce a new transactional operation, the *durability operation*, denoted Dr. The durability operation can be considered an intermediate commit of a transaction that protects the results so far from backward recovery. For a set of operations P we denote the subset of durability operations with $Dur(P)$.

To support partial backward recovery, we introduce another new transactional operation: the *rollback operation*, written as Rb. A transaction may consist of multiple rollback operations. Each occurrence of a rollback operation implies backward recovery of the transaction up to the last non-compensatable or durability operation. From here, it is assumed that the local state of the transactional program is restored and the application resumes execution. For a set of operations P, we denote the subset of rollback operations as $Rbk(P)$.

Similar to a rollback operation, we assume that an abort operation also implies backward recovery of the transaction up to the last non-compensatable or durability operation. However, in this case it is assumed that backward recovery is followed by forward recovery. Forward recovery is basically the completion of a transaction by means of the execution of a set of partially ordered operations. This set remains implicit in the execution model. The commit operation remains the same as in the conventional model.

The extensions to the execution model result in an extended definition of a transaction. The transaction with extended recovery semantics is defined as follows.

Definition 6.2.1.1 (Transaction with extended recovery semantics) *A transaction with extended recovery semantics is a tuple* $T = (P, <)$ *with:*

- P *a set of compensatable and non-compensatable data operations, durability operations, rollback operations and one termination operation* Tm*, with* $\mathrm{Tm} \in \{\mathrm{Cm}, \mathrm{Ab}\}$*,*
- $<$ *a partial order on* P*, called the intra-transaction order, with:*
  - $\bigwedge_{p \in P, q \in P \setminus Cmp(P)} p < q \vee q < p$*, and*

– $\bigwedge_{p \in P \setminus \mathrm{Tm}} p < \mathrm{Tm}$.

The definition extends the conventional definition 2.1.0.1 by distinguishing the new compensating operations, non-compensating, durability and rollback operations. In addition, the partial order is defined such, that all non-compensatable, durability and rollback operations are totally ordered with respect to other operations. This way, the backward recovery scope is uniquely defined.

6.2.2 Recovery model

We assume that a rollback operation and an abort operation imply backward recovery up to the last non-compensatable operation. In line with the unifying theory of concurrency and recovery, we assume that backward recovery is implemented by means of compensating operations. We denote p^- to denote the compensating operation of p.

As a notational convenience, we introduce a shorthand to denote the first operation before and after some operation p, including p, that is *not* a compensatable operation in the transaction T of p:

$\mathrm{Pr}(p) =$ the operation q such that $q \in P \setminus Cmp(P) \wedge q \leq p \wedge (\neg \exists r \in P \setminus Cmp(P) : q < r < p)$, and is undefined otherwise.

$\mathrm{Fr}(p) =$ the operation q such that $q \in P \setminus Cmp(P) \wedge p \leq q \wedge (\neg \exists r \in P \setminus Cmp(P) : p < r < q)$, and is undefined otherwise.

Note that we omit the context of p's transaction $(P, <)$ to improve readability. Each operation p in the context of transaction $(P, <)$ is associated with its backward recovery operation p^- if it is not followed by a non-compensatable or commit operation:

$$\begin{aligned} \mathrm{backward}_{(P,<)}(p) &= \{p^-\}, \text{ if } \mathrm{Fr}(p) \in (Rbk(P) \cup \{\mathrm{Ab}\}) \\ \mathrm{backward}_{(P,<)}(p) &= \emptyset, \text{ otherwise} \end{aligned}$$

For abort operations it is assumed that backward recovery is followed by forward recovery. Forward recovery is performed as a set of partially ordered operations that is executed after the backward recovery. This set of operations can be different for each non-compensatable and durability operation of the transaction. Therefore, we associate each non-compensatable and durability operation p with a forward recovery set $FR(p)$ in the context of its transaction. We have $FR(p) = \emptyset$ for compensatable operations p.

For each transaction, we associate the abort operation with the forward recovery set of the last non-compensatable operation. We define:

$$\begin{aligned} \mathrm{forward}_{(P,<)}(\mathrm{Ab}) &= FR(\mathrm{Pr}(\mathrm{Ab})) \cup \{\mathrm{Cm}\} \\ \mathrm{forward}_{(P,<)}(p) &= \emptyset, \text{ if } p \neq \mathrm{Ab} \end{aligned}$$

The definitions of backward and forward can be used to make the recovery of a transaction explicit. As a notational convenience, we additionally define:

$$\begin{aligned} &\mathrm{expand}_{(P,<)}(p) = \\ &\quad (\{p\} \cap (Cmp(P) \cup NCmp(P))) \cup \mathrm{backward}_{(P,<)}(p) \cup \mathrm{forward}_{(P,<)}(p) \end{aligned}$$

A transaction in which the recovery related operations are made explicit is called the forward expanded transaction.

Definition 6.2.2.1 (Forward expanded transaction) *Let $T = (P, <)$ be a transaction. The forward expansion of transaction T is a transaction $FX(T) = (P', <')$ with:*

- $P' = \bigcup_{p \in P} \text{expand}_{(P,<)}(p)$
- $<' = \{(r, s) \mid p, q \in P \wedge p < q \wedge r \in \text{expand}_{(P,<)}(p) \wedge s \in \text{expand}_{(P,<)}(q)\}$

The forward expansion T' of a transaction T is a transaction that includes all database operations of the original transaction T and all recovery related operations. In addition, the intra-transaction order is defined such that the recovery related operations are ordered in the same way as their corresponding rollback or abort operations.

We define *operations* that are followed by a durability, non-compensatable or commit operation as *committed*. Also, operations that are followed by a rollback or abort operation are defined *rollbacked*. An operation that is followed by an abort operation is defined *aborted*.

We define:[2]

$$\begin{aligned} \text{Cm}(p) &= (\text{Fr}(p) \in \mathit{NCmp}(P) \cup \mathit{Dur}(P) \cup \{\text{Cm}\}) \\ \text{Rb}(p) &= (\text{Fr}(p) \in \mathit{Rbk}(P) \cup \{\text{Ab}\}) \\ \text{Ab}(p) &= (\text{Fr}(p) = \text{Ab}) \end{aligned}$$

To reason about the total effect of a transaction on the database, we need not only consider the committed operations, but also the forward recovery operations for aborted transactions. This is captured by the following definition.

Definition 6.2.2.2 (Committed and forward projection of a transaction) *Let $T = (P, <)$ be a transaction. The committed and forward projection of transaction T is a transaction $CF(T) = (P', <')$ with:*

- $P' = \{p \mid p \in P \wedge \text{Cm}(p)\} \cup \{p \mid \text{Ab} \in P \wedge p \in \text{forward}_{(P,<)}(p)\}$,
- $<' = \{(p, q) \mid p, q \in P \wedge p < q \wedge \text{Cm}(p) \wedge \text{Cm}(q)\} \cup \{(p, s) \mid p, q \in P \wedge \text{Cm}(p) \wedge s \in \text{forward}_{(P,<)}(q)\}$

The committed and forward projection of a transaction contains all committed and forward recovery operations. In addition, the intra-transaction order is defined identical to the order of the original transaction, but additionally orders the forward recovery operations at the end of the transaction.

6.2.3 Sufficient semantics of operations

Now we have defined the recovery structure of transactions, we are ready to discuss the semantics of the operations in the context of recovery. We take the same approach as in chapter 3. There we assumed an assignment of semantics to a transaction and considered the semantics of operations sufficient, if their semantics imply the semantics of the transaction. This time however, we apply the definition to the forward expanded transaction.

[2] Again, we omit the transaction context $(P, <)$ to improve readability.

Definition 6.2.3.1 (Expansion sufficient operation semantics) *Let $T = (P, <)$ be a transaction. Let $FX(T) = (P', <')$ be the forward expansion of T. Let R_p be a semantics for each operation $p \in P'$ and let R_T be a correct semantics for transaction T. The operation semantics are expansion sufficient for transaction T if:*

$$(\bigcup_{p \in P'} R_p, <') \subseteq R_T$$

The definition is similar to the definition 3.4.2.3 of sufficient semantics. However, in this case, the definition also constrains the semantics for compensating and forward recovery operations. However, the definition does not imply that each compensating operation removes all effects of its corresponding compensatable operation. This means that backward recovered operations cannot be ignored when verifying for serializability. Instead, their semantics must be considered, which could result in lower degrees of concurrency. This is illustrated by the following example.

Example 6.2.3.2 (Insufficient backward recovery semantics) *Let T be a transaction with $T = (\{inc, \mathrm{Rb}, reset, \mathrm{Cm}\}, \{(inc, \mathrm{Rb}), (\mathrm{Rb}, reset), (reset, \mathrm{Cm})\})$. Let* dec *be the compensating operation of* inc*. Let the following semantics be assigned to the transaction and its operations:*

$$\begin{aligned} R_T &= (x_{post} = 0 \wedge Inv(\{x\})) \\ R_{inc} &= (x_{post} = x_{pre} + 1 \wedge Inv(\{x\})) \\ R_{dec} &= (Inv(\{x\})) \\ R_{reset} &= (x_{post} = 0 \wedge Inv(\{x\})) \end{aligned}$$

Let $FX(T) = (P', <')$ with:

$$\begin{aligned} P' &= \{inc, dec, reset, \mathrm{Cm}\} \\ <' &= \{(inc, dec), (dec, reset), (reset, \mathrm{Cm})\} \end{aligned}$$

The semantics for T are sufficient, because:

$$\begin{aligned} & (\textstyle\bigcup_{p \in P'} R_p, <') \subseteq R_T \\ \Leftrightarrow\ & R_{inc} \oplus R_{dec} \oplus R_{reset} \subseteq R_T \\ \Leftrightarrow\ & \cdots \\ \Leftrightarrow\ & TRUE \end{aligned}$$

However, even though the operation semantics are expansion sufficient, the semantics of R_{dec} are not sufficient to conclude that the semantics of R_{inc} can be ignored in a schedule.

We can resolve the issue, by imposing extra conditions on the semantics of uncommitted operations. In particular, we require that the semantics of a compensatable and its compensating operation are such that their joint semantics implement the *semantic identity relation* on the database. The semantic identity relation R_{Id} is an application specific relation, such that for all $(v, v') \in R_{Id}$ the states v and v' are identical from the perspective of the application.

Definition 6.2.3.3 (Sufficient compensation semantics) *Let R and R' be two semantics. Semantics R and R' are compensation sufficient if:*

$$[R \oplus R']_{DB} \subseteq [R_{Id}]_{DB}$$

In the context of this definition, we can consider operation q to be the compensating operation of compensatable operation p, if R_p and R_q are assigned to operation p and q respectively and R_q compensates for semantics R_p.

Note that definition 6.2.3.3 does not require that compensating operations remove the effects from the local state of a program. This is not required, because it is assumed that the required local state for each compensating operation can be retrieved from persistent storage. In addition, it is assumed that the local state required for forward recovery can be retrieved from persistent storage.

Unfortunately, definition 6.2.3.3 imposes additional constraints not only on the compensating operation, but also on the compensatable operation. This means, that the semantics for a compensatable operation may be stronger than the sufficient semantics. This is illustrated by the following example.

Example 6.2.3.4 (Stronger semantics for compensatable operations) *Assume operations* inc *and* dec *with semantics:*

$$\begin{aligned} R_{inc} &= (x_{post} > x_{pre} \wedge Inv(\{x\})) \\ R_{dec} &= (x_{post} = x_{pre} - 1 \wedge Inv(\{x\})) \end{aligned}$$

We verify whether R_{dec} is sufficient for dec to be a compensating operation for operation inc with semantics R_{inc}. We have:

$$\begin{aligned} & R_{inc} \oplus R_{dec} \subseteq R_{Id} \\ \Leftrightarrow\ & (x_{post} > x_{pre} \wedge Inv(\{x\})) \oplus (x_{post} = x_{pre} - 1 \wedge Inv(\{x\})) \subseteq R_{Id} \\ \Leftrightarrow\ & \exists (xmid, \ldots) : (x_{post} > x_{pre} \wedge Inv(\{x\}))[(x_{post}, \ldots) \leftarrow (xmid, \ldots)] \wedge (x_{post} = x_{pre} - 1 \wedge Inv(\{x\}))[(x_{pre}, \ldots) \leftarrow (xmid, \ldots)] = Id \\ \Leftrightarrow\ & \exists (xmid, \ldots) : (xmid > x_{pre} \wedge Inv(\{x\})[(x_{post}, \ldots) \leftarrow (xmid, \ldots)] \wedge x_{post} = xmid - 1 \wedge Inv(\{x\})[(x_{pre}, \ldots) \leftarrow (xmid, \ldots)] \subseteq R_{Id} \\ \Leftrightarrow\ & (x_{post} > x_{pre} - 1 \wedge Inv(\{x\})) \subseteq R_{Id} \\ \Leftrightarrow\ & FALSE \end{aligned}$$

We conclude that the semantics of R_{inc} and R_{dec} are not *sufficient for dec to be a compensatable operation for* inc*. This means, that stronger semantics are required. In case R_{dec} are the actual semantics of dec, but R_{inc} are not the actual semantics of inc, we can only consider stronger semantics for operation inc. For example, if the actual semantics for inc would be R'_{inc}, with:*

$$R'_{inc} = (x_{post} = x_{pre} + 1 \wedge Inv(\{x\}))$$

... then the semantics for both operations would be sufficient for dec *to be the compensating operation of* inc*, since we have:*

$R_{inc} \oplus R_{dec} \subseteq R_{Id}$

$\Leftrightarrow (x_{post} = x_{pre} + 1 \wedge Inv(\{x\})) \oplus (x_{post} = x_{pre} - 1 \wedge Inv(\{x\})) \subseteq R_{Id}$

$\Leftrightarrow \cdots$

$\Leftrightarrow \text{Inv}(\emptyset) \subseteq R_{Id}$

$\Leftrightarrow TRUE$

The example illustrates that in case of semantic compensation stronger semantics may have to be considered in the context of backward recovery. This may have a negative effect on the degree of concurrency, even if no failures occur.Fortunately, we can ignore these stronger semantics for compensatable operations that have committed.

In the remainder of this chapter, we assume expansion sufficient semantics for each operation p and sufficient compensation semantics for uncommitted compensatable operations and their corresponding compensating operations.

6.3 Semantic forward reducibility (*SFRED*)

Now we have defined backward and forward recovery of transactions, we can define correctness of schedules for the extended execution model. We redefine a schedule for the extended execution model in a similar way as a conventional schedule. To reason about the semantics of the backward and forward recovery in the schedule, we introduce a *forward expanded schedule*. The forward expanded schedule is defined in a similar way as an expanded schedule.[3] The difference now is that we consider the forward expansions of transactions in the extended execution model.

The forward expanded schedule is defined as follows.

Definition 6.3.0.5 (Forward expanded schedule) *Let $S = (T_S, P_S, <_S)$ be a schedule. The forward expansion of S is a schedule $FX(S) = (T'_S, P'_S, <'_S)$ with:*

- $T'_S = \bigcup_{T \in T_S} FX(T)$
- $P'_S = \bigcup_{(P', <') \in T'_S} P'$
- $<'_S = \{(r, s) \mid (P_p, <_p), (P_q, <_q) \in T_S \wedge p \in P_p \wedge q \in P_q \wedge p <_S q \wedge r \in \text{expand}_{(P_p, <_p)}(p) \wedge s \in \text{expand}_{(P_q, <_q)}(q)\}$

The forward expansion of a schedule S contains all forward expansions of the transactions T_S of S. In addition, the operations that are in S and $FX(S)$ are ordered the same. The newly introduced recovery operations are ordered in the same way as their originating rollback or abort operation was ordered. In analogy of transactions, we additionally define the committed and forward projection of a schedule.

[3] See definition 2.3.0.8

Definition 6.3.0.6 (Committed and forward projection of a schedule)
Let $S = (T_S, P_S, <_S)$ be a schedule and let $FX(S) = (T_X, P_X, <_X)$ be its forward expansion. The committed and forward projection of schedule S is a schedule $CF(S) = (T_C, P_C, <_C)$ with:

- $T_C = \bigcup_{T \in T_S} CF(T)$
- $P_C = \bigcup_{(P,<) \in T_C} P$
- $<_C = \{(p, q) | p, q \in P_C \wedge p <_X q\}$

With these definitions, we can now reason about the correctness of schedules in the presence of rollback and abort operations. We take a similar approach as discussed in section 2.3. We define a schedule *semantic forward reducible*, if its forward expansion can be rewritten to a serial schedule by the commute, compensate and order rules.

Definition 6.3.0.7 (Semantic forward reducible schedule (*SFRED*))
Let $S = (T_S, P_S, <_S)$ be a schedule. Let $FX(S) = (T'_S, P'_S, <'_S)$ be the forward expanded schedule of S. For each operation $p \in P'_S$, let R_p be a semantics for operation p that are sufficient in the context of its transaction and compensation sufficient. Schedule S is semantic forward reducible if $FX(S)$ can be rewritten to a serial schedule of transactions with the following rules:

- commute rule: *for an adjacent ordered pair of operations $p_i <'_S p_j$ reverse the order, if $\neg SCON(R_{p_i}, R_{p_j})$ and not $p_i <'_k p_j$ for some $(P'_k, <'_k) \in T'_S$.*
- compensate rule: *remove any adjacent ordered pair $p_i <'_S p_i^-$, where p_i^- is the compensating operation of p_i.*
- order rule: *an unordered pair p_i and p_j is ordered either $p_i <'_S p_j$ or $p_j <'_S p_i$.*

We illustrate semantic forward reducibility by means of the following example.

Example 6.3.0.8 (SFRED) *Consider two transactions T and T', with:*

$$\begin{aligned} T &= (\{intake, deliver, \text{Ab}\}, \{(intake, deliver), (deliver, \text{Ab})\}) \\ T' &= (\{update, \text{Cm}\}, \{(update, \text{Cm})\}). \end{aligned}$$

We assume that operation deliver *is compensatable, and operation* intake *is non-compensatable with $FR(intake) = \{bill\}$. We assign the following operation semantics:*

$$\begin{aligned} R_{intake} &= (valid(n_{pre}) \Rightarrow valid(a_{post}) \wedge Inv(\{a\})) \\ R_{deliver} &= (d_{post} = a_{pre} \wedge p_{post} = d_{pre} \wedge Inv(\{d, p\})) \\ R_{deliver^-} &= (d_{post} = p_{pre} \wedge Inv(\{d\})) \\ R_{bill} &= (valid(b_{post}) \wedge Inv(\{b\})) \\ R_{update} &= (valid(a_{post}) \wedge Inv(\{a\})) \end{aligned}$$

It can be verified that $(R_{intake}, R_{update}), (R_{update}, R_{deliver}) \in SCON$. Next, consider schedule S:

T	*intake*			*deliver*	Ab		
T'		*update*	Cm				

We verify whether $S \in SFRED$.

First, we construct the forward expanded schedule $FX(S)$:

T	*intake*			*deliver*	*deliver*$^-$	*bill*	Cm
T'		*update*	Cm				

Then, we verify whether it can be rewritten to a serial schedule of transactions:

T	*intake*			*deliver*	*deliver*$^-$	*bill*	Cm
T'		*update*	Cm				
$\Leftrightarrow$ *(compensate rule)*							
T	*intake*			*bill*	Cm		
T'		*update*	Cm				
$\Leftrightarrow$ *(commute rule)*							
T	*intake*	*bill*	Cm				
T'				*update*	Cm		

As the resulting schedule is a serial schedule of transactions T and T', we conclude that $S \in SFRED$.

6.3.1 Limited knowledge about forward recovery

So far, we assumed that the semantics of the forward recovery are known. For some applications, it may however be the case that it is impossible or hard to specify how forward recovery will be performed. One can think of applications where human intervention is required to perform the forward recovery.

These cases can be supported by the model by extending the forward recovery set with all operations that may be required for the forward recovery. Alternatively, an artificial operation can be introduced that captures the semantics of all these operations. In the extreme, if *no* knowledge is available about the exact forward recovery, the only semantics that can be assumed is that database consistency is restored. For this we can use the following semantics for the forward recovery set:

$$F_I = I[(x1, \ldots, xm) \leftarrow (x1_{post}, \ldots, xm_{post})] \wedge \text{Inv}(DB)$$

. . . with I the database consistency constraint.

6.4 Concurrency control for *SFRED*

In this section, we develop a concurrency control for *SFRED* based on graph testing and a hybrid concurrency control based on locking and graph testing.

6.4.1 Graph testing protocol

We consider a graph testing protocol for *SFRED* that consists of two phases. The first phase verifies reducibility by removing all compensating operations from the forward expanded schedule. The second phase verifies whether the reduced schedule is serializable.

Reducibility of a schedule can be verified in a similar way as verifying *RED* by reducing the conflict graph of the forward expanded schedule (see section 2.3.1). We construct a conflict graph of the forward expansion of the original schedule S. We refer to this graph as the *forward expanded conflict graph* $FXCG(S)$, with $FXCG(S) = CG(FX(S))$. Graph $FXCG(S)$ can be reduced by the conflict graph reduction protocol (see definition 2.3.1.4). If after completion of the procedure $FXCG(S)$ does not contain any node that represents a compensating operation, then $FXCG(S)$ is said to be *reducible*. If $FXCG(S)$ is reducible, we need to verify that the remaining schedule is serializable in order to verify *SFRED*. Assuming that the backward recovered operations can be reduced by the reduction protocol, the remaining schedule is identical to $CF(S)$. If $CF(S)$ is also serializable, then $S \in SFRED$.

Serializability can be verified by testing the serialization graph of the committed and forward recovery projection of schedule S for cycles. We define such serialization graph as the *committed forward serialization graph* $CFSG(S)$, with $CFSG(S) = SG(CF(S))$.

Reducibility of $FXCG(S)$ and acyclicity of $CFSG(S)$ are necessary and sufficient for $S \in SFRED$. This is expressed by theorem 6.4.1.1 .

Theorem 6.4.1.1 *$S \in SFRED \Leftrightarrow$ ($FXCG(S)$ is reducible and $CFSG(S)$ is acyclic)*

Proof 6.4.1.2 *($\Rightarrow$) Assume $S \in SFRED$ and $FXCG(S)$ is not reducible. Because $S \in SFRED$, there is an order of commute and compensate rules that reduces the forward expanded schedule S' of S. Let operations p_i and p_i^- be the next pair of operations that are reduced. Then, they can be moved together by the commute rule and subsequently removed by the compensate rule. If they can be moved together, $FXCG(S)$ cannot contain a conflict path between p_i and p_i^- and thus the operations p_i and p_i^- can be removed from $FXCG(S)$. Because $S \in SFRED$ this can also be done for the next pair, until all pairs p_i and p_i^- are removed. This implies that $FXCG(S)$ must be reducible.*

Assume $S \in SFRED$ and $CFSG(S)$ contains a cycle. Because $S \in SFRED$, $FX(S)$ can be rewritten to $CF(S)$ by the commute and order rules. In addition, $S \in SFRED$ implies that there is a procedure of commute rules that interchanges the remaining operations such that a serial schedule remains. Such interchange procedure does not change the order of conflicting operations, such that $CFSG(S)$ must have the same conflict order as a serial schedule. This contradicts that $CFSG(S)$ contains a cycle.

($\Leftarrow$) Assume $FXCG(S)$ is reducible, $CFSG(S)$ is acyclic and $S \notin SFRED$. If $S \notin SFRED$, then:

a) either some compensating operation cannot be removed from the forward expansion by the compensate rule, or

b) the remaining schedule cannot be written to a serial schedule.

Case a). If $FXCG(S)$ is reducible, then there is an iteration of protocol 2.3.1.4 where each time two operations p_i and p_i^- can be identified such that there is no conflict path between p_i and p_i^-. If there is no conflict path between p_i and p_i^-, the operations can be

moved together by means of the commute rule and subsequently removed by the compensate rule. This means that all compensating operations can be removed by some procedure of commute and compensate rules.

Case b). If the remaining schedule cannot be rewritten to a serial schedule, then there must be a transaction with operations p_i and p_j such that p_i and p_j cannot be moved together by the commute rule. This means that there is a conflict path $p_i < p_k < \ldots < p_m < p_j$ over operations of different transactions. This however contradicts that $CFSG(S)$ is acyclic.

Thus, if $FXCG(S)$ is reducible and $CFSG(S)$ is acyclic we must have that $S \in SFRED$.

By theorem 6.4.1.1 we can verify *FRED* by reducing $FXCG(S)$ and verifying that $CFSG(S)$ has no cycles.

6.4.2 Hybrid protocol

The complexity of the graph testing protocol is at least polynomial, due to the serialization graph testing. This may be too high for many applications. A more efficient protocol can be developed if we use a combination of locking and graph testing.

For the protocol to guarantee *SFRED*, we take a similar approach as the development of protocol 2.3.1.8 for *RED* in section 2.3.1 . The development of protocol 2.3.1.8 for *RED* was constructed based on the classes *BSF* and *CSR*, such that $BSF \cap CSR \subset RED$. The hybrid protocol is essentially based on the fact that *BSF* can be verified by graph testing and *CSR* can be guaranteed by two-phase locking.

In a similar way, for guaranteeing *SFRED* instead of *RED*, we introduce a class *semantic forward recovery backward safe* (*SFBSF*) as the equivalent of *BSF* in our recovery model. As the equivalent of *CSR*, we introduce the class of committed and forward recovery semantic conflict serializable schedules (*CFSCSR*). We will demonstrate that $SFBSF \cap CFSCSR \subseteq SFRED$. The extension of *SFBSF* with respect to *BSF* is done by treating non-compensatable operations in a similar way as commit operations in *BSF*. In particular, the first non-compensatable operation $\mathrm{Fr}(p)$ after a compensatable operation p is considered as 'its' commit. In a similar way, $\mathrm{Pr}(p)$ is the last non-compensatable before operation p. As a result, the forward recovery set that would be executed if operation p aborts, is defined by $FR(\mathrm{Pr}(p))$. In addition, forward recovery and rollback operations are added to the definition.

We define *SFBSF* as follows.

Definition 6.4.2.1 (Forward recovery backward safe (*SFBSF*)) *Let $S = (T, P, <_S)$ be a schedule. For each operation $p \in P$, let R_p be the semantics assigned to operation p. Schedule S is forward recovery backward safe if:*

$\forall T_i, T_j \in T, p_i \in P_i, p_j \in P_j, T_i \neq T_j, T_i = (P_i, <_i), T_j = (P_j, <_j)$:

- $p_i <_S p_j$
- $(\mathrm{Fr}(p_i), p_j) \notin <_S$
- $SCON(R_{p_j}, R_{p_i^-})) \Rightarrow$
  - $\mathrm{Cm}(p_j) \Rightarrow (\mathrm{Cm}(p_i) \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j))$

  - $\mathrm{Rb}(p_i) \Rightarrow (\mathrm{Rb}(p_j) \wedge (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S)$

- $(\bigvee_{p_{i_s} \in FR(\mathrm{Pr}(p_i))} SCON(R_{p_j}, R_{p_{i_s}})) \Rightarrow$
  - $\mathrm{Cm}(p_j) \Rightarrow (\neg \mathrm{Ab}(p_i) \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j))$
  - $\mathrm{Ab}(p_i) \Rightarrow (\mathrm{Rb}(p_j) \wedge (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S)$

- $(\bigvee_{p_{j_s} \in FR(\mathrm{Pr}(p_j))} SCON(R_{p_{j_s}}, R_{p_i^-})) \vee$
  $(\bigvee_{p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i))} SCON(R_{p_{j_s}}, R_{p_{i_s}})) \Rightarrow$
  - $\mathrm{Ab}(p_j) \Rightarrow (\mathrm{Fr}(p_j), \mathrm{Fr}(p_i)) \notin <_S$

The class *SFBSF* imposes an order on commit, rollback and abort operations, such that the compensating operations and forward recovery operations can always be moved backwards in the forward expanded schedule.

The first implication in the definition guarantees that any backward recovery operation of T_i can be moved backward by delaying the commit of operation p_j. This way, p_j can always be backward recovered when p_i must be backward recovered.

The second implication guarantees that any forward recovery operation of T_i can be moved backward by delaying the commit of operation p_j.

The third implication considers the case where p_j is aborted. In this case, backward recovery of p_i may be jeopardized by a forward recovery operation of T_j. If such conflicting forward recovery operation is executed to forward recovery T_j, transaction T_i is aborted before T_j, to guarantee that the backward and forward recovery operations of T_i can be moved to p_i.

The class *SFBSF* is not sufficient to guarantee *SFRED*, because it does not consider serializability of the reduced schedule. To capture the notion of serializability of the reduced schedule, we introduce the class of committed forward semantic conflict serializable schedules (*CFSCSR*). The class considers semantic conflict serializability of the committed projection of the forward expanded schedule, including the forward recovery operations.

Definition 6.4.2.2 (Committed forward semantic conflict serializability (*CFSCSR*)) *A schedule $S = (T, P, <_S)$ is committed forward semantic conflict serializable, if $CF(S) \in SCSR$.*

From the following theorem it follows that *SFBSF* and *CFSCSR* are sufficient for *SFRED*.

Theorem 6.4.2.3 *$SFBSF \cap CFSCSR \subseteq SFRED$*

Proof 6.4.2.4
(By induction)
The induction is over the prefixes of some arbitrary schedule S. The induction hypothesis is that for some prefix S_k of S, we have $S_k \in SFBSF \cap CFSCSR$ and $S_k \in SFRED$, i.e. in the forward expanded schedule S'_k of S_k the compensating operations can be removed by the commute, compensate and order rules in the context of the backward semantics and subsequently the schedule can be rewritten to a serial schedule by the commute rule in the context

of the forward semantics.

(Base case)
The base case involves an empty schedule. As the forward expanded schedule is then also empty, the induction hypothesis trivially holds.

(Inductive case)
We consider a prefix S_{k-1} that satisfies the induction hypothesis that $S_{k-1} \in SFBSF \cap CFSCSR$ and $S_{k-1} \in SFRED$. We have to prove that prefix S_k also satisfies this induction hypothesis, where S_k is prefix S_{k-1} with an additional operation p_k. The induction hypothesis is satisfied, if the forward expansion S'_k of prefix S_k can be rewritten according to definition 6.3.0.7.

We distinguish the cases where operation p_k is a compensatable operation, a non-compensatable operation, an abort operation, a rollback operation and a commit operation.

Case: p_k is compensatable
In the forward expanded schedule S'_k, the operations p_k and p_k- are adjacent. By applying the compensate rule, the forward expanded schedule of S'_{k-1} is achieved. As S'_{k-1} can be reduced by assumption, such procedure also exists for S'_k.

Case: p_k is a non-compensatable operation
If $S_k \notin SFRED$, then either the uncommitted operations cannot be removed from S'_k by the commute, compensate or undo rules, or otherwise, the remaining schedule cannot be rewritten to a serial schedule by the commute rule. We consider both cases.

Case: uncommitted operations cannot be removed from S'_k
We demonstrate that all uncommitted operations can be removed from S'_k by iteratively removing the first compensating operation. Assume that this first compensating operation cannot be removed. Then, there must be a conflict path $p_i < \ldots < p_j < p_i^-$ in S'_k due to operation p_j. We demonstrate that operation p_j cannot exist.
Assume that operation p_j is an uncommitted operation. By definition of SFBSF we have that $p_j < p_i^-$ implies that $\mathrm{Rb}(p_j)$ and $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$. But then, by definition of a forward expanded schedule $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$ implies that $(p_i^-, p_j-) \notin <_S$. This means that p_j^- can be moved to p_j, which contradicts that the conflict path exists.
Assume that operation p_j is a committed operation. Then, by definition of SFBSF we have that $\mathrm{Cm}(p_i)$ and $\mathrm{Fr}(p_i) < \mathrm{Fr}(p_j)$. This contradicts that p_i^- is in S'_k.
Assume that operation p_j is a forward recovery operation. By definition of SFBSF we would have $(\mathrm{Ab}_j, \mathrm{Ab}_i) \notin <_S$ or $(\mathrm{Ab}_j, Rb_i) \notin <_S$. By definition of the forward expanded schedule, this would result for either case in $(p_j, p_i^-) \notin <_S$. This however contradicts our path.
This means, that the conflict $p_j < p_i^-$ cannot exist in S'_k and thus p_i^- can be moved to p_i by means of the commute rule. Then, the compensate rule can be used to remove p_i and p_i^-. This process can be repeated until all pairs operations are removed from S'_k.

Case: the committed and forward recovery projection of S'_k cannot be serialized
Let S''_k be the remaining schedule of S'_k after removing all pairs p_i and p_i^-. If S''_k cannot be se-

rialized by the commute and order rules, there must be some conflict path $p_i < \ldots < p_m < p_{i_n}$ of committed and forward recovery operations with $p_i, p_{i_n} \in P'_i$ and $p_m \in P'_m$ with $T_i \neq T_m$. According to the definition of the forward expanded schedule, the conflict order among forward recovery operations and committed operations is consistent with the order of the corresponding abort operations of the forward recovery operations. This means, that we can rewrite the conflict path by merging adjacent forward recovery operations of the same transaction by its corresponding abort operation, without changing the order of abort operations with respect to committed operations. The resulting conflict path would then also be reflected in the committed projection and abort operations with forward recovery semantics. This however contradicts the fact that the committed projection with abort operations is semantic conflict serializable (i.e. $S_k \in CFSCSR$).

Case: p_k is an abort operation
We follow a similar reasoning as the case with the non-compensatable operation. If $S_k \notin SFRED$, then either the uncommitted operations cannot be removed from S'_k by the commute, compensate or order rules, or otherwise, the remaining schedule cannot be rewritten to a serial schedule by the commute, compensate and order rules. We consider both cases.

Case: uncommitted operations cannot be removed from S'_k
We demonstrate that all compensating operations can be removed from S'_k from left to right in the schedule. Let p_i^- be the first compensating operation that cannot be removed from S'_k due to conflict path $p_i < \ldots < p_j < p_i^-$.
If p_j is not a forward recovery operation, then by definition of SFBSF we have $\mathrm{Rb}(p_j) \wedge (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$. This means, by definition of the forward expanded schedule that p_j^- can be ordered before p_i^- by the order rule, which contradicts the fact that p_i^- is the first compensating operation.
If p_j is a forward recovery operation, then by definition of SFBSF we have $(\mathrm{Ab}_j, \mathrm{Ab}_i) \notin <_S$. By definition of the forward expanded schedule this implies that $(p_j, p_i^-) \notin <'_S$. This contradicts the conflict path.

Case: the committed and forward recovery projection of S'_k cannot be serialized
In this case, there must be a conflict path $p_i < \ldots < p_j < p_{i_s}$ due to some forward recovery operation $p_{i_s} \in P'_i$ and committed or forward recovery operation p_j.
If p_j is a committed operation, then by definition of SFBSF we have $\mathrm{Ab}(p_j)$ and $(\mathrm{Ab}_i, \mathrm{Ab}_j) \notin <_S$. This however contradicts that p_j is committed.
If p_j is a forward recovery operation, then by definition of SFBSF we have $(\mathrm{Ab}_j, \mathrm{Ab}_i) \notin <_S$. By definition of the forward expanded schedule, this means that $(p_j, p_{i_s}) \notin <_S$. This however contradicts our conflict path.
We conclude that the new conflict path cannot exist, and thus S''_k can be serialized by the commute and order rules. This implies that $S_k \in SFRED$.

Case: p_k is a rollback operation
This case is similar to the case of an abort operation without forward recovery operations.

Case: p_k is a commit operation

A commit operation behaves in a similar way as a non-compensatable operation p_k with null semantics and $FR(p_k) = \emptyset$. Therefore, the same reasoning applies as with a non-compensatable operation.

We have demonstrated that the induction hypothesis is preserved for all possible extensions of prefix S_{k-1} to prefix S_k. Therefore, we conclude that $SFBSF \cap CFSCSR \subseteq SFRED$.

With theorem 6.4.2.3 we are ready to propose our protocol for *SFRED*. We develop the protocol in two steps. First, we develop a protocol for *SFBSF* and then we develop a protocol for *CFSCSR*. We demonstrate that the combination of both protocols guarantees *SFRED*.

For *SFBSF* we propose a graph testing protocol that includes graphs to represent the obligatory commit order $<_{\mathrm{Cm}}$ and abort order $<_{\mathrm{Ab}}$ among active transactions according to the definition of *SFBSF*. In addition, we consider orders for the case of rollbacks in the schedule. We refer to these orders as the rollback commit order $<_{\mathrm{RCm}}$ and the rollback abort order $<_{\mathrm{RAb}}$. The orders are represented by corresponding graphs.

Protocol 6.4.2.5 (SFBSF) *Consider a commit, Rcommit, abort and Rabort graph with transactions as nodes and edges between the nodes according to the orders $<_{\mathrm{Cm}}$, $<_{\mathrm{RCm}}$, $<_{\mathrm{Ab}}$ and $<_{\mathrm{RAb}}$. When operation p_j is the submitted operation, then schedule according to the following rules:*

- *Compensatable operation p_j:*
  - *include transaction T_j of p_j to the commit and abort graph, and for some non-terminated $p_i \in T_i$:*
    - *add an edge $T_i <_{\mathrm{Cm}} T_j$ in the commit graph, if $\bigvee_{p_{is} \in FR(\Pr(p_i))} SCON(R_{p_j}, Rp_{i_s})$*
    - *add an edge $T_i <_{\mathrm{RCm}} T_j$ in the Rcommit graph, if $SCON(R_{p_j}, R_{p_i^-})$*
    - *add an edge $T_i <_{\mathrm{Ab}} T_j$ in the abort graph, if $\bigvee_{p_{js} \in FR(\Pr(p_j)), p_{is} \in FR(\Pr(p_i))} SCON(R_{p_{js}}, Rp_{i_s})$*
    - *add an edge $T_i <_{\mathrm{RAb}} T_j$ in the Rabort graph, if $\bigvee_{p_{js} \in FR(\Pr(p_j)} SCON(R_{p_{js}}, R_{p_i^-}))$*
- *Non-compensatable operation p_j/Commit Cm_j:*
  - *if $T_i <_{\mathrm{Cm}} T_j$ or $T_i <_{\mathrm{RCm}} T_j$ then delay p_j or Cm_j*
  - *else remove T_j from all graphs*
- *Abort Ab_j:*
  - *group abort T_j with all T_i that satisfy $T_j <_{\mathrm{Cm}} T_i$ or $T_j <_{\mathrm{RCm}} T_i$ or $T_i <_{\mathrm{Ab}} T_j$ or $T_i <_{\mathrm{RAb}} T_j$*
  - *remove the aborted transactions from all graphs*
- *Rollback Rb_j:*

- *group rollback T_j with all T_i that satisfy $T_j <_{\mathrm{RCm}} T_i$*
- *remove the rollbacked transactions from all graphs except the abort graph*

The following theorem states that the protocol guarantees the class *SFBSF*.

Theorem 6.4.2.6 *$Gen(SFBSF) \subseteq SFBSF$*

Proof 6.4.2.7 *(By induction) We prove by induction of the prefixes of some schedule S.*

(Base case)
The base case involves an empty schedule, which trivially satisfies the induction hypothesis.

(Inductive case)
As induction hypothesis we assume that schedule prefix $S_k \in Gen(SFBSF)$ and $S_k \in SFBSF$. Consider a schedule prefix S_{k-1} that satisfies the induction hypothesis, and schedule prefix S_k with $S_k \in Gen(SFBSF)$ but $S_k \notin SFBSF$ due to the additional operation p_k.

By definition, SFBSF can only be violated if operation p_k is not a compensatable operation. We consider the cases where p_k is a non-compensatable operation, an abort operation or a rollback operation. We do not consider a commit operation, as this operation can be considered a non-compensatable operation with an empty forward recovery set. For all cases, we assume that the initial condition of SFBSF is true: $p_i <_S p_j \wedge (\mathrm{Fr}(p_i), p_j) \notin <_S$.

Case: p_k is a non-compensatable operation

Case: $p_k = \mathrm{Fr}(p_i)$

In this case, SFBSF is violated only if:

a) $(SCON(R_{p_j}, R_{p_i^-})) \vee (\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, R_{p_{i_s}})) \wedge$
$\mathrm{Cm}(p_j) \wedge (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$

b) $((\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})) \vee$
$(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s}))) \wedge$
$\mathrm{Ab}(p_j) \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$

Case a)
In this case, we must have the execution order: $p_i < p_j$, with $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$.
If $\vee_{p_{i_s} \in FR(\mathrm{Pr}(p_i))} SCON(R_{p_j}, Rp_{i_s})$ the compensatable operation rule inserts edge $T_i <_{\mathrm{Cm}} T_j$ after executing p_j. But then, the non-compensatable operation rule would delay $\mathrm{Fr}(p_j)$ until $\mathrm{Fr}(p_i)$ has executed, which contradicts $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$.
Likewise, if $SCON(R_{p_j}, R_{p_i^-})$ the compensatable operation rule inserts edge $T_i <_{\mathrm{RCm}} T_j$ after executing p_j. The non-compensatable operation rule delays $\mathrm{Fr}(p_j)$ after $\mathrm{Fr}(p_i)$, which contradicts $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$.

Case b)
In this case, we must have execution order: $p_i < p_j < \mathrm{Fr}(p_i) < \mathrm{Ab}_j$.

If $\vee_{p_{j_s} \in FR(\mathrm{Pr}(p_j))} SCON(R_{p_{j_s}}, R_{p_i^-})$ *then after operation* p_j*, an edge* $T_i <_{\mathrm{RAb}} T_j$ *is inserted. Upon abort* Ab_j*, the edge* $T_i <_{\mathrm{RAb}} T_j$ *implies that* T_i *is aborted as well. This contradicts that* $\mathrm{Fr}(p_i)$ *is a non-compensatable operation.*
If $\vee_{p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i))} SCON(R_{p_{j_s}}, Rp_{i_s})$ *then after operation* p_j*, an edge* $T_i <_{\mathrm{Ab}} T_j$ *is inserted. The edge* $T_i <_{\mathrm{Ab}} T_j$ *implies that* T_i *is aborted as well, which contradicts that* $\mathrm{Fr}(p_i)$ *is a non-compensatable operation.*

Case: $p_k = \mathrm{Fr}(p_j)$

In this case, SFBSF is violated only if:

a) $SCON(R_{p_j}, R_{p_i^-}) \wedge$
$(\mathrm{Fr}(p_i) = \mathrm{Ab}_i \vee \mathrm{Fr}(p_i) = \mathrm{Rb}_i \vee (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S)$

b) $\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s}) \wedge$
$(\mathrm{Fr}(p_i) = \mathrm{Ab}_i \vee (\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S)$

Case a)
We have the execution order $p_i < p_j$ *such that* $SCON(R_{p_j}, R_{p_i^-})$ *implies that* $T_i <_{\mathrm{RCm}} T_j$ *after operation* p_j*. If* $\mathrm{Fr}(p_i) = \mathrm{Ab}_i$ *or* $\mathrm{Fr}(p_i) = \mathrm{Rb}_i$ *then the abort and rollback rules would imply the abort or rollback of transaction* T_j *as well, which contradicts that* $\mathrm{Fr}(p_j)$ *is a non-compensatable operation. The non-compensatable operation rule implies that* $\mathrm{Fr}(p_j)$ *is delayed until* $\mathrm{Fr}(p_i)$ *has executed. Therefore,* $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$ *cannot occur either.*

Case b)
We have the execution order $p_i < p_j$ *and because* $\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s})$ *an edge* $T_i <_{\mathrm{Cm}} T_j$ *is inserted after operation* p_j*. By the non-compensatable operation rule, operation* $\mathrm{Fr}(p_j)$ *is not executed before* $\mathrm{Fr}(p_i)$*, which contradicts* $(\mathrm{Fr}(p_i), \mathrm{Fr}(p_j)) \notin <_S$*. Also, if* $\mathrm{Fr}(p_i) = \mathrm{Ab}_i$ *then the abort rule would imply the abort of* T_j*, which contradicts that* $\mathrm{Fr}(p_j)$ *is a non-compensatable operation.*

From the discussion above, we can conclude that the protocol guarantees SFBSF if p_k *is a non-compensatable operation.*

Case: p_k is an abort operation

Case: $p_k = \mathrm{Fr}(p_i)$

In this case, SFBSF is violated only if:

a) $(SCON(R_{p_j}, R_{p_i^-})) \vee (\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s}))) \wedge$
$\mathrm{Fr}(p_j) \neq \mathrm{Ab}_j \wedge \mathrm{Fr}(p_j) \neq \mathrm{Rb}_j$

b) $(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})) \vee$
$(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s})) \wedge$
$\mathrm{Fr}(p_j) = \mathrm{Ab}_j \wedge \mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$

Case a)
If $SCON(R_{p_j}, R_{p_i^-})$ *then an edge* $T_i <_{\mathrm{RCm}} T_j$ *was inserted after operation* p_j*. The non-compensatable operation rule would delay* $\mathrm{Fr}(p_j)$ *until* $\mathrm{Fr}(p_i)$ *executed. Then, when* Ab_i *executes, by the abort rule,* T_j *would also be aborted. This contradicts that* $\mathrm{Fr}(p_j) \neq \mathrm{Ab}_j$*. Likewise, if* $\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s})$ *then an edge* $T_i <_{\mathrm{Cm}} T_j$ *was inserted after operation* p_j*. Again, the non-compensatable operation rule would delay* $\mathrm{Fr}(p_j)$ *until* $\mathrm{Fr}(p_i)$ *executed, and* T_j *would be aborted when* Ab_i *executes.*

Case b)
If $\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})$ *then an edge* $T_i <_{\mathrm{RAb}} T_j$ *was inserted after operation* p_j*. By the abort rule, the abort of transaction* T_j *would then result in a group abort together with transaction* T_i*. This contradicts* $\mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$*. Likewise, if* $(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s}))$ *then an edge* $T_i <_{\mathrm{Ab}} T_j$ *was inserted after operation* p_j*. Again, by the abort rule, the abort of transaction* T_j *results in a group abort with* T_i*, which contradicts* $\mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$*.*

Case: p_k=$\mathrm{Fr}(p_j)$

In this case, SFBSF is violated only if:

a) $SCON(R_{p_j}, R_{p_i^-}) \wedge (\mathrm{Fr}(p_i) = \mathrm{Ab}_i \vee \mathrm{Fr}(p_i) = \mathrm{Rb}_i) \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$

b) $\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s}) \wedge \mathrm{Fr}(p_i) = \mathrm{Ab}_i \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$

c) $((\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})) \vee$
$(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s}))) \wedge$
$\mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$

Case a)
If $SCON(R_{p_j}, R_{p_i^-})$ *then an edge* $T_i <_{\mathrm{RCm}} T_j$ *was inserted after operation* p_j*. Then, by the abort rule, the abort of* T_i *implies a group abort with* T_j*. This contradicts* $\mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$*. In a similar way, the rollback rule also implies a group abort with* T_i*, which contradicts* $\mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$*.*

Case b)
The same reasoning can be applied as for Case a). An edge $T_i <_{\mathrm{Cm}} T_j$ *is inserted after operation* p_j*. Then, by the abort rule the abort of* T_i *implies a group abort with* T_j*, which contradicts* $\mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$*.*

Case c)
If $\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})$ *then an edge* $T_i <_{\mathrm{RAb}} T_j$ *was inserted after operation* p_j*. By the abort rule, the abort of* T_j *implies a group abort with* T_i*. This however contradicts* $\mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$*.*
Likewise, because $\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s}))$*, an edge* $T_i <_{\mathrm{Ab}} T_j$ *was inserted after operation* p_j*. By the abort rule,* T_j *implies a group abort with* T_i*, which contradicts* $\mathrm{Fr}(p_j) <_S \mathrm{Fr}(p_i)$*.*

Case: p_k is a rollback operation

Case: $p_k = \mathrm{Fr}(p_i)$

In this case, SFBSF is violated only if:

a) $SCON(R_{p_j}, R_{p_i^-}) \wedge$
$\mathrm{Fr}(p_j) \neq \mathrm{Ab}_j \wedge \mathrm{Fr}(p_j) \neq \mathrm{Rb}_j$

b) $(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)))SCON(R_{p_{j_s}}, R_{p_i^-})) \vee$
$(\vee(p_{j_s} \in FR(\mathrm{Pr}(p_j)), p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_{j_s}}, Rp_{i_s})) \wedge \mathrm{Fr}(p_j) = \mathrm{Ab}_j \wedge \mathrm{Fr}(p_j) <_S$
$\mathrm{Fr}(p_i)$

Case a)
If $SCON(R_{p_j}, R_{p_i^-})$ *then an edge* $T_i <_{\mathrm{RCm}} T_j$ *was inserted after operation* p_j*. The non-compensatable operation rule would delay* $\mathrm{Fr}(p_j)$ *until* $\mathrm{Fr}(p_i)$ *executed. Then, when* Rb_i *executes, by the abort rule,* T_j *would also be aborted. This contradicts that* $\mathrm{Fr}(p_j) \neq \mathrm{Ab}_j$*.*

Case b)
The reasoning is the same as for Case b) with $p_k = \mathrm{Ab}_i$*.*

Case: $p_k = \mathrm{Fr}(p_j)$

In this case, SFBSF is violated only if:

a) $SCON(R_{p_j}, R_{p_i^-}) \wedge (\mathrm{Fr}(p_i) = \mathrm{Ab}_i \vee \mathrm{Fr}(p_i) = \mathrm{Rb}_i) \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$

b) $\vee(p_{i_s} \in FR(\mathrm{Pr}(p_i)))SCON(R_{p_j}, Rp_{i_s}) \wedge \mathrm{Fr}(p_i) = \mathrm{Ab}_i \wedge \mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$

Case a)
The reasoning is the same as for Case a) with $p_k = \mathrm{Ab}_i$*.*

Case b)
The same reasoning can be applied as for Case a). An edge $T_i <_{\mathrm{Cm}} T_j$ *is inserted after operation* p_j*. Then, by the abort rule the abort of* T_i *implies a group abort with* T_j*, which contradicts* $\mathrm{Fr}(p_i) <_S \mathrm{Fr}(p_j)$*.*

From the discussion above, we can conclude that the protocol guarantees SFBSF if p_k *is a rollback operation.*

At this point we recall that only non-compensatable, abort or rollback operations can violate SFBSF (where commit operations are considered non-compensatable operations with empty forward recovery set). As the protocol guarantees SFBSF in all possible problematic cases, we can conclude that the protocol guarantees SFBSF.

In addition to the protocol for *SFBSF*, we also need a protocol for *CFSCSR*. We start with two-phase locking, because we know that this guarantees conflict serializability. Two-phase locking is applied to all operations, including uncommitted operations. This means, that the protocol is stronger than necessary for *CFSCSR* alone. This is however not a problem, because we will see that our final protocol requires two-phase locking anyway.

To guarantee semantic conflict serializability with respect to forward recovery operations, we include an additional abort rule. The abort rule aborts any transaction T_i that hold a lock on a forward recovery operation that is executed due to the abort of transaction T_j. This way, an aborting transaction can always acquire the required locks for forward recovery. In addition, we add a rule for rollbacks of transactions. This rule is added to guarantee two-phase locking with respect to the committed operations, even in the presence of rollbacks.

The protocol is defined as follows.

Protocol 6.4.2.8 (*CFSCSR***)** *Let R_T be the semantics assigned to transaction T and let R_p be the semantics for each operation p.*

- Lock acquisition rule: *before operation p is executed, a lock R_p is acquired on all accessed objects.*
- Lock conflict rule: *a lock R_p is granted only if no lock R_q is held by a different transaction with $SCON(R_q, R_p)$.*
- Lock release rule: *after a lock R_p is released, no lock R_q of the same transaction is acquired. All locks are released as soon as possible.*
- Abort rule: *upon abort T_j, group abort all transactions that hold a lock R_p with $\vee_{r \in FR(q)} SCON(R_p, R_r)$ and q is the last non-compensatable operation of T_j.*
- Rollback rule: *upon rollback T_j, release all locks that have been acquired after the last non-compensatable operation. The lock release rule is considered only with respect to locks that have been acquired before the last non-compensatable operation of T_j.*

The first three lock rules of the *CFSCSR* protocol represent conventional two-phase locking. The abort rule is the additional rule to guarantee that each aborted transaction can acquire forward recovery locks. The rollback rule releases all locks that have been acquired after the last non-compensatable operation, because these operations are backward recovered. To guarantee two-phase locking after rollback, the lock release rule is interpreted with respect to the locks that have been acquired before the last non-compensatable operation. This implies that once one of these locks has been released during execution, no new locks can be acquired by the transaction.

We note that the protocol is *not* sufficient for *CFSCSR*. This is due to the fact that a transaction is allowed to release locks before it aborts. This way, the locks of the transaction including the forward recovery locks may not be acquired two-phase, such that serializability with respect to committed and forward recovery operations is guaranteed.

This issue is solved by combining the *CFSCSR* protocol with the *SFBSF* protocol. This is achieved by the *SFBSF* protocol, by guaranteeing that any operations that are interleaved between committed and forward recovery operations are uncommitted.

Theorem 6.4.2.9 *$Gen(SFBSF) \cap Gen(CFSCSR) \subseteq CFSCSR$*

Proof 6.4.2.10 *A schedule is not CFSCSR if the committed and forward recovery projection of a schedule is not serializable. In such case, there is a transaction T_i with conflict path $p_i < p_m < \ldots < p_j < p_{i_2}$, with $p_i, p_{i_2} \in P_i$, p_i is a committed operation and p_m, p_j and p_{i_2} are either committed or forward recovery operations. By Theorem 6.4.2.6 we know that the combinations with p_{i_2} a forward recovery operation do not occur if $S \in Gen(SFBSF)$. This leaves us to prove that the case where p_{i_2} is a committed operation.*

First, we consider the case where p_m is a forward recovery operation. If T_i held the lock for p_i at the time that T_m aborted, the abort rule would have induced the abort of T_i, which contradicts the execution of p_{i_2}. Therefore, we conclude that p_m must be a committed operation.

We proceed by proving that the conflict path cannot exist if p_j is a committed operation, or a forward recovery operation.

Case: p_j is a committed operation.

Because $p_i < p_m$, T_i must have released a lock before operation p_j. Also, because $p_j < p_{i_2}$, T_i must have acquired a lock after p_j. This contradicts the two-phase locking regime of the protocol CFSCSR.

Case: p_j is a forward recovery operation.

Because $p_i < p_m$, T_i must have acquired all locks before T_j aborts. This means that lock p_{i_2} was held by T_i at the time T_j aborts. By the abort rule, T_i would have been aborted, which contradicts that operation p_{i_2} committed.

This leaves us to prove that the assumption of two-phase locking with respect to committed operations in the presence of rollbacks is justified.

Reconsider the conflict path $p_i < p_m < \ldots < p_j < p_{i_2}$. This time, we consider the case where T_i is rollbacked during execution. We consider all cases where the rollback could occur in the conflict path and demonstrate that two-phase locking with respect to the committed operations is always guaranteed.

Case $\mathrm{Rb}_i < p_i < p_m \ldots < p_j < p_{i_2}$

If some lock was released before Rb_i*, then by the rollback rule, the lock for p_{i_2} must have been acquired before* Rb_i*. This however contradicts that p_{i_2} is acquired after p_j. If no lock was released before* Rb_i*, then, because p_i releases its lock, lock p_{i_2} must have been acquired before p_i. This however contradicts that p_{i_2} is acquired after p_i.*

Case $p_i < \mathrm{Rb}_i < p_m < \ldots < p_j < p_{i_2}$

If some lock was released before Rb_i*, then by the rollback rule, lock p_{i_2} was acquired before* Rb_i*. This contradicts $p_j < p_{i_2}$. If no lock was released before* Rb_i*, then all locks are*

still held after rollback. Because $p_i < p_m$, lock p_{i_2} was held before p_m, which contradicts that lock p_{i_2} is acquired after p_j.

Case $p_i < p_m < \mathrm{Rb}_i < p_j < p_{i_2}$

Lock p_i was released before Rb_i, such that lock p_{i_2} must have been acquired before Rb_i. This however contradicts that p_{i_2} is acquired after p_j.

Case $p_i < p_m < \ldots < p_j < \mathrm{Rb}_i < p_{i_2}$

Lock p_i was released before Rb_i, such that the lock p_{i_2} must have been acquired before p_i. This contradicts that the lock p_{i_2} is acquired after p_j.

Case $p_i < p_m < \ldots < p_j < p_{i_2} < \mathrm{Rb}_i$

As p_{i_2} is committed, the schema satisfies two-phase locking and thus the path cannot exist.

Hence, we have demonstrated that no conflict path can occur between committed and forward recovery operations of a transaction, such that CFSCSR is guaranteed.

By theorem 6.4.2.6 and theorem 6.4.2.9, the protocols *SFBSF* and *CFSCSR* are sufficient for *SFBSF*$\cap$*CFSCSR*. From theorem 6.4.2.3 it follows that $Gen(SFBSF) \cap Gen(CFSCSR) \subseteq SFRED$. In other words, the protocols *SFBSF* and *CFSCSR* are sufficient to guarantee *SFRED*.

6.5 Semantic forward reducibility for the tree model

The results of the previous sections can be applied to the tree model by combining the definitions for expanded transactions and schedules with the definitions of transaction trees and ordered tree schedules. For each non-leaf operation, we consider the expansion with respect to its children.

First, we define a transaction tree with extended recovery semantics as follows.[4]

Definition 6.5.0.11 (Transaction tree with extended recovery semantics) *A transaction tree with extended recovery semantics is a tuple $T = (P, <_a, <_i)$ with:*

- *P a set of data operations and transactional operations,*
- *$<_a$ a hierarchical order on P, called the abstraction order,*
- *$<_i$ a partial order on P, called the intra-transaction order,*
- $\forall k \in P : \mathrm{Ch}(k) \neq \emptyset \Rightarrow$
  - $\mathrm{Ch}(k)$ *is a set of compensatable and non-compensatable data operations, durability operations, rollback operations and one termination operation* Tm_k*, with* $\mathrm{Tm}_k \in \{\mathrm{Ab}_k, \mathrm{Cm}_k\}$

[4] Recall the definition of $\mathrm{Ch}(k)$ in section 2.4.

  - $\bigwedge_{p \in \mathrm{Ch}(k), q \in \mathrm{Ch}(k) \setminus Cmp(\mathrm{Ch}(k))} (p <_i q \vee q <_i p)$, *and*
  - $\bigwedge_{p \in \mathrm{Ch}(k) \setminus \{\mathrm{Tm}_k\}} (p <_i \mathrm{Tm}_k)$.
- $\forall p_i, p_j, p'_i, p'_j \in P : p_i <_i p_j \wedge p_i <_a p'_i \wedge p_j <_a p'_j \Rightarrow p'_i <_i p'_j$

Essentially, the transaction tree with extended recovery semantics is similar to a transaction tree, except that the new recovery operations are distinguished among the children of each non-leaf operation p_k.

The forward expanded transaction tree can be defined in a similar way as definition 6.2.2.1 of the forward expanded transaction, except that forward expansion is now defined in the context of parent opeations and their children.

Definition 6.5.0.12 (Forward expanded transaction tree) *Let $T = (P, <_a, <_i)$ be a transaction tree. The forward expansion of transaction tree T is a transaction tree $FX(T) = (P', <'_a, <'_i)$ with:*

- $P' = \bigcup_{k \in P, p \in \mathrm{Ch}(k)} \mathrm{expand}_{(\mathrm{Ch}(k), <_i)}(p)$
- $<'_a = \{(r, s) \mid k, l \in P \wedge p \in \mathrm{Ch}(k) \wedge q \in \mathrm{Ch}(l) \wedge p <_a q \wedge r \in \mathrm{expand}_{(\mathrm{Ch}(k), <_i)}(p) \wedge s \in \mathrm{expand}_{(\mathrm{Ch}(l), <_i)}(q)\}$
- $<'_i = \{(r, s) \mid k, l \in P \wedge p \in \mathrm{Ch}(k) \wedge q \in \mathrm{Ch}(l) \wedge p <_i q \wedge r \in \mathrm{expand}_{(\mathrm{Ch}(k), <_i)}(p) \wedge s \in \mathrm{expand}_{(\mathrm{Ch}(l), <_i)}(q)\}$.

In a similar way, we define the *ordered tree forward expanded schedule* as follows.

Definition 6.5.0.13 (Forward expanded ordered tree schedule) *Let $S = (T, P, <_m, <_d)$ be an ordered tree schedule. The forward expansion of S is an ordered tree schedule $FX(S) = (T', P', <'_m, <'_d)$ with:*

- $T' = \bigcup_{T'' \in T} FX(T'')$
- $P' = \bigcup_{(P'', <'') \in T'} P''$
- $<'_m = \{(r, s) \mid (P_p, <_{ap}, <_{ip}), (P_q, <_{aq}, <_{iq}) \in T \wedge k \in P_p \wedge l \in P_q \wedge p \in \mathrm{Ch}(k) \wedge q \in \mathrm{Ch}(l) \wedge p <_m q \wedge r \in \mathrm{expand}_{(\mathrm{Ch}(k), <_{ip})}(p) \wedge s \in \mathrm{expand}_{(\mathrm{Ch}(l), <_{iq})}(q)\}$
- $<'_d = \{(r, s) \mid (P_p, <_{ap}, <_{ip}), (P_q, <_{aq}, <_{iq}) \in T \wedge k \in P_p \wedge l \in P_q \wedge p \in \mathrm{Ch}(k) \wedge q \in \mathrm{Ch}(l) \wedge p <_d q \wedge r \in \mathrm{expand}_{(\mathrm{Ch}(k), <_{ip})}(p) \wedge s \in \mathrm{expand}_{(\mathrm{Ch}(l), <_{iq})}(q)\}$

The forward expanded ordered tree schedule is defined as a schedule of forward expanded transaction trees, where the composition orders and dynamic orders are extended over the newly introduced operations according to the orders on their originating operations.

The class of ordered semantic tree forward reducible schedules (*OSTFRED*) can then be defined by combining the definition for *OSTCSR* (definition 4.3.0.10) and *SFRED* (definition 6.3.0.7).

Definition 6.5.0.14 (Ordered semantic tree forward reducible (*OSTFRED*)) *An ordered tree schedule $S = (T, P, <_m, <_d)$ is ordered semantic tree forward reducible if the semantics of the committed and forward recovery operations of all transaction trees are sufficient and schedule S can be transformed into a serial tree schedule by the following rules:*

- commute rule: *for an adjacent pair of ordered leaf operations $p <_d q$ reverse the order, if $\neg SCON(R_p, R_q)$ and not $p <'_i q$ for some $(P', <'_a, <'_i) \in T$.*
- compose rule: *for all operations k, remove the operations* $\mathrm{Ch}(k)$*, if* $\mathrm{Ch}(k)$ *are leaf operations according to $<_m$ and are non-interleaved by other leaf operations,*
- compensate rule: *remove any adjacent ordered pair of leaf operations $p <_d p^-$, where p^- is the compensating operation of p.*
- order rule: *an unordered pair of leaf operations p and q is ordered either $p <_d q$ or $q <_d p$.*

The definition of *OSTFRED* is similar to *OSTCSR*, except for the fact that the forward ordered tree expanded schedule is considered instead of the conventional tree expanded schedule and reducibility is required by means of the compensate rule.

We develop the concurrency control for *OSTFRED* such that each level of the tree schedule is scheduled according to the concurrency control developed in section 6.4.2. To guarantee consistency of the conflict order among the levels, we adopt the strategy of including the parent lock in the two-phase regime of its children, along the lines of locking protocol *OSTCSR* in section 4.3.1. We develop the concurrency control for *OSTFRED* such that *SFBSF* and *CFSCSR* are guaranteed for each level of the ordered tree schedule and additionally guarantee consistency of the conflict orders among the different levels. Both criteria are sufficient for *OSTFRED*. This is expressed by theorem 6.5.0.15 .

Theorem 6.5.0.15 (Level-by-level *SFBSF* and *CFSCSR*) *Let S be a semantic ordered tree schedule. We have $S \in OSTFRED$ if for each level $S_k : S_k \in SFBSF \cap CFSCSR$ and the conflict order on the committed and forward recovery operations in the forward expanded schedule is consistent with the conflict order at level $k+1$.*

Proof 6.5.0.16 *(By induction)*
We prove by induction over the levels of tree schedule S.

(Base case)
At level S_0 we have by theorem 6.4.2.3 that $S_0 \in SFBSF \cap CFSCSR \Rightarrow S_0 \in OSTFRED$.

(Induction case)
Let schedule S_k be the schedule at level k and let S_k be the first level-by-level schedule of the ordered semantic tree forward expanded schedule of S that cannot be reduced and serialized. Because $S_k \in SFBSF \cap CFSCSR$ we have by theorem 6.4.2.3 that $S_k \in SFRED$. This means that at each level a procedure exists for the forward expanded schedule to remove uncommitted operations and serialize the committed operations. The fact that $S_k \notin OSTFRED$ however indicates that this is not true if S_k was composed from level S_k+1. This means, that through the procedure OSTSPRED a conflict path $p_i < p_j < \ldots < p_m < p'_i$ arises at some point

in the rewriting procedure where p_i and p'_i should be moved together if S_k is composed from $S_k + 1$, while the conflict path does not arise when S_k is considered separately. Apparently, the procedure of OSTFRED must contradict some conflict order that was assumed by the procedure of SFRED at level S_k.
Let these two operations be p_i and p_j, with $p_i < p_j$ in S_k and $p_j < p_i$ the order that results from the procedure of OSTFRED after composing level $S_k + 1$. In the latter case, operations p_j and p_i were composed by the compose rule, such that the order of the children of operations p_j and p_i must have been $p_{j_1} < \ldots < p_{j_n} < p_{i_1} < \ldots < p_{j_m}$ in the level S_{k+1}. Operations p_i and p_j must conflict, because otherwise the commute rule could have been applied to reverse the order. Similarly, some of the children of operations p_i and p_j must conflict, because otherwise the commute rule could reverse the order of all children of operations p_i and p_j in level S_{k+1}. The operations p_i and p_j cannot be an uncommitted operation or compensatable operation, because $S_{k+1} \in SFRED$ and thus these operations are removed from S_{k+1} before the compose rule is applied. This leaves us committed and forward recovery operations. But then the conflict order of the children of p_i and p_j is inconsistent with the conflict order of its parents, which contradicts the assumption that this order be consistent.
The fact that $S_k \notin OSTFRED$ can therefore not exist and thus $S \in OSTFRED$.

The implication of theorem 6.5.0.15 is that for preserving *OSTFRED* we can re-use the protocols for *SFBSF* and *CFSCSR* for each level of the schedule, as long as we guarantee the consistency of the conflict orders on committed operations and forward recovery operations. This conflict order can be guaranteed to be consistent in a similar way as protocol $OST2PL$. For this, we extend the protocol *CFSCSR* according to the same principle of including the parent operation in two-phase locking. In addition to this, we apply the protocols *SFBSF* and *CFSCSR* level by level.

Protocol 6.5.0.17 (*OSTCFSCSR*)
Let R_p be the semantics for each operation p.

- Lock acquisition rule: *before operation p is executed, a lock R_p is acquired on all accessed objects.*
- Lock conflict rule: *a lock R_p is granted only if no lock R_q is held by a different parent operation than the parent of p with $SCON(R_q, R_p)$ such that operations p and q share a level.*
- Lock release rule: *after a lock R_p is released, no lock R_q is acquired if p and q have the same parent or either is the parent of the other. All locks are released as soon as possible.*
- Abort rule: *upon abort of operation l, group abort all operations k that share a level with l and hold a lock R_p with $\vee_{r \in FR(q)} SCON(R_p, R_r)$ and q is the last non-compensatable operation of operation l.*
- Rollback rule: *upon rollback of operation l, release all locks that have been acquired after the last non-compensatable operation. The lock release rule is considered only with respect to locks that have been acquired before the last non-compensatable operation of l.*

The first extension of protocol *OSTCFSCSR* with respect to protocol *CFSCSR* is the replacement of all references to transactions by references to non-leaf operations. Additionally, the lock conflict rule is bound to the levels in which the operations appear. To guarantee the consistency of the conflict orders over the schedule levels, the lock release rule is extended such that the lock for the parent operation is included in the growing phase. This is similar to the *OST*2*PL* protocol (see definition 4.3.1.1).

6.6 SFRED for semantically decomposed transactions

The results of the previous sections can also be applied to the case where transactions are semantically decomposed. The particular property of semantically decomposed transactions is that operations may have different relevant semantics for each unit of consistency of the same transaction. This was the motivation to define correctness class *DSCSR* in terms of semantic conflict serializability with respect to each unit of consistency.

The question now arises how failure semantics of operations should be defined in the context of semantically decomposed transactions. We follow the approach of the previous sections by considering each operation an atomic unit with respect to recovery. This means, that the decomposed semantics of an operation either all appear in the database or none. This is achieved by considering the compensation sufficient semantics for each uncommitted operation and its compensating operation (see definition 6.2.3.3).

Given this assumption, we define the class of *decomposed forward reducible schedules* (*DSFRED*). The problem however is that *DSCSR* was not defined in terms of a rewriting procedure, such as the alternative definition 2.2.0.18 for *CSR*. Therefore, we introduce a definition for *DSFRED*, where the requirement for *DSCSR* for the reduced schedule is stated explicitly in the definition. We define the class as follows.

Definition 6.6.0.18 (Decomposed semantic forward reducible schedule (*DSFRED*)) *For each operation p, let R_p be the compensation sufficient semantics of p. A schedule $S = (T, P, <)$ is decomposed semantic forward reducible if its forward expanded schedule $FX(S) = (T', P', <')$ can be rewritten to a decomposed semantic conflict serializable schedule of transactions with the following rules:*

- commute rule: *for an adjacent ordered pair of operations $p < q$ reverse the order, if $\neg SCON(R_p, R_q)$ and not $p <_k q$ for some $(P_k, <_k) \in T$.*
- compensate rule: *remove any adjacent pair $p < p^-$, where p^- is the compensating operation of p.*
- order rule: *an unordered pair p and q is ordered either $p < q$ or $q < p$.*

The definition is a straightforward interpretation of the definition of *SFRED*. However, two aspects are worth noting here. First, the reduced schedule is required to be *decomposed semantic conflict serializable*, instead of a serial schedule. Second, the semantics under consideration of the commute rule are the compensation sufficient semantics (see definition 6.2.3.3) and not the sufficient decomposed semantics in the context of *DSCSR*.

For the concurrency control of *DSFRED*, this means that we can leave *SFBSF* and its protocol unmodified to guarantee reducibility of a schedule. For preserving *DSCSR* for

the reduced schedule, we do have to introduce a different class than *CFSCSR*: the class of decomposed committed and forward semantic conflict serializable schedules (*CFDSCSR*). *CFDSCSR* is the semantic unit of consistency interpretation of *CFSCSR*.

Definition 6.6.0.19 (*CFDSCSR*) *A schedule $S = (T, P, <)$ is committed forward decomposed semantic conflict serializable, if $CF(FX(S)) \in DSCSR$.*

The protocol for *CFDSCSR* can be derived by combining the protocol *CFSCSR* and *D2PL* (see protocol 5.4.2.1). We define protocol *CFD2PL* as follows.

Protocol 6.6.0.20 (*CFD2PL*) *Let $(R_T, <_T)$ be the decomposed semantics for transaction T and let $(R_p, <_p)$ be the decomposed semantics for each operation p, where R_{p_i} represents the semantics of operation p in the context of transition $R_{T_i} \in R_T$.*

- Lock acquisition rule: *before operation p is executed, a lock R_{p_i} is acquired on all objects accessed by p for all $R_{p_i} \in R_p$.*
- Lock conflict rule:*a lock R_{p_i} is granted only if no lock R_{q_j} is held by a different transaction with $SCON(R_{q_j}, R_{p_i})$ on the same object.*
- Lock release rule: *after a lock R_{p_i} is released, no lock R_{q_i} is acquired. All locks are released as soon as possible.*
- Abort rule: *upon abort T_j, group abort all transactions that hold a lock R_{p_i} with $\vee(r \in FR(q), R_{T_s} \in R_T)SCON(R_{p_i}, R_{r_s})$ and q is the last non-compensatable operation of T_j.*
- Rollback rule: *upon rollback T_j, release all locks that have been acquired after the last non-compensatable operation. The lock release rule is considered only with respect to locks that have been acquired before the last non-compensatable operation of T_j.*

The first three rules implement the *D2PL* protocol. The abort and rollback rules are taken from the protocol *CFSCSR*. Note that in protocol *SBSF* we need to consider the compensation sufficient semantics of the operations.

6.7 Conclusions

Backward recovery is the conventional recovery technique for database applications. However, backward recovery has important drawbacks. One obvious drawback is that upon backward recovery a considerable amount of work may be lost. Another drawback, is that backward recovery cannot be applied to applications with non-compensatable operations. To overcome these drawbacks, forward recovery was proposed in the literature as an alternative recovery technique.

Forward recovery guarantees atomicity by completing a transaction, even in case of failure. To guarantee that a transaction can complete successfully, we introduced a forward recovery set for each non-compensatable operation that contains the operations that are sufficient for the forward recovery.

To enhance the flexibility of recovery, we extended the execution model with durability operations and rollback operations. Durability operations are operations that guarantee persistence of intermediate results of a transaction. Rollback operations imply backward recovery up to the last non-compensatable or durability operation. This operation enhances execution flexibility considerably, because after a rollback, control can be returned to the application to complete the transaction in arbitrary ways.

To formalize these ideas, we introduced the definitions of forward expanded transactions and forward expanded schedules as extensions to the unifying theory of concurrency and recovery in the context of our extended execution model. In addition, we defined sufficient semantics of operations in the context of forward recovery.

As a correctness criterion, we introduced the class of semantic forward reducible schedules (*SFRED*). We developed a concurrency control based on graph testing, and a hybrid protocol based on graph testing and locking. We proved both protocols correct.

In the remainder of the chapter, we defined *SFRED* in the context of the tree model and in the context of semantic decomposition. With this, we demonstrated the applicability of the approach in these contexts as well. For the tree model as well for the semantic decomposition case we sketched possible concurrency controls.

Chapter 7

Case study

In this chapter we demonstrate how our developed theory can be applied to a realistic application to improve concurrency and recovery. For the case, we consider the web-shop application proposed for the TPC -W benchmark [24]. We choose the TPC-W benchmark, because this is a benchmark proposed by the Transaction Processing Council to evaluate the performance of database systems for novel applications. As such, the web-shop application can be considered a blueprint for novel applications with high demands for transaction processing. The chapter discusses how the developed theory can be applied to the application and we demonstrate that our theory results in improved concurrency and recovery with respect to conventional transaction theory.

7.1 TPC-W benchmark

The TPC-W benchmark involves a transactional web application where users can order books through a web interface. The application's data is stored in eight relational tables, including three tables that store the customer's data, a relation *item* that stores the book information and the stock level of each book, and a relation *author* that stores the author information of the books. The relations *order* and *orderline* store the order information. Finally, there is a relation that contains credit card transactions.

The TPC-W application is described by a web page interaction diagram as given in Figure 7.1. Each circle represents a web page that is presented to the user. From each web page, a user can navigate to another page either by pressing a button or a link on the page.

The *Home* page is the initial state. From this page, a user can request the list of new products (*New Product*), the list of bestsellers (*BestSeller*), view its current shopping cart (*ShoppingCart*), or invoke the search page (*SearchRequest*). All three pages result in a list of books. The user can click on the link of one book to see the product's details (*ProductDetail*). Either from this web page or the previous three pages, a book can be added to the shopping cart (*ShoppingCart*). From this page, the user can repeat the addition of books, or start the order sequence *CustomerRegistration, BuyRequest, BuyConfirm*. In addition to these web pages, the *AdminRequest* and *AdminConfirm pages* can be used to update the information about a book, and the *OrderInquiry* and *OrderDisplay* pages can be used to get the current

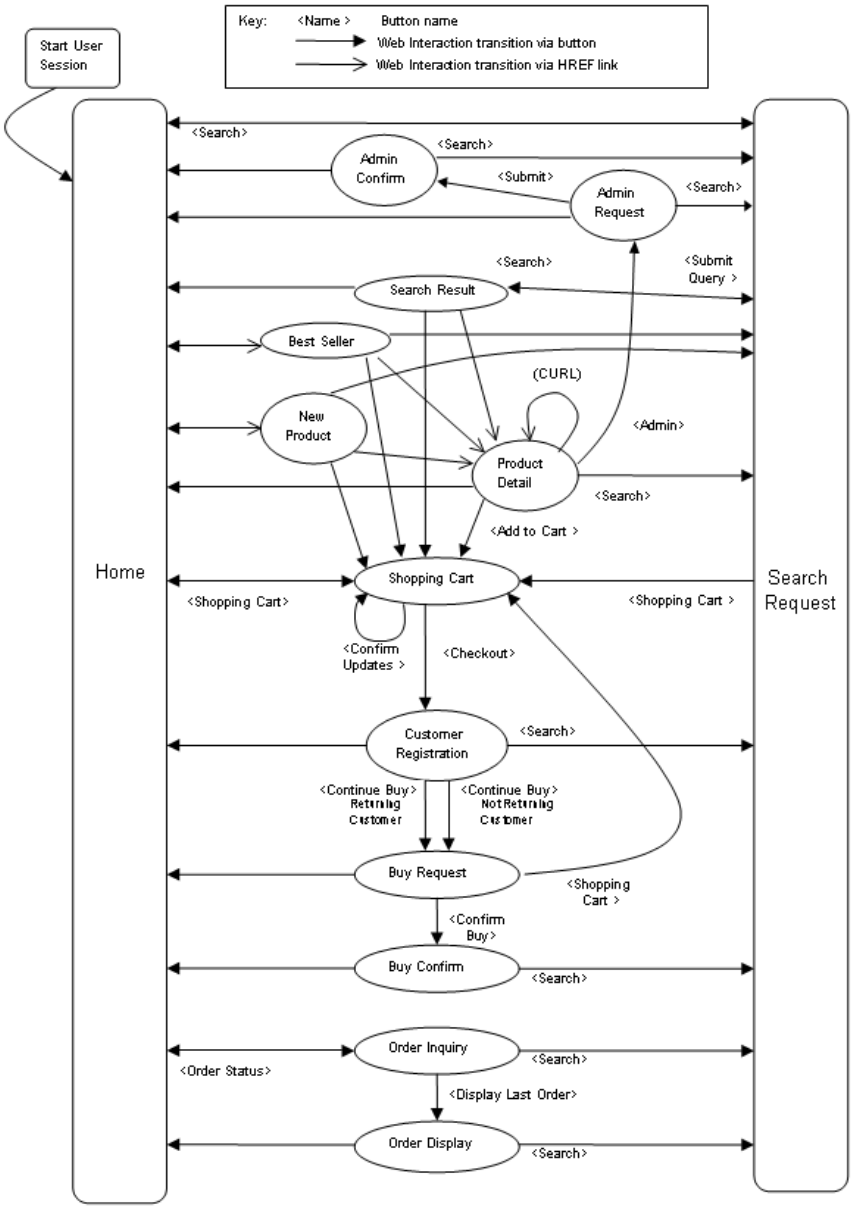

Figure 7.1: TPC-W application's web interactions [24]

order status.

7.2 Case study description

In this case study we consider a representative subset of the possible transactions and a subset of the data to demonstrate the strength of our theory.

For the data, we consider only the relations *item*, *author*, *order* and *orderline*. In addition, we consider the shopping cart as a database relation, as opposed to the TPC-W specification, where the shopping cart relation is a global program variable. In addition, we distinguish three relations to generate unique keys: *sessionid*, *orderid* and *orderlineid*. For the sake of the example, we consider only a subset of the attributes of the relations, with:

author(A_ID,A_NAME)
item(I_ID,I_A_ID,I_TITLE,I_COST,I_STOCK)
order(O_ID,O_TOTAL)
orderline(OL_ID,OL_O_ID,OL_I_ID,OL_QTY,OL_REMAIN)
cart(C_SESSION_ID,C_I_ID,C_COST,C_QTY)
sessionid(S_ID)
orderid(O_ID)
orderlineid(OL_ID)

The *author* relation stores primary key A_ID and the name of the author in A_NAME. Relation *item* stores books with primary key I_ID, a foreign key I_A_ID to A_ID, a title I_TITLE, the price of the book I_COST and the current stock level I_STOCK of the book. Relation *order* stores all orders with primary key O_ID and the total cost of the order O_TOTAL. Relation *orderline* contains the details of each order, with OL_ID the primary key, OL_O_ID a foreign key to the order, OL_I_ID a foreign key to the book ordered, OL_QTY the number of books delivered and OL_REMAIN the number of books that remain to be delivered if the stock level was insufficient. Relation *cart* represents the shopping cart, with C_SESSION_ID the key of the current user session, C_I_ID a foreign key to an item in the cart, C_COST the price of the item and C_QTY the number of items. The relations *sessionid*, *orderid* and *orderlineid* contain the current session number, order number and orderline number. In addition to the database relations, we consider program variables. We denote the name of program variables with a preceding colon.

For the transactions, we consider three traces through the web interaction diagram of figure 7.1 . We consider an *order transaction*, an *update transaction* and a *status transaction*. For each transaction we describe the operations with their implementation below.

7.2.1 Order transaction

The *order* transaction is the transaction where a user orders a set of books. We assume that the user iteratively accesses the *Home*, *SearchRequest*, *SearchResult* and *ShoppingCart* pages to add books to his shopping cart. After the last iteration, the order is placed through the *CustomerRegistration*, *BuyRequest* and *BuyConfirm pages*. The program logic between the web page transitions is captured by the transaction's operations given in table 7.1.

From page	To page	Operation
StartUserSession	Home	start
Home	SearchRequest	-
SearchRequest	SearchResult	search
SearchResult	ShoppingCart	add
ShoppingCart	CustomerRegistration	-
CustomerRegistration	BuyRequest	-
BuyRequest	BuyConfirm	order

Table 7.1: Operations of transaction *order*

From page	To page	Operation
StartUserSession	Home	start
Home	SearchRequest	-
SearchRequest	SearchResult	search
SearchResult	ProductDetail	detail
ProductDetail	AdminRequest	update
AdminRequest	AdminConfirm	-

Table 7.2: Operations of transaction *update*

We did not include an operation for the transition between the *Home* - *SearchRequest*, *ShoppingCart* - *CustomerRegistration*, and *CustomerRegistration* - *BuyRequest* because these transitions do not change data within our scope. The implementation of each operation is given in figure 7.2.

7.2.2 Update transaction

The update transaction searches for a book, displays the book's details and then updates its cost. For this, the user accesses the *Home*, *SearchRequest*, *SearchResult*, *ProductDetail*, *AdminRequest* and *AdminConfirm* pages in this order. For each page transition, we assume the operations as given in table 7.2.

For the update transaction, we consider the same operations *start* and *search* to identify the product to be updated. The new operations include detail and update, that retrieve and update the price of the particular product. We have omitted the operation for the transition between AdminRequest and AdminConfirm, because it only returns a confirmation. The implementation of the update transaction is defined as in figure 7.4.

7.2.3 Status transaction

The status transaction retrieves the status of a specific order. The user accesses the *Home* page to start, then the user identifies itself by the *OrderInquiry* page. Subsequently, the status of the last order is provided by the *OrderDisplay* page. For each page transition, we assume the operations as given in table 7.3.

For reasons of simplicity, we merge the identification of the user and the display of the last order. The implementation of the *status* transaction then becomes as in figure 7.4.

OPERATION start:

SELECT S_ID INTO :session_id FROM SESSIONID
UPDATE SESSIONID SET S_ID = :session_id + 1

OPERATION search:

/* the user provides the name of an author that is stored in local variable :author_name */
USER_TEXT(:author_name)
SELECT I_ID,I_TITLE,I_COST, A_NAME INTO :author_books FROM ITEM,AUTHOR WHERE I_A_ID = A_ID AND A_NAME = :author_name

OPERATION add:

/* all books in :author_books are presented to the user and for one book with book ID :book_id the user fills out the quantity :quant */
USER_SELECT(:author_books, :book_id, :quant)
SELECT I_ID, I_COST INTO :book FROM :author_books WHERE I_ID = :book_id
INSERT INTO CART VALUES (:session_id,:book.I_ID,:book.I_COST,:quant)

OPERATION order:

SELECT O_ID INTO :order_id FROM ORDERID
UPDATE ORDERID SET O_ID = :order_id + 1
SELECT SUM(C_COST) INTO :total_cost FROM CART WHERE C_SESSION_ID = :session_id
INSERT INTO ORDER VALUES (:order_id,:total_cost)
SELECT C_SESSION_ID,C_I_ID,C_COST,C_QTY INTO :cart FROM CART WHERE C_SESSION_ID = :session_id
DELETE FROM CART WHERE C_SESSION_ID=:session_id
FOR (:c IN :cart)

    SELECT OL_ID INTO :orderline_id FROM ORDERLINEID

    UPDATE ORDERLINEID SET OL_ID = :orderline_id + 1

    SELECT MIN(:c.C_QTY,I_STOCK) INTO :filled[:c] FROM ITEM WHERE I_ID = :c.C_I_ID

    UPDATE ITEM SET I_STOCK = I_STOCK - :filled[:c] WHERE I_ID = :c.C_I_ID

    INSERT INTO ORDERLINE VALUES (:orderline_id,:order_id,:c.C_I_ID,:filled[:c],:c.QTY-:filled[:c])

Figure 7.2: Implementation of the transaction *order*

OPERATION start:
/* AS DEFINED IN TRANSACTION ORDER */

OPERATION search:
/* AS DEFINED IN TRANSACTION ORDER */

OPERATION detail:

USER_SELECT(:AUTHOR_BOOKS, :BOOK_ID)

OPERATION update:

/* THE USER ENTERS THE NEW COST FOR THE BOOK */

USER_TEXT(:COST)

UPDATE ITEM SET I_COST = :COST WHERE I_ID = :BOOK_ID

FIGURE 7.3: IMPLEMENTATION OF THE TRANSACTION *update*

From page	To page	Operation
StartUserSession	Home	start
Home	OrderInquiry	-
OrderInquiry	OrderDisplay	status

Table 7.3: Operations of the transaction *status*

OPERATION start:
/* AS DEFINED IN TRANSACTION ORDER */

OPERATION status:

SELECT MAX(O_ID) INTO :LAST_ORDER_ID FROM ORDER

SELECT O_ID, O_TOTAL FROM ORDER INTO :LAST_ORDER WHERE O_ID = :LAST_ORDER_ID

SELECT OL_ID, OL_O_ID, OL_I_ID, OL_QTY, OL_REMAIN FROM ORDER-LINE INTO :LAST_ORDERLINE WHERE OL_O_ID = :LAST_ORDER_ID

FIGURE 7.4: IMPLEMENTATION OF THE TRANSACTION *status*

CON	start	search	add	order	detail	update	status
start	+	-	-	-	-	-	-
search		-	-	-	-	+	-
add			-	-	-	-	-
order				+	-	-	+
detail					-	-	-
update						+	-
status							-

Table 7.4: Conflict relation CON

7.3 Semantics-based concurrency control

In this section we demonstrate that the semantics of the TPC-W application can be exploited to achieve a higher degree of concurrency than without these semantics. We demonstrate this by applying the approaches of chapter 3, 4 and 5 in section 7.3.1, section 7.3.2 and section 7.3.3.

7.3.1 Semantics-based concurrency control for the basic model

To demonstrate the improvement of the semantics-based approach over the conventional approach, we derive the conventional conflict relation *CON* on the operations of the transactions *order*, *update* and *status*. The conflict relation is derived from the implementations provided in the previous section. Then, we define the semantic conflict relation *SCON* in two steps. First, we specify the semantics for the transactions *order*, *update* and *status*. Second, we derive sufficient semantics for the operations of the transactions. We conclude that concurrency is improved, if the semantic conflict relation *SCON* is weaker than the conflict relation *CON*. By definition 2.2.0.11 of conflict relation *CON*, two operations are conflicting if their composed functional semantics are different in either order. The conflict table is given in table 7.4. For each pair of operations p and q, we write '+' if $CON(p, q)$ and '-' if $\neg CON(p, q)$ or p and q never access the same variable. As the conflict relation is symmetric, only the upper right corner is marked.

We can explain the conflicts as follows:

- $CON(start, start)$: in either order the returned value for :session_id is different,
- $CON(search, update)$: in either order the returned price of a book is different,
- $CON(order, order)$: in either order the returned value for :order_id is different, and the stock levels may be different,
- $CON(update, update)$: in either order the final price of the book is different,
- $CON(order, status)$: in either order the last order of a customer is different.

Next, we demonstrate that for some of the conflicting operations sufficient semantics can be identified that are not *semantically* conflicting. For this, we first define the semantics of the three transactions. Based on these semantics, sufficient semantics for their operations are derived. These semantics are used to define the semantic conflict relation *SCON*.

$R_{T_{order}} = ($
$\exists ordered\_books, order\_id, order\_id\_new, session\_id, session\_id\_new, orderline\_id[] :$
$(order\_id, \ldots) \notin order_{pre} \ \wedge \ (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post}$
$\wedge$
$(session\_id, \ldots) \notin cart_{pre} \wedge$
$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post}$
$\wedge$
/* each ordered book must have a price and author which name is provided by the user */
$ordered\_books \subseteq \{(i\_id, i\_cost, :qty_{pre}[i\_id]) \mid$
$(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge (a\_id, :author\_name_{pre}[a\_id]) \in author_{pre}\}$
$\wedge$
/* the new order is added to the order relation */
$order_{post} = order_{pre} \cup \{(order\_id, \sum_{(b\_id, b\_i\_cost, b\_qty) \in ordered\_books} b\_i\_cost)\}$
$\wedge$
/* for each ordered book, the stock is maximally lowered with the quantity ordered */
$item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge$
$((i\_id, b\_i\_cost, b\_qty) \in ordered\_books \Rightarrow i\_stock\_post = min(i\_stock\_pre, b\_qty)) \wedge$
$((i\_id, b\_i\_cost, b\_qty) \notin ordered\_books \Rightarrow i\_stock\_post = i\_stock\_pre)\}$
$\wedge$
/* for each ordered book an orderline is inserted in the orderline relation with the delivered items and the remaining items */
$orderline_{post} = orderline_{pre} \cup$
$\{(orderline\_id[b], order\_id, b\_id, i\_stock\_pre - i\_stock\_post, b\_qty - (i\_stock\_pre - i\_stock\_post)) \mid$
$(b\_id, b\_i\_cost, b\_qty) \in ordered\_books \wedge$
$(b\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge$
$(b\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \in item_{post}\}$
$\wedge$
/* orderline ID is unique */
$\forall b \in ordered\_books : (orderline\_id[b], \ldots) \notin orderline_{pre}$
$\wedge$
$(orderline\_id\_new, \ldots) \in orderlineid_{post} \wedge (orderline\_id\_new) \notin orderline_{post}$
$\wedge$
$\forall b_i, b_j \in ordered\_books : b_i \neq b_j \Rightarrow orderline\_id[b_i] \neq orderline\_id[b_j]$
$\wedge$
$\mathrm{Inv}(\{:session\_id, :author, :author\_books, :book, cart, :cart, :order\_id, orderid,$
$:total\_cost, :orderline\_id, orderlineid, :filled[], item, orderline\})$
$)$

Figure 7.5: Semantics of the transaction *order*

Semantics of the order transaction

The semantics of the *order* transaction are given in figure 7.5. The first line introduces a set of variables. Importantly, these variables do not represent application variables. Instead, they are only used to support the definition of the semantics and are therefore only defined in the context of these semantics specification. In addition, we captured the multiple entry of the :author_name by the user by the array :author_name[]. Variable *order_id* represents the order id of the transaction. The order id must be unique for each order. This is guaranteed by the conjunct $(order\_id, ...) \notin order_{pre}$. Such *order_id* is guaranteed to exist, because the relation *orderid* maintains the numbers that are not yet used as an order id. This fact is guaranteed by the conjunct:

$$(order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post})$$

. . . for some arbitrary variable *order_id_new*.

It is important to note that these semantics for the *order_id* and *orderid* relation are weaker than the *actual* semantics defined for *order_id* and *orderid*. The reason is that *order_id* is not required to have the same value as $orderid_{pre}$. We will see that this is of primary importance to the gain in concurrency. The semantics for the variables *session_id* and *orderline_id* are defined in a similar way. For the variables *orderline_id*[*b*] an additional constraint is imposed to guarantee that the id's of each of the added orderlines are unique. In order to achieve the semantics of the transaction, we assign the semantics to the operations as given in figure 7.6. Note that in this case program variables are introduced. It can be verified that the operation semantics are sufficient for the transaction.

Semantics of the update transaction

The *update* transaction updates the cost of a book that is selected by the user by means of the author name. The semantics of the *update* transaction are defined as in figure 7.7. The book that is updated, is represented by variable *selected_book*. The selected book is bound to the states $item_{pre}$ and $author_{pre}$ in a similar way as was the case for *ordered_books* in transaction *order*. The author name of the selected book must be equal to $:author\_name_{pre}$. The last conjunct specifies that the price of the selected book is updated.

The update transaction is implemented with the operations *start*, *search*, *detail* and *update*. This leaves us the specification of the semantics for the *detail* and *update* operations. Their semantics are given in figure 7.8. Again it can be verified that the semantics of the operations *start*, *search*, *detail* and *update* are sufficient for the *update* transaction.

Semantics of the status transaction

The *status* transaction displays the order information of the last order for some user. The semantics of the transaction are defined as in figure 7.9. The semantics of the transaction describe the transaction parameters *:last_order_id*, *:last_order* and *:last_orderline* that represent the last order id, its corresponding order and its corresponding orderlines that are returned to the web page. The second conjunct guarantees that this order is the order with the highest order id.

The *status* transaction is implemented by means of the operations *start* and *status*. The semantics of the *start* operation are considered the same as the previously defined semantics.

$R_{start} = (\exists session\_id\_new :$
$(:session\_id_{pre}, \ldots) \notin cart_{pre} \wedge$
$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post}$
$\wedge$
$\text{Inv}(\{:session\_id, sessionid\}))$

$R_{search} = (:author\_books_{post} = \{(i\_id, i\_title, i\_cost, a\_name) \mid$
$(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge (a\_id, :author\_name_{pre}) \in author_{pre}\}$
$\wedge$
$\text{Inv}(\{:author\_books\}))$

$R_{add} = (\exists book\_cost : (:book\_id_{pre}, book\_cost) \in :author\_books_{pre}$
$\wedge$
$cart_{post} = cart_{pre} \cup \{(:session\_id_{pre}, :book\_id_{pre}, book\_cost, :quant_{pre})\}$
$\wedge$
$\text{Inv}(\{cart, :book\}))$

$R_{order} = (\exists order\_id\_new, orderline\_id\_new :$
$(:order\_id_{post}, \ldots) \notin order_{pre} \wedge (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post}$
$\wedge$
$order_{post} = order_{pre} \cup \{(:order\_id_{pre}, \sum_{(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre}} c\_cost)\}$
$\wedge$
$:cart_{post} = \{(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \mid$
$(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} \wedge c\_session\_id \neq :session\_id_{pre}\}$
$\wedge$
$item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge$
$((:session_{pre}\_id, i\_id, c\_cost, c\_qty) \in cart_{pre} \Rightarrow i\_stock\_post = min(i\_stock\_pre, c\_qty) \wedge$
$(:session_{pre}\_id, i\_id, c\_cost, c\_qty) \notin cart_{pre} \Rightarrow i\_stock\_post = i\_stock\_pre)\}$
$\wedge$
$orderline_{post} = orderline_{pre} \cup \{$
$(orderline\_id[c], :order\_id_{pre}, c\_i\_id, i\_stock\_pre - i\_stock\_post, c\_qty - (i\_stock\_post - i\_stock\_pre)) \mid$
$(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} \wedge$
$(c\_i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge$
$(c\_i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \in item_{post}\}$
$\wedge$
$\forall c \in cart_{pre} : (orderline\_id[c], \ldots) \notin orderline_{pre}$
$\wedge$
$(orderline\_id\_new) \in orderlineid_{post} \wedge (orderline\_id\_new, \ldots) \notin orderline_{post}$
$\wedge$
$\forall c_i, c_j \in cart_{pre} : c_i \neq c_j \Rightarrow orderline\_id[c_i] \neq orderline\_id[c_j]$
$\wedge$
$\text{Inv}(\{:order\_id, orderid, :total\_cost, order, :cart, cart, :orderline\_id, orderlineid, orderline\}))$

Figure 7.6: Operation semantics for transaction *order*

$R_{T_{update}} = ($
$\exists selected\_book, session\_id\_new :$

$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{pre}$
$\wedge$
/* each selected book must have a price and author with name author_name that once existed in the database */
$selected\_book \in \{(i\_id, i\_title, i\_cost, :author\_name_{pre}) \mid$
$(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge (a\_id, :author\_name_{pre}) \in author_{pre}\}$

/* the cost of the selected book is updated */
$item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost\_post, i\_stock) \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost\_pre, i\_stock) \in item_{pre} \wedge$
$(i\_id, \ldots) = selected\_book \Rightarrow i\_cost\_post = :cost_{pre} \wedge$
$(i\_id, \ldots) \neq selected\_book \Rightarrow i\_cost\_post = i\_cost\_pre\}$
$\wedge$
$\text{Inv}(\{:session\_id, sessionid, :author\_name, :author\_books, :book\_id, item\})$
$)$

Figure 7.7: Semantics of the transaction *update*

$SCON$	R_{start}	R_{search}	R_{add}	R_{order}	R_{detail}	R_{update}	R_{status}
R_{start}	-!	-	-	-	-	-	-
R_{search}		-	-	-	-	+	-
R_{add}			-	-	-	-	-
R_{order}				+	-	-	+
R_{detail}					-	-	-
R_{update}						+	-
R_{status}							-

Table 7.5: Semantic conflict relation

The semantics of the *status* operation are the same as the semantics of the transaction, except for the predicates on the sessionid relation. The semantics of the update operation are given in figure 7.8. Again, it can be verified that the semantics of the *start* and *status* operations are sufficient for the semantics of the transaction *status*.

The semantic conflict relation

Now we have identified operation semantics that are sufficient for the transaction semantics, we can derive the corresponding semantic conflict relation. The resulting conflict table is given in table 7.5. The differences with respect to the previous conflict relation of table 7.4 are marked with an exclamation mark.

The semantics result in the disappearance of one conflict with respect to CON. The conflict $CON(start, start)$ disappears in $SCON$, because R_{start} only states that the session

$$
\begin{array}{l}
R_{detail} = (\\
(:book\_id_{pre}, \ldots) \in :author\_books_{pre} \\
\wedge \\
\mathrm{Inv}(\{:book\}) \\
)
\end{array}
$$

$$
\begin{array}{l}
R_{update} = (\\
item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost\_post, i\_stock) \mid \\
(i\_id, i\_a\_id, i\_title, i\_cost\_pre, i\_stock) \in item_{pre} \wedge \\
i\_id = :book\_id_{pre} \Rightarrow i\_cost\_post = :cost_{pre} \wedge \\
i\_id \neq :book\_id_{pre} \Rightarrow i\_cost\_post = i\_cost\_pre\} \\
\wedge \\
\mathrm{Inv}(\{item\}) \\
)
\end{array}
$$

$$
\begin{array}{l}
R_{status} = (\\
:last\_order\_id_{post} = max\{o\_id \mid (o\_id, o\_total) \in order_{pre}\} \\
\wedge \\
:last\_order_{post} \in \{(o\_id, o\_total) \mid (o\_id, o\_total) \in order_{pre} \wedge o\_id = :last\_order\_id_{post}\} \\
\wedge \\
:last\_orderline_{post} = \{(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \mid \\
(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \in orderline_{pre} \wedge ol\_o\_id = :last\_order\_id_{post}\} \\
\wedge \\
\mathrm{Inv}(\{:last\_order\_id, :last\_order, :last\_orderline, orderid\}))
\end{array}
$$

Figure 7.8: Semantics of the operations *detail*, *update* and *status*

$$
\begin{array}{l}
R_{T_{status}} = (\\
\exists session\_id\_new : \\
\\
(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post} \\
\wedge \\
:last\_order\_id_{post} = max\{o\_id \mid (o\_id, o\_total) \in order_{pre}\} \\
\wedge \\
:last\_order_{post} \in \{(o\_id, o\_total) \mid (o\_id, o\_total) \in order_{pre} \wedge o\_id = :last\_order\_id_{post}\} \\
\wedge \\
:last\_orderline_{post} = \{(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \mid \\
(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \in orderline_{pre} \wedge ol\_o\_id = :last\_order\_id_{post}\} \\
\wedge \\
\mathrm{Inv}(\{:last\_order\_id, :last\_order, :last\_orderline, orderid\}) \\
)
\end{array}
$$

Figure 7.9: Semantics of the transaction *status*

id must be unique. This contrasts with the definition of *CON* where the *value* of :*session_id* must be the same. Even though only one conflict has disappeared, this may already result in higher degrees of concurrency, because *start* is the first operation of each transaction. However, we will see that we achieve far better results in the context of the tree model.

7.3.2 Semantics-based concurrency control for the tree model

The consideration of semantics in the previous section improved concurrency with respect to the conventional approach. In this section we investigate the improvement in concurrency in the context of the tree model.

The main limitation of the semantics-based approach of the previous section is the fact that only a single database transition is considered for the semantics of transaction order. In the context of the tree model, we can consider multiple transitions, i.e. the transitions of each of the operations. We reference to the transaction tree for transaction order as Tr_{order}.

To capture these multiple transitions in the specification of transaction order, we refer to the application variables of operation p by a superscript p. Thus, if operation p references variables $item_{pre}$ and $item_{post}$, we write $item^p_{pre}$ and $item^p_{post}$. The semantics of transaction tree Tr_{order} can now be specified as $R_{Tr_{order}}$ as given in figure 7.10.

We remark that the specification of $R_{Tr_{order}}$ is much weaker than the earlier specification R_{T_order}. This follows directly from the fact that the references to the initial and final states of the operations are *hidden* through the new variables, i.e. $order^{order}_{pre,post}$, $orderline^{order}_{pre,post}$, $item^{search}_{pre,post}$ and are therefore not related anymore to the initial and final states of the transaction. We can allow this, because we assume that each of the operations satisfies the original semantics as given in section 7.3. However, to guarantee that the transaction semantics are correct[1], we must extend the weak specification of $R_{Tr_{order}}$ with the following constraints on the database:

- $(orderid, \ldots) \in order_{post} \wedge$
  $\forall b \in ordered\_books : (orderlineid[b], order\_id, \ldots) \in orderline_{post}$
- $\forall (i\_id, b\_i\_cost, b\_qty) \in ordered\_books, (i\_id, i\_a\_id, i\_title, i\_cost, i\_stock) \in item_{post} :$
  $i\_stock \geq 0$

The first constraint implies that the added order is in the database, and that all orderlines of the order are in the database. The second constraint implies that the modified stock levels are never less than zero.

The sufficient semantics of the operations in the context of the new specification are now also weaker, because they need only implement the semantics $R_{Tr_{order}}$. The corresponding operation semantics are given in figure 7.11. We describe the semantics for the transactions T_{update} and T_{status} in a similar way. We refer to figures 7.12 and 7.13 for the semantics of transaction trees Tr_{update} and Tr_{status}, and to figure 7.14 for the semantics of the operations *detail*, *update* and *status* in the context of these semantics. We note that an additional predicate was added to the semantics of transacton tree status, to guarantee that all orderlines are read by the status transaction. This is also reflected in the semantics of operation

[1]The semantics of a transaction must preserve database consistency (see definition 3.4.1.1).

$SCON$	R_{start}	R_{search}	R_{add}	R_{order}	R_{detail}	R_{update}	R_{status}
R_{start}	-!	-	-	-	-	-	-
R_{search}		-	-	-	-	-!	-
R_{add}			-	-	-	-	-
R_{order}				-!	-	-	-!
R_{detail}					-	-	-
R_{update}						-!	-
R_{status}							-

Table 7.6: Semantic conflict relation for the tree model

status. The new semantic conflict relation is represented in table 7.6. The differences with respect to table 7.5 are marked with an exclamation mark.

The weaker semantics have *dramatic* consequences for the semantic conflict relation. With the refined semantics for the transaction *order*, *all* semantic conflicts have disappeared. In particular, the semantic conflict between the semantics of the *update* operation and the *search* operation has disappeared. This is explained from the fact that in either order, the specification of $:author\_books_{post}$ is met. The semantic conflict between semantics of two *order* operations has also disappeared, because in either order the *orderid* and *orderlineid*'s remain unique, and the added orders and orderlines exist in either order in the database. The semantic conflict between operations *order* and *status* has disappeared, because operation *order* always adds all orderlines for each order. Even the semantic conflict between two *update* operations has disappeared, because in either order they perform their update on the database.

Obviously, now all conflicts have disappeared, semantic conflict serializability is guaranteed for *all possible interleavings* of operations. In fact, no concurrency control is necessary at the higher level, such that we achieve *maximum* concurrency here. We need only guarantee semantic conflict serializability with respect to the individual operations.

7.3.3 Semantic decomposition

In this section, we take the approach of semantic decomposition to improve concurrency. We start with the semantics of section 7.3, to clearify the added value of the approach with respect to the basic semantic approach.[2] We start with the semantics of section 7.3. We observe that the semantics of the *order* transaction as given in figure 7.5 are too strong, because they state that the observed stock for all ordered books must come from the same database snapshot. Instead, it is also sufficient if the observed stock levels come from different database snapshots. This weaker requirement can be captured by distinguishing multiple consistent transitions. We decompose transaction *order* into a transition $T_{order.select}$ that selects the books to be ordered, a set of transitions $T_{order.stock[b]}$ that update the stock for each ordered book b, and a transition $T_{order.order}$ that updates the *order* and *orderline* relations. In addition, we define a partial order $<_{R_{T_{order}}}$ on the transitions, with:

[2]We note that it is possible to use semantic decomposition in the context of the tree model as well.

$R_{Tr_{order}} = (\exists order^{order}_{pre,post}, orderline^{order}_{pre,post}, item^{search[]}_{pre,post},$
$ordered\_books, order\_id, order\_id\_new, session\_id, session\_id\_new, orderline\_id[] :$
$(order\_id, \ldots) \notin order_{pre} \ \wedge \ (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post}$
$\wedge$
$(session\_id, \ldots) \notin cart_{pre} \wedge$
$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post}$
$\wedge$
/* each ordered book must have a price and author which name is provided by the user */
$ordered\_books \subseteq \{(i\_id, i\_cost, :qty_{pre}[i\_id]) \mid (i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item^{search[i\_id]}_{pre} \wedge (a\_id, :author\_name_{pre}[a\_id]) \in author^{search[i\_id]}_{pre}\}$
$\wedge$
/* the new order is added to the order relation */
$order^{order}_{post} = order^{order}_{pre} \cup \{(order\_id, \sum_{(b\_id, b\_i\_cost, b\_qty) \in ordered\_books} b\_i\_cost)\} \wedge$
$(orderid, \ldots) \in order_{post}$
$\wedge$
/* for each ordered book, the stock is maximally lowered with the quantity ordered */
$item^{order}_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item^{order}_{pre} \wedge$
$((i\_id, b\_i\_cost, b\_qty) \in ordered\_books \Rightarrow i\_stock\_post = min(i\_stock\_pre, b\_qty)) \wedge$
$((i\_id, b\_i\_cost, b\_qty) \notin ordered\_books \Rightarrow i\_stock\_post = i\_stock\_pre)\}$
$\wedge$
$\forall (i\_id, b\_i\_cost, b\_qty) \in ordered\_books,$
$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock) \in item_{post} : i\_stock \geq 0$
$\wedge$
/* for each ordered book an orderline is inserted in the orderline relation with the delivered items and the remaining items */
$orderline^{order}_{post} = orderline^{order}_{pre} \cup$
$\{(orderline\_id[b], order\_id, b\_id, i\_stock\_pre - i\_stock\_post, b\_qty - (i\_stock\_pre - i\_stock\_post)) \mid$
$(b\_id, b\_i\_cost, b\_qty) \in ordered\_books \wedge$
$(b\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item^{order}_{pre} \wedge$
$(b\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \in item^{order}_{post}\}$
$\wedge$
$\forall b \in ordered\_books : (orderlineid[b], orderid, \ldots) \in orderline_{post}$
$\wedge$
/* orderline ID is unique */
$\forall b \in ordered\_books : (orderline\_id[b], \ldots) \notin orderline_{pre}$
$\wedge$
$(orderline\_id\_new, \ldots) \in orderlineid_{post} \wedge (orderline\_id\_new) \notin orderline_{post}$
$\wedge$
$\forall b_i, b_j \in ordered\_books : b_i \neq b_j \Rightarrow orderline\_id[b_i] \neq orderline\_id[b_j]$
$\wedge$
$\mathrm{Inv}(\{:session\_id, :author, :author\_books, :book, cart, :cart, :order\_id, orderid,$
$:total\_cost, :orderline\_id, orderlineid, :filled[], item, orderline\}))$

Figure 7.10: Semantics of the transaction tree *order*

$$R'_{start} = R_{start}$$

$$R'_{search} = (\exists item^{search}_{pre}, author^{search}_{pre} :$$
$$:author\_books_{post} = \{(i\_id, i\_title, i\_cost, a\_name) \mid$$
$$(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item^{search}_{pre} \wedge (a\_id, :author\_name_{pre}) \in author^{search}_{pre}\}$$
$$\wedge$$
$$\mathrm{Inv}(\{:author\_books\}))$$

$$R'_{add} = R_{add}$$

$$R'_{order} = (\exists item^{order}_{pre,post}, orderline^{order}_{pre,post}, order\_id\_new, orderline\_id\_new :$$
$$(:order\_id_{post}, \ldots) \notin order_{pre} \wedge (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post}$$
$$\wedge$$
$$order^{order}_{post} = order^{order}_{pre} \cup \{(:order\_id_{pre}, \textstyle\sum_{(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre}} c\_cost)\}$$
$$\wedge$$
$$(:order\_id_{pre}, \ldots) \in order_{post}$$
$$\wedge$$
$$:cart_{post} = \{(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \mid$$
$$(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} \wedge c\_session\_id \neq :session\_id_{pre}\}$$
$$\wedge$$
$$item^{order}_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \mid$$
$$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item^{order}_{pre} \wedge$$
$$((:session_{pre}\_id, i\_id, c\_cost, c\_qty) \in cart_{pre} \Rightarrow i\_stock\_post = min(i\_stock\_pre, c\_qty) \wedge$$
$$(:session_{pre}\_id, i\_id, c\_cost, c\_qty) \notin cart_{pre} \Rightarrow i\_stock\_post = i\_stock\_pre)\}$$
$$\wedge$$
$$\forall (:session\_id_{pre}, i\_id, c\_cost, c\_qty) \in cart_{pre},$$
$$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock) \in item_{post} : i\_stock \geq 0$$
$$\wedge$$
$$orderline^{order}_{post} = orderline^{order}_{pre} \cup \{$$
$$(orderline\_id[c], :order\_id_{pre}, c\_i\_id, i\_stock\_pre - i\_stock\_post, c\_qty - (i\_stock\_post - i\_stock\_pre)) \mid$$
$$(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} \wedge$$
$$(c\_i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item^{order}_{pre} \wedge$$
$$(c\_i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \in item^{order}_{post}\}$$
$$\wedge$$
$$\forall c \in cart_{pre} : (orderline\_id[c], \ldots) \in orderline_{post} \wedge$$
$$\forall c \in cart_{pre} : (orderline\_id[c], \ldots) \notin orderline_{pre}$$
$$\wedge$$
$$(orderline\_id\_new) \in orderlineid_{post} \wedge (orderline\_id\_new, \ldots) \notin orderline_{post}$$
$$\wedge$$
$$\forall c_i, c_j \in cart_{pre} : c_i \neq c_j \Rightarrow orderline\_id[c_i] \neq orderline\_id[c_j]$$
$$\wedge$$
$$\mathrm{Inv}(\{:order\_id, orderid, :total\_cost, order, :cart, cart, :orderline\_id, orderlineid, orderline\}))$$

Figure 7.11: Operation semantics for the transaction tree *order*

$R_{Tr_{update}} = ($
$\exists selected\_book, session\_id\_new, item^{update}_{pre,post}, item^{search[]}_{pre}, author^{search[]}_{pre} :$

$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{pre}$
$\wedge$
/* each selected book must have a price and author with name author_name that once existed in the database */
$selected\_book \in \{(i\_id, i\_title, i\_cost, :author\_name_{pre}) \mid$
$(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item^{search[i\_id]}_{pre} \wedge$
$(a\_id, :author\_name_{pre}) \in author^{search[i\_id]}_{pre}\}$

/* the cost of the selected book is updated */
$item^{update}_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost\_post, i\_stock) \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost\_pre, i\_stock) \in item^{update}_{pre} \wedge$
$(i\_id, \ldots) = selected\_book \Rightarrow i\_cost\_post = :cost_{pre} \wedge$
$(i\_id, \ldots) \neq selected\_book \Rightarrow i\_cost\_post = i\_cost\_pre\}$
$\wedge$
$\mathrm{Inv}(\{:session\_id, sessionid, :author\_name, :author\_books, :book\_id, item\})$
$)$

Figure 7.12: Semantics of the transaction tree *update*

$R_{Tr_{status}} = ($
$\exists session\_id\_new, order^{status}_{pre}, orderline^{status}_{pre} :$

$(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post} \wedge :last\_order\_id_{post} = max\{o\_id \mid (o\_id, o\_total) \in order^{status}_{pre}\}$
$\wedge$
$:last\_order_{post} \in \{(o\_id, o\_total) \mid (o\_id, o\_total) \in order^{status}_{pre} \wedge o\_id = :last\_order\_id_{post}\}$
$\wedge$
$:last\_orderline_{post} = \{(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \mid$
$(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \in orderline^{status}_{pre} \wedge ol\_o\_id = :last\_order\_id_{post}\}$
$\wedge$
$\forall (ol\_id, :last\_order\_id_{post}, ol\_i\_id, ol\_qty) \in orderline_{post} :$
$(ol\_id, :last\_order\_id_{post}, ol\_i\_id, ol\_qty) \in :last\_orderline_{post}$
$\wedge$
$\mathrm{Inv}(\{:last\_order\_id, :last\_order, :last\_orderline, orderid\})$
$)$

Figure 7.13: Semantics of the transaction tree *status*

$$R'_{detail} = R_{detail}$$

$$
\begin{aligned}
&R'_{update} = (\\
&\exists item^{update}_{pre,post} : item^{update}_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost\_post, i\_stock) \mid \\
&(i\_id, i\_a\_id, i\_title, i\_cost\_pre, i\_stock) \in item^{update}_{pre} \wedge \\
&i\_id = :book\_id_{pre} \Rightarrow i\_cost\_post = :cost_{pre} \wedge \\
&i\_id \neq :book\_id_{pre} \Rightarrow i\_cost\_post = i\_cost\_pre\} \\
&\wedge \\
&\mathrm{Inv}(\{item\}) \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&R'_{status} = (\\
&\exists order^{status}_{pre}, orderline^{status}_{pre} : \\
&\\
&:last\_order\_id_{post} = max\{o\_id \mid (o\_id, o\_total) \in order^{status}_{pre}\} \\
&\wedge \\
&:last\_order_{post} \in \{(o\_id, o\_total) \mid (o\_id, o\_total) \in order^{status}_{pre} \wedge o\_id = :last\_order\_id_{post}\} \\
&\wedge \\
&:last\_orderline_{post} = \{(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \mid \\
&(ol\_id, ol\_o\_id, ol\_i\_id, ol\_qty, ol\_remain) \in orderline^{status}_{pre} \wedge ol\_o\_id = :last\_order\_id_{post}\} \\
&\wedge \\
&\forall (ol\_id, :last\_order\_id_{post}, ol\_i\_id, ol\_qty) \in orderline_{post} : \\
&(ol\_id, :last\_order\_id_{post}, ol\_i\_id, ol\_qty) \in :last\_orderline_{post} \\
&\wedge \\
&\mathrm{Inv}(\{:last\_order\_id, :last\_order, :last\_orderline, orderid\}) \\
&)
\end{aligned}
$$

Figure 7.14: Semantics of the operations *detail*, *update* and *status* for the transaction trees

$<_{R_{T_{order}}} = \bigcup_b \{(R_{T_{order.select}}, R_{T_{order.stock[b]}}), (R_{T_{order.stock[b]}}, R_{T_{order.order}})\}$

The semantics are defined as in figure 7.15.

Unfortunately, the implementation suggested by the TPC-W specification cannot exploit these weaker semantics. This is due to the fact that the *order* operation atomically updates the stock of all ordered books. As a consequence, the stock levels of all books are always observed from and updated into a single snapshot.

To still be able to demonstrate the power of the approach, we split the *order* operation into an operation *order'* and multiple operations *orderline* for each ordered book. The operation *order'* contains the implementation for the update of the *order* and *cart* relations. The *orderline* operations contain the updates for the stock levels and the orderlines. The implementations of the operations *order'* and *orderline* operations are given in figure 7.16.

The operation *order'* inserts a new order in the order relation and moves the contents of the shopping cart to the local variable *:cart*. The operations *orderline* update the stock for each ordered book and insert the number of delivered books into the orderline relation.

The next step is to assign decomposed semantics to each of new the operations. For transition $T_{order.select}$ we have the relevant operation semantics as given in figure 7.18. According to these semantics, the transition $T_{order.select}$ is completely implemented by the first three operations of the order transaction.

For transition $T_{order.stock}$ we have the relevant operation semantics as given in figure 7.3.3. None of the semantics of operations *start*, *search* and *stock* are assigned to transition $T_{order.stock}$. Instead, part of the semantics of operations *order'* and *orderline* are assigned to $T_{order.stock}$. The semantics for *order'* include the preparation of the local variable *:cart* for the set of *orderline* operations. The relevant operation semantics of the *orderline* operations involve the update of the stock level of each item in *:cart*. The amount filled is stored in local variables *:filled[:c]* for each item $:c \in :cart_{pre}$.

For transition $T_{order.order}$ we have the relevant operation semantics as given in figure 7.19. Again, no semantics of the *start*, *search* and *add* operations are assigned to transition $T_{order.order}$. The relevant semantics of operation *order'* include the generation of a unique *:order_id*, the clean-up of the *cart* relation and the insertion of the order in the *order* relation. The semantics of the *orderline* operations involve the insertion of an orderline for each ordered book. For this, a unique *orderline_id* is generated.

With the decomposition of the *order* operation into the operations *order'* and *orderline* with corresponding decomposed semantics, a new semantic conflict relation is defined. The new semantic conflict relation is depicted in table 7.7 . We abbreviated *order'.stock*, *orderline.stock*, *order'.order*, *orderline.order* to *o'.s*, *ol.s*, *o'.o* and *ol.o* respectively. The differences with respect to the previous conflict relation of table 7.5 are marked with an exclamation mark.

With the decomposition of the *order* operation, the previous conflict $SCON(R_{order}, R_{order})$ is now spread over the semantic conflicts on the operations *order'.order*, *orderline.order*, *order'.stock*, and *orderline.stock*. If the decomposition did not have any effect on concurrency, we would have conflicts for all combinations of these operation semantics. However, the table suggests that this is not the case. Instead, we see that with the new semantics we even have only one conflict $SCON(R_{orderline.stock}, R_{orderline.stock})$ left. The reason for this conflict

$$
\begin{aligned}
&R_{T_{order.select}} = \\
&\exists session\_id\_new, ordered\_books : \\
&(:session\_id_{post}, \ldots) \in cart_{pre} \wedge \\
&(session\_id\_new) \in sessionid_{post} \wedge (session\_id\_new, \ldots) \notin cart_{post} \\
&\wedge \\
&ordered\_books \subseteq \{(i\_id, i\_cost, :qty(i\_id) \mid \\
&(i\_id, a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge (a\_id, :author\_name[a_i d]) \in author_{pre}\} \\
&\wedge \\
&cart_{post} = cart_{pre} \cup \{(:session\_id_{pre}, i\_id, i\_cost, i\_qty) \mid \\
&(i\_id, i\_cost, i\_qty) \in ordered\_books\} \\
&\wedge \\
&\mathrm{Inv}(\{:session\_id, sessionid, :author\_books, cart, :book, :author\_name, :book\_id\}) \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&R_{T_{order.stock}}[(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre}] = (\\
&:filled_{pre}[c\_i\_id] = min(c\_qty, \\
&max\{i\_stock \mid (i\_id, i\_a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge i\_id = c\_qty\}) \\
&\wedge \\
&item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_post) \mid \\
&(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge \\
&(i\_id = c\_i\_id \Rightarrow i\_stock\_post = min(i\_stock\_pre, c\_i\_qty)) \wedge \\
&(i\_id \neq c\_i\_id \Rightarrow i\_stock\_post = i\_stock\_pre)\} \\
&\wedge \\
&\mathrm{Inv}(\{item, :filled[]\}) \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&R_{T_{order.order}} = (\\
&\exists order\_id, order\_id\_new, orderline\_id[], orderline\_id\_new : \\
&(order\_id, \ldots) \notin order_{pre} \wedge (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post} \\
&\wedge \\
&order_{post} = order_{pre} \cup \{(order\_id, \textstyle\sum_{:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty \in cart_{pre}} c\_i\_cost)\} \\
&\wedge \\
&orderline_{post} = orderline_{pre} \cup \{ \\
&(orderline\_id[c\_i\_id], order\_id, c\_i\_id, :filled_{pre}[c\_i\_id], c\_qty - :filled[c\_i\_id]) \mid \\
&(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre}\} \\
&\wedge \\
&\forall (:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} : (orderline\_id[c\_i\_id], \ldots) \notin orderline_{pre} \\
&\wedge \\
&\forall c_i, c_j \in cart_{pre} : c_i \neq c_j \Rightarrow orderline\_id[c_i] \neq orderline\_id[c_j] \\
&\wedge \\
&\mathrm{Inv}(\{cart, :cart, :orderline\_id, orderlineid, orderline, :order\_id, orderid, :total\_cost, order\}) \\
&)
\end{aligned}
$$

Figure 7.15: Decomposed transaction semantics

OPERATION order':

SELECT ORDER_ID INTO :ORDER_ID FROM ORDERID
UPDATE ORDERID SET ORDER_ID = :ORDER_ID + 1

SELECT SUM(C_COST) INTO :TOTAL_COST FROM CART WHERE C_SESSION_ID = :SESSION_ID

INSERT INTO ORDER VALUES (:ORDER_ID,:TOTAL_COST)

SELECT C_SESSION_ID,C_I_ID,C_COST,C_QTY INTO :CART FROM CART WHERE C_SESSION_ID = :SESSION_ID

DELETE FROM CART WHERE C_SESSION_ID=:SESSION_ID

OPERATION orderline (:c ∈ :cart):

SELECT OL_ID INTO :ORDERLINE_ID FROM ORDERLINEID
UPDATE ORDERLINEID SET OL_ID = :ORDERLINE_ID + 1

SELECT MIN(:C.QTY,I_STOCK) INTO :FILLED[:C] FROM ITEM WHERE I_ID = :C.C_I_ID
UPDATE ITEM SET I_STOCK = I_STOCK - :FILLED[:C] WHERE I_ID = :C.C_I_ID

INSERT INTO ORDERLINE VALUES
(:ORDERLINE_ID,:ORDER_ID,:C.C_I_ID,:FILLED[:C],:C.C_QTY−:FILLED[:C])

FIGURE 7.16: IMPLEMENTATION OF THE *order'* AND *orderline* OPERATIONS

$R_{start.select} = R_{start}$

$R_{search.select} = R_{search}$

$R_{add.select} = R_{add}$

$R_{order'.select} = Inv(\emptyset)$

$R_{orderline.select} = Inv(\emptyset)$

Figure 7.17: Operation semantics of transition *order.select*

$R_{start.stock} = Inv(\emptyset)$

$R_{search.stock} = Inv(\emptyset)$

$R_{add.stock} = Inv(\emptyset)$

$R_{order'.stock} = ($
$:cart_{post} = \{(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \mid$
$(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \in :cart_{pre} \wedge c_s ession_i d = :session\_id_{pre}\}$
$\wedge$
Inv($\{:cart\}$)
)

$R_{orderline.stock}[(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \in :cart_{pre}] = ($
/* for each ordered book, the stock is maximally lowered with the quantity ordered */
$:filled_{post}[c\_i\_id] = min(c\_i\_qty, max\{i\_stock \mid$
$(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock) \in item_{pre} \wedge i\_id = c\_i\_qty\})$
$\wedge$
$item_{post} = \{(i\_id, i\_a\_id, i\_title, i\_cost, i\_stock\_pre) \in item_{pre} \wedge$
$(i\_id = c\_i\_id \Rightarrow i\_stock\_post = min(i\_stock\_pre, c\_i\_qty)) \wedge$
$(i\_id \neq c\_i\_id \Rightarrow i\_stock\_post = i\_stock\_pre)\}$
$\wedge$
Inv($\{item, :filled[]\}$)
)

Figure 7.18: Operation semantics for transition $T_{order.stock}$

SCON	R_{start}	R_{search}	R_{add}	$R_{o'.s}$	$R_{ol.s}$	$R_{o'.o}$	$R_{ol.o}$	R_{detail}	R_{update}	R_{status}
R_{start}	-	-	-	-	-	-	-	-	-	-
R_{search}		-	-	-	-	-	-	-	+	-
R_{add}			-	-	-	-	-	-	-	-
$R_{o'.s}$				-!	-!	-!	-!	-	-	-!
$R_{ol.s}$					+	-!	-!	-	-	-!
$R_{o'.o}$						-!	-!	-	-	+
$R_{ol.o}$							-!	-	-	+
R_{detail}								-	-	-
R_{update}									+	-
R_{status}										-

Table 7.7: Semantic conflict relation for decomposed semantics

$R_{start.order} = Inv(\emptyset)$

$R_{search.order} = Inv(\emptyset)$

$R_{add.order} = Inv(\emptyset)$

$R_{order'.order} = ($
$\exists order\_id, order\_id\_new :$
$(order\_id, \ldots) \notin order_{pre} \wedge (order\_id\_new) \in orderid_{post} \wedge (order\_id\_new, \ldots) \notin order_{post}$
$\wedge$
$order_{post} = order_{pre} \cup \{(order\_id, \sum_{(:session\_id_{pre}, c\_i\_id, c\_cost, c\_qty) \in cart_{pre}} c\_cost\}$
$\wedge$
$cart_{post} = \{(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \mid$
$(c\_session\_id, c\_i\_id, c\_cost, c\_qty) \in cart_{pre} \wedge c\_session\_id \neq :session\_id\}$
$\wedge$
$\mathrm{Inv}(\{order, cart, :order\_id, order, :total\_cost\})$
$)$

$R_{orderline.order}(:s \in :cart) = ($
$unique(:orderline\_id_{post})$
$\wedge$
$orderline_{post} = orderline_{pre} \cup$
$\{(:orderline\_id_{post}, :order\_id_{pre}, :s.I\_ID, :s.I\_COST, :filled_{pre}, :stock\_old_{pre} - :filled_{pre})\}$
$\wedge$
$\mathrm{Inv}(\{orderline, :orderline\_id\})$
$)$

Figure 7.19: Operation semantics for transition $T_{order.order}$

operation	operation$^-$	description
start	start$^-$	the session ID is decreased by one
search	search$^-$	the search results are discarded
add	add$^-$	the added item is removed from the shopping cart
order'	order'$^-$	the session ID is decreased by one and the order is removed from the order relation
orderline	orderline$^-$	the orderline ID is decreased by one, the orderline is removed from the orderline relation, the book stock level is increased
detail	detail$^-$	the detail data of the book are discarded
update	update$^-$	the original price of the book is restored
status	status$^-$	the order status information is discarded

Table 7.8: Overview of compensating operations

is that the values of OL_QTY and OL_REMAIN may be different if two orderline operations are executed in different order. Also note that the semantic decomposition for operations *order'* and *orderline* results in the removal of semantic conflicts between $R_{order'.stock}$ and R_{status}, and $R_{orderline.stock}$ and R_{status}. The reason for this is that the status is not influenced by changes in the stock level.

We note that with the decomposition, the desirable property of arbitrary interleaving of operations is lost. This is due to the fact that the *status* operation cannot be interleaved with the *order'* and *orderline* operations for transitions $T_{order.order}$. The intuition behind this, is that in the case of interleaving, the *status* operation would not read all orderlines. This is however the only guarantee that must be provided by the concurrency controller, and thus we conclude that concurrency is improved over the case without decomposition.

7.4 Semantics-based recovery

In this section, we demonstrate that recovery can be improved by exploiting application semantics. We proceed by first applying the approach where no application semantics are exploited and the conventional conflict relation *CON* is used. In the second step, we exploit application semantics.

We start with the observation that all operations are compensatable. The compensating operations for each compensatable operation are described in table 7.8 together with an informal description of their semantics.

For verifying reducibility of a schedule, we need to define the conflict relation among the compensatable and compensating operations. Note that we need not define the conflict relation among two compensating operations, because the protocols developed in chapter 6 do not require these conflicts. The relevant conflicts are given in table 7.9 .

The operations *start* and *start*$^-$ conflict in *CON*, because *start* returns a different session ID in the local variable :session_id when executed in different order. This also applies to $CON(order', order'^-)$ and $CON(orderline, orderline^-)$ with respect to the program variables :*order_id* and :*orderline_id*. The operation *search* and *update*$^-$ conflict, because

CON	$start^-$	$search^-$	add^-	$order'^-$	$orderline^-$	$detail^-$	$update^-$	$status^-$
start	+	-	-	-	-	-	-	-
search	-	-	-	-	-	-	+	-
add	-	-	-	-	-	-	-	-
order'	-	-	-	+	-	-	-	-
orderline	-	-	-	-	+	-	-	-
detail	-	-	-	-	-	-	+	-
update	-	-	-	-	-	-	+	-
status	-	-	-	+	+	-	-	-

Table 7.9: Conflict relation on compensating operations

$update^-$ restores the previous price and thus the retrieved price by search would be different in either order. The same applies to operation *detail* and $update^-$. The operations *update* and $update^-$ conflict, because in either order the resulting price of the book is different. Finally, the $order'^-$ and $orderline^-$ operations conflict with operation *status*, because status returns different results in either order.

Next, we consider the case where application semantics are exploited. The resulting semantic conflict table is depicted in table 7.10. Again, the differences with respect to the conventional case are marked with an exclamation mark. In this case, we see that the conflict between *start* and $start^-$ disappears. The reason is that from the application's point of view, it is not required that $start^-$ compensates for operation *start*, because it is only required that the session ID is unique. Hence, we can consider both states semantically equivalent.[3] As a result, the compensating operation for operation *start* can have null semantics. Unfortunately, this conflict is also the only conflict that disappears when considering the application semantics in this way (see table 7.10). This should not come as a surprise, because backward recovery imposes very strong constraints on the semantics of the compensatable and compensating operations by definition 6.2.3.3. Fortunately, the semantic conflict relation has no disastrous effects on concurrency, except that commits may be delayed for a relatively long time, and the chance of cascaded aborts is enlarged. For example, if a status operation reads from an uncommitted order, it cannot commit and aborts if the order aborts.
At this point, the power of forward recovery can be exploited. As a start, we can prevent the *status* operation to read uncommitted orders by declaring the operation non-compensatable. Then, by protocol *SFBSF* (see protocol 6.4.2.5) the operation *status* will only read committed orders. In a similar way, we can exploit forward recovery by making the order operation non-compensatable with the corresponding orderline operations as forward recovery set. This way, it is avoided that the order is lost after the order was added to the *order* relation.

7.5 Conclusions

In this chapter we demonstrated how application semantics can be exploited by our theory to improve concurrency and recovery. We demonstrated this for the TPC-W benchmark

[3] See definition 6.2.3.3

$SCON$	R_{start^-}	R_{search^-}	R_{add^-}	$R_{order'^-}$	$R_{orderline^-}$	R_{detail^-}	R_{update^-}	R_{status^-}
R_{start}	-!	-	-	-	-	-	-	-
R_{search}	-	-	-	-	-	-	+	-
R_{add}	-	-	-	-	-	-	-	-
$R_{order'}$	-	-	-	+	-	-	-	-
$R_{orderline}$	-	-	-	-	+	-	-	-
R_{detail}	-	-	-	-	-	-	+	-
R_{update}	-	-	-	-	-	-	+	-
R_{status}	-	-	-	+	+	-	-	-

Table 7.10: Semantic conflict relation on semantics of compensating operations

application. We choose this application, because it is representative for novel applications with high transaction processing demands. For the discussion, we considered a representative subset of the application data and three possible transactions *order*, *update* and *status*.

In the first part of the chapter, we demonstrated improvement of concurrency by comparing the conflict relation CON with the semantic conflict relation $SCON$. For the application, the semantic conflict relation $SCON$ proves to be slightly weaker than the conflict relation CON, such that concurrency is improved under conflict based concurrency protocols. After that, we modeled the semantics by means of the tree model. Here, it appears that we can allow *arbitrary* interleaving of the operations while preserving correctness. This means that we achieve *maximum* concurrency for the application. Finally, we demonstrated how the decomposition approach can be applied to improve concurrency. Initially, it appears, that the decomposition has no effect on concurrency because the granularity of the operations is too high. Therefore, we split the problematic operation into two smaller operations. In this situation, it appears that we *almost* achieve arbitrary interleaving: we only require that the *status* operation is not interleaved between the newly introduced operations.

In the second part of the chapter, we demonstrated improvement of recovery by exploiting application semantics. As a first step, we derived the state-based conflict relation CON for each pair of compensatable and compensating operation. In a second step, we derived the semantic conflict relation $SCON$ based on the application semantics. We demonstrated that $SCON$ is only slightly weaker than CON: only one conflict disappears. We argue that this can be expected for backward recovery, because of the strong definition of sufficient compensating semantics. This motivates our extension of conventional theory with forward recovery for the application. We sketched how this theory can be applied to the application.

Chapter 8

Conclusions and future work

8.1 Conclusions

In this thesis we have demonstrated how application semantics can be exploited for improving concurrency and recovery in database systems. We developed a theory and practical concurrency controls. We recall the research questions defined in the introduction and discuss the answers to these questions below.

What application semantics should be included in the transaction model?

We have answered this question in two steps. First, we considered application semantics to improve concurrency in chapter 3, chapter 4 and chapter 5. Second, we considered semantics to improve recovery in chapter 6.

For improving concurrency, we have proposed the following application semantics:

- operation and transaction semantics,
- semantics of sets of operations,
- decomposed transaction semantics.

In chapter 3 we proposed to exploit *operation and transaction semantics* to improve concurrency. We defined these semantics in terms of *relations* on the states of database variables, program variables and transaction parameters. In line with existing theory, we have defined a schedule correct, if the semantics of the schedule imply the semantics of a serial execution of transactions.

The motivation for expressing semantics in terms of *relations* on application states, instead of functions, is that it allows for expressing correctness in terms of a *set* of acceptable states, instead of *a single state* in conventional approaches. Because more states can be defined correct, more concurrency can be allowed.

An important result we achieved by this approach is that we established a formal relationship between transaction semantics and operation semantics, so that the weakest operation

semantics can be identified that are sufficient in the context of the transaction semantics. This is an important result, because weaker operation semantics result in more concurrency.

In chapter 4 we extended this result to exploit semantics of *sets* of operations in the context of transaction trees in the tree model. We demonstrated that these extra semantics result in more concurrency for certain applications.

In chapter 5 we proposed a definition of decomposed semantics of transactions, where semantics are expressed in terms of multiple transitions on the application state. Each transition is assumed to transform a consistent database state into another consistent database state. This has motivated us to define correctness in terms of serializability with respect to transitions. As transitions are generally smaller than transactions, more schedules can be considered serializable and thus a higher degree of concurrency can be achieved.

The results on the improved concurrency are supported by the case study in chapter 7. We demonstrated improved concurrency by comparing the conventional state-based conflict relation with our semantic conflict relation on the operations of the application. We demonstrated that the semantic conflict relation is weaker than the conventional conflict relation, which translates in improved concurrency. For this particular application, we even achieved *maximum concurrency* in the context of the tree model. The concurrency is maximal, because the operations of the three transactions can be arbitrarily interleaved. In addition, we demonstrated that concurrency is improved by decomposing the semantics of the order transaction. It appeared that the refined semantics can be exploited only if one of the operations is decomposed in smaller operations.

For improving recovery, we have proposed the following semantics:

- compensatability and non-compensatability of operations,
- forward recovery semantics,
- rollback and durability semantics for transactions.

We started our discussion on recovery by extending the execution model with compensatable and non-compensatable operations. To guarantee atomicity after a non-compensatable operation, we proposed forward recovery.

Even though forward recovery is not new, the novel aspect of our recovery theory is that we improve concurrency by exploiting the *semantics* of forward recovery. For this, we assume that the set of operations that is required for forward recovery is known by the scheduler in case a non-compensatable operation is executed. In particular, the model supports *limited* knowledge of forward recovery, such that it is a generalization of models that support either no or full knowledge.

We have improved the flexibility of the model by allowing applications to use alternative forward recovery strategies and by allowing applications to make intermediate results durable. We support alternative forward recovery strategies by means of rollback operations. A rollback operation backward recoveres the uncommitted work of a transaction after which control can be returned to the application. This way, the application can make an attempt to complete the transaction.

Durability is improved in our model, because applications can make intermediate results durable by submitting a durability operation. After a durability operation has been submitted, the current results of a transaction are never backward recovered, even in the presence

of failures. Although its semantics are related to a classical savepoint operation, a savepoint does not protect intermediate results from failures. It appears that durability operations can be considered non-compensatable operations with null semantics. A desirable consequence is that durability operations can be seemlessly integrated in the model.

Finally, we considered the semantics of transactions and operations in the context of recovery. In particular, we established that the semantics of compensatable operations generally need to be made stronger to guarantee compensation. This may have a negative effect on the degree of concurrency. Fortunately, these stronger semantics are not required after compensatable operations have committed.

How can the transactional semantics be integrated into a single model?

As a general approach to the integration of the semantics into a single model, we adopted serializability theory, because this theory has proven itself as a sound basis for the development of concurrency theory with efficient protocols. In addition, conflict serializability theory has also proven to be extendible with recovery semantics, referred to as the unifying theory of concurrency and recovery.

With respect to the semantics for improving concurrency, we achieved an alignment of the semantics of transactions with conflict serializability theory by deriving relevant semantics of operations from the semantics of transactions. We introduced a conflict relation based on these weaker relevant operation semantics, defined as the *semantic* conflict relation. With this semantic conflict relation we have been able to extend conventional conflict serializability theory with application semantics. Additionally, in chapter 4 we have demonstrated that it is possible to extend the conflict serializability theory in the context of the tree model as well.

In chapter 5 we proposed a refinement of the definition of transactional semantics by distinguishing semantics of smaller transitions in a transaction. Importantly, we have been able to define a correctness criterion for transitions in line with conventional conflict serializability. For this, we introduced a definition for decomposed operation semantics. Each semantic element of an operation is associated with a transition, such that the union of these elements implement the corresponding transition. These elements can be used to define conflict serializability in the context of transitions.

With respect to the semantics for improving recovery, we have demonstrated that the unifying theory of concurrency and recovery can be extended with non-compensatable, rollback and durability operations. In addition, we demonstrated that the extended unifying theory of concurrency and recovery can be combined with the semantic conflict serializability theory. We discussed how the extended unifying theory of concurrency and recovery can be applied in the context of the tree model and in the context of semantically decomposed transactions.

What transaction management technique is suitable for scheduling?

The fact that we have developed our theory based on conflict serializability theory has the desirable property that efficient concurrency controls can be built. Indeed, we have been able to develop such efficient concurrency controls. In particular we succeeded in developing concurrency control protocols based on locking, graph testing and a combination of both.

For the case where no failures occur, we developed graph testing protocols for the flat model, tree model and the model with semantically decomposed transactions. To overcome the polynomial complexity of the graph testing protocols, we additionally proposed locking protocols, generally based on two-phase locking. In chapter 4 we discussed an interesting subclass of the tree model that can exploit the high-level semantics of prefixes of transactions. A remarkably simple locking protocol can be used to schedule this class, while still exploiting high-level semantics.

For the case where failures occur, we developed hybrid protocols along the lines of the protocols of the conventional unifying theory of concurrency and recovery. The hybrid protocols adopt two-phase locking to guarantee conflict serializability and graph testing for guaranteeing recoverability. The complication of the proposed protocols over those already proposed in literature, is that both backward recovery operations due to rollback operations, and forward recovery operations due to aborts must be considered as well. Finally, we have been able to establish protocols for the extended theory in combination with the tree model and model with semantically decomposed transactions.

8.2 Future work

In this thesis we have proposed a novel theory that exploits semantics of applications to improve concurrency and recovery. We have demonstrated that concurrency and recovery are indeed improved over existing models, because larger classes of schedules are accepted.

One important question that remains however, is *how* large the improvement is for particular applications. Of course, this question cannot be answered in general, because the answer is application dependent. Experiments have to be carried out to demonstrate the quantitative improvement for particular applications.

Even in case of performance improvement, the question remains whether the performance gain outweighs the additional complexity of program analysis. At this point we can mention an additional advantage of the model: as the model aligns with conventional conflict serializability theory, the semantics need only be analyzed for those parts of applications that are responsible for performance bottlenecks. Such analysis may prove that two operations that conflict according to the state-based interpretation, may not conflict when considering the application semantics.

Still, the complexity may impair practical application. We see two solutions to this problem. One solution is that the analysis of the semantics of applications are done in an informal way, based on the intuition of the application designer. The obvious disadvantage of this approach is that correctness is not guaranteed. Alternatively, a theorem prover can be used to derive sufficient semantics for operations, and derive the conflict relation for each pair of operations. Such formal approach would guarantee correctness of the conflict relation. The challenge however is to keep the problem computationally tractible. This is an interesting direction for future work.

Appendix A

Summary

Transaction management theory and its techniques have proven extremely successful for many applications to preserve correctness of a database in the presence of concurrency and failures. This is witnessed by their advance into modern database systems such as Oracle, IBM DB2 and Microsoft SQL Server. At the same time, it has been acknowledged that transaction management theory and its techniques are still insufficient for applications that involve long running transactions, because they result in considerable loss of throughput. As a result, novel applications such as electronic commerce applications and business process management applications cannot be adequatly supported.

This thesis therefore addresses the question how current transaction management theory can be improved such that these novel applications can be better supported by transaction management.

The approach taken in this thesis is to exploit more semantics of the particular application than in existing approaches. We exploit application semantics to improve concurrency and application semantics to improve recovery.

To improve concurrency we exploit the fact that certain transactions do not need to execute in full isolation, i.e. can tolerate some interference while still achieving their intended result. We formalize this approach by specifying *application-relevant* semantics in terms of a relation on application states. Such semantics are defined for operations, transactions and schedules. The main appeal of this approach, is that *sufficient* application-relevant operation semantics can be derived for each transaction. Sufficient operation semantics may be weaker than the actual semantics of operations, which potentially results in higher degrees of concurrency. We demonstrate that even when using conventional concurrency control techniques based on serialization graph testing and locking, weaker semantics result in higher degrees of concurrency than with conventional approaches.

In addition, we demonstrate that the approach can be generalized to the case where the semantics of subsets of operations of transactions are exploited to improve concurrency. This results in higher degrees of concurrency, because the semantics for a subset of operations may be generally weaker than the composition of the individual operation semantics.

Another direction to exploit application semantics is by refining the definition of transaction semantics. Instead of assuming a single result for each transaction, we assume that each transaction performs multiple smaller transitions that transform a consistent database state

into another consistent database state. Whereas in the conventional approach transactions must be serialized, in this approach only transitions need to be serialized. Because the unit of serialization is smaller, concurrency is improved.

To improve recovery, we extend the existing unifying theory of concurrency and recovery with forward recovery. Forward recovery is induced after an operation was executed that cannot be backward recovered. This means, that all results of a transaction before a non-compensatable operation are considered durable. We propose a specific case of such operation: the durability operation. The durability operation implies that the results of a transaction are made durable, without compromising isolation. This operation is particularly relevant for long running transactions, because expensive intermediate results can be protected from backward recovery.

In addition, we extend the execution model with a rollback operation. The rollback operation backward recovers all results of a transaction up to the last non-compensatable operation. From here, the application can retake control. An important property of the rollback operation is that it backward recovers both the database as well as the application data.

We demonstrate with our model that forward recoverability can be guaranteed, even if no knowledge is present about the forward recovery of a transaction. We also demonstrate that concurrency is improved if *more* semantics of forward recovery are known to the scheduler.

Further, we demonstrate how the recovery model can be integrated with the models for the concurrency control. In addition, we demonstrate that the application-relevant operation semantics can be too weak for backward recovery, and that in some cases stronger semantics are required. We develop hybrid protocols for the models based on locking and graph testing.

We validate our approach by applying our model to the TPC-W benchmark application. The TPC-W benchmark application represents a generic web-shop application and is representative for the novel applications targeted in this thesis. We demonstrate improvement of concurrency by comparing the conventional conflict relation with the semantic conflict relation, based on the semantics of the application. It turns out the semantic conflict relation is weaker, which is an indication of improved concurrency. Even, for this particular application, we achieved *maximal* concurrency when considering application semantics in one of our models. Although this cannot be expected for all applications, it demonstrates the potential improvement in concurrency by our approach to application semantics. In addition, we established improvements when considering the semantic decomposition of transactions.

We demonstrated improvement in recovery, by deriving a weaker conflict relation with respect to backward recovery operations. We further suggested improvements for the application by exploiting forward recovery.

We conclude that our approach can be expected to present similar results for different applications, because the TPC-W benchmark application is representative for many novel applications.

Appendix B

Samenvatting

Transactie management theorie en haar technieken zijn erg succesvol gebleken om de correctheid van een database te waarborgen in de context van concurrente toegang en fouten. Dit blijkt uit het feit dat zij zijn toegepast in moderne database systemen zoals Oracle, IBM DB2 en Microsoft SQL Server. Tegelijkertijd wordt onderkend dat transactie management theorie en haar technieken onvoldoende zijn voor applicaties met lang lopende transacties, omdat zij resulteren in een aanmerkelijk verlies in doorloopsnelheid. Als gevolg hiervan kunnen moderne applicaties zoals electronic commerce en bedrijfsproces ondersteunende applicaties onvoldoende worden ondersteund.

Dit proefschrift behandelt de vraag hoe transactie management theorie kan worden verbeterd, opdat deze nieuwe applicaties beter kunnen worden ondersteund door transactie management.

Om dit te bereiken, volgen we een aanpak waarbij meer semantiek van applicaties wordt gebruikt dan in bestaande aanpakken. We gebruiken applicatiesemantiek om concurrente toegang tot data en herstelprocedures na systeemfouten te verbeteren.

Om concurrente toegang tot data te verbeteren gebruiken we de wetenschap dat bepaalde transacties niet volledige isolatie vereisen, d.i. enige interferentie is mogelijk door andere transacties terwijl het bedoelde eindresultaat van de transactie toch wordt gehaald. We formalizeren deze aanpak door *applicatie-relevante* semantiek te definieren in termen van een relatie op applicatie toestanden. Deze semantiek is gedefinieerd voor operaties, transacties en executies. Het aantrekkelijke aspect van deze benadering is dat *voldoende* applicatierelevante operatie semantiek kan worden afgeleid voor iedere transactie. Voldoende applicatiesemantiek kan zwakker zijn dan de eigenlijke semantiek van operaties, zodat een potentieel hogere mate van concurrente toegang kan worden verwacht. We tonen aan dat zelfs met conventionele technieken, gebaseerd op graaf testen en het zetten van locks, zwakkere semantiek resulteert in een hogere mate van concurrente toegang dan de traditionele benadering.

Daarbij tonen we aan dat deze aanpak kan worden gegeneralizeerd naar groepen van operaties. Dit resulteert in hogere graad van concurrente toegang, omdat de semantiek van een groep van operaties in het algemeen zwakker is dan de compositie van de individuele semantiek van de operaties.

Een andere manier om applicatie semantiek te gebruiken is door de definitie van trans-

actiesemantiek te verfijnen. In plaats van één resultaat per transactie nemen we nu aan dat een transactie meerdere kleinere transities uitvoert die een consistente database toestant transformeren in een andere consistente database toestant. Terwijl in de conventionele benadering transacties moeten worden geserializeerd, hoeven nu slechts de transities worden geserializeerd. Omdat de eenheid van serializatie kleiner is, wordt de graad van concurrente toegang verbeterd.

Om de herstelprocedures na een systeemfout te verbeteren, breiden we bestaande theorie uit met *voorwaardse* herstelprocedures. De voorwaardse herstelprocedure wordt gebruikt indien een operatie is uitgevoerd die niet kan worden hersteld door een compenserende operatie uit te voeren. Dit betekent dat alle resultaten van een transactie voor een niet-compenseerbare operatie worden beschouwd als duurzaam. We introduceren een specifiek geval van een dergelijke operatie: de duurzame operatie. De duurzame operatie impliceert dat de resultaten van een transactie duurzaam worden gemaakt, zonder dat isolatie van een transactie wordt opgeheven. Deze operatie is in het bijzonder van belang voor lang lopende transacties, omdat dure tussenresultaten kunnen worden beschermd tegen achterwaardse herstelprocedures.

Daarbij breiden we het executiemodel uit met een terugrol operatie. De terugrol operatie herstelt de resultaten van een transactie tot de laatste niet-compenseerbare operatie. Vanaf dat punt kan de applicatie de controle overnemen. Een belangrijke eigenschap van de terugrol operatie is dat zowel de database als ook de applicatie data achterwaards wordt hersteld.

We tonen aan dat met ons model voorwaards herstel kan worden gegarandeerd, zelfs als geen kennis aanwezig is van de voorwaardse herstelprocedure. Ook tonen we aan dat concurrente toegang verbetert als *meer* semantiek van de voorwaardse herstelprocedure bekend is.

Daarbij tonen we aan hoe het herstelmodel kan worden geïntegreerd met de modellen voor de concurrente toegangscontrole. Ook tonen we aan dat applicatie-relevante operatiesemantiek te zwak kan zijn voor een achterwaardse herstelprocedure, en dat sterkere semantiek nodig kan zijn. We ontwikkelen hybride protocollen voor de modellen, gebaseerd op het zetten van locks en het testen van grafen.

We valideren onze aanpak door ons model toe te passen op de TPC-W benchmark applicatie. De TPC-W benchmark applicatie representeert een generieke web-shop applicatie en is representatief voor de nieuwe applicaties waarop dit proefschrift zich richt. We tonen verbetering van concurrente toegang aan door de conventionele conflict relatie te vergelijken met de semantische conflict relatie, die is gebaseerd op de semantiek van de applicatie. Het blijkt dat de semantische conflict relatie zwakker is, hetgeen wijst op verbeterde concurrente toegang. We hebben voor deze toepassing zelfs *maximale* concurrente toegang bereikt voor een van onze modellen. Hoewel dit niet kan worden verwacht voor alle applicaties, toont het wel de potentiele verbetering aan van onze aanpak met applicatie semantiek. Ook hebben we verbeteringen aangetoond middels de semantische decompositie van transacties.

We tonen verbetering van herstelprocedures aan door een zwakkere conflict relatie af te leiden voor compenserende operaties. Ook geven we suggesties voor verbeteringen voor de applicatie door voorwaardse herstelprocedures te gebruiken.

We besluiten dat het verwacht mag worden dat onze aanpak dezelfde resultaten zal opleveren voor andere applicaties, omdat de TPC-W benchmark applicatie representatief is voor veel nieuwe applicaties.

Bibliography

[1] Divyakant Agrawal, John L. Bruno, Amr El Abbadi, and Vashudha Krishnaswamy. Relative serializability (extended abstract): an approach for relaxing the atomicity of transactions. In *PODS '94: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 139–149, New York, NY, USA, 1994. ACM Press.

[2] Divyakant Agrawal, Amr El Abbadi, and Ambuj K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, 1993.

[3] Gustavo Alonso, Stephen Blott, Armin Fessler, and Hans-Joerg Schek. Correctness and parallelism in composite systems. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 197–208, New York, NY, USA, 1997. ACM Press.

[4] Gustavo Alonso, Armin Fessler, Guy Pardon, and Hans-Joerg Schek. Correctness in general configurations of transactional components. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 285–293, New York, NY, USA, 1999. ACM Press.

[5] Gustavo Alonso, Armin Fessler, Guy Pardon, and Hans-Joerg Schek. Transactions in stack, fork, and join composite systems. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 150–168, London, UK, 1999. Springer-Verlag.

[6] Gustavo Alonso, Radek Vingralek, Divyakant Agrawal, Yuri Breitbart, Amr El Abbadi, Hans-J. Schek, and Gerhard Weikum. Unifying concurrency control and recovery of transactions. *Inf. Syst.*, 19(1):101–115, 1994.

[7] Paul Amman, Sushil Jajodia, and Indrakshi Ray. *Semantic-Based Decomposition of Transactions*. 1997.

[8] ANSI. Ansi x3.135-1992, american national standard for information systems - database language - sql. 1992.

[9] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A model for concurrency in nested transactions systems. *J. ACM*, 36(2):230–269, 1989.

[10] Catriel Beeri, Philip A. Bernstein, Nathan Goodman, Ming-Yee Lai, and Dennis E. Shasha. A concurrency control theory for nested transactions (preliminary report). In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 45–62, New York, NY, USA, 1983. ACM Press.

[11] Catriel Beeri, Philip A. Bernstein, Nathan Goodman, Ming-Yee Lai, and Dennis E. Shasha. A concurrency control theory for nested transactions (preliminary report). In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 45–62, New York, NY, USA, 1983. ACM Press.

[12] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[13] Arthur J. Bernstein, David S. Gerstl, and Philip M. Lewis. Concurrency control for step-decomposed transactions. *Inf. Syst.*, 24(9):673–698, 1999.

[14] Arthur J. Bernstein, David Scott Gerstl, Wai-Hong Leung, and Philip M. Lewis. Design and performance of an assertional concurrency control system. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 436–445, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Arthur J. Bernstein, David Scott Gerstl, Philip M. Lewis, and S. Lu. Using transaction semantics to increase performance. *International Workshop on High Performance Transaction Systems*, pages 26–29, 1999.

[16] Arthur J. Bernstein and Philip M. Lewis. *Concurrency in programming and database systems*. Jones and Bartlett Publishers, Inc., , USA, 1993.

[17] Arthur J. Bernstein and Philip M. Lewis. Transaction decomposition using transaction semantics. *Distrib. Parallel Databases*, 4(1):25–47, 1996.

[18] Arthur J. Bernstein, Philip M. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, pages 57–66, Washington, DC, USA, 2000. IEEE Computer Society.

[19] Philip A. Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The asilomar report on database research. *SIGMOD Rec.*, 27(4):74–80, 1998.

[20] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control-theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[21] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[22] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering SE-5*, 3:203–216, 1979.

[23] Panos K. Chrysanthis and Krithi Ramamritham. Acta: the saga continues. pages 349–397, 1992.

[24] Transaction Processing Council. "tpc benchmark w (web commerce)" - specification version 1.8 - 19 february 2002, 2002.

[25] Juliane Dehnert and Wijnand Derks. Ein pragmatischer korrektheitsbegriff fuer die modellierung von geschaeftsprozessen mit petrinetzen. *Visuelle Verhaltensmodellierung verteilter und nebenlaeufiger Software-Systeme*, 24/00-I:51–56, 2000.

[26] Juliane Dehnert and Wijnand Derks. Relaxed soundness - ein pragmatischer korrektheitsbegriff fuer die geschaeftsprozess-modellierung. *Algorithmen und Werkzeuge fuer Petrinetze, AWPN 2000, Fachberichte Informatik*, pages 63–68, 2000.

[27] Wijnand Derks, Zef Damen, Matthijs Duitshof, and Henk Ensing. Business-to-business e-commerce in a logistics domain. *Proceedings of the CAISE*00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing (IDSO'00), Stockholm June 5-6*, 2000.

[28] Wijnand Derks, Juliane Dehnert, Paul Grefen, and Willem Jonker. Customized atomicity specification for transactional workflows. *CTIT Technical report*, 00-24, 2001.

[29] Wijnand Derks, Juliane Dehnert, Paul Grefen, and Willem Jonker. Customized atomicity specification for transactional workflows. In *CODAS '01: Proceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications*, page 140, Washington, DC, USA, 2001. IEEE Computer Society.

[30] Wijnand Derks, Sietse Dijkstra, Henk Enting, Willem Jonker, and Jeroen Wijnands. Experimenting numa for scaleable cdr processing. *Proceedings of the 11th International Conference on Database and Expert Systems Applications, Lecture Notes in Computer Science 1873*, 2000.

[31] Wijnand Derks, Sietse Dijkstra, Jeroen Wijnands, and Willem Jonker. Assessment of scaleable database architectures for cdr analysis - an experimental approach. *LNCS Proceedings Workshop Databases in Telecommunications*, 2000.

[32] Wijnand Derks and Paul Grefen. Requirements and architecture for automatic setup and enactment of reliable cooperative services. Technical Report TR-CTIT-02-26, CTIT, February 2002.

[33] Wijnand Derks and Willem Jonker. Practical experiences with materialized views in a very large data store for telecommunication service management. *Proceedings of the DEXA Workshop*, pages 881–886, 1998.

[34] Ahmed K. Elmagarmid, Yungho Leu, Witold Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–518, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[35] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

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

[37] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[38] 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, New York, NY, USA, 1987. ACM Press.

[39] Dimitrios Georgakopoulos and Mark F. Hornick. A framework for enforceable specification of extended transaction models and transaction workflows. *International Journal of Cooperative Information Systems*, 3(3):599–617, 1994.

[40] Dimitrios Georgakopoulos, Mark F. Hornick, and Frank Manola. Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):630–649, 1996.

[41] Jim Gray. The transaction concept: virtues and limitations. pages 140–150, 1988.

[42] Jim Gray, P. Homan, Henry F. Korth, and Ron Obermarck. A straw man analysis of the probability of waiting and deadlock in a database system. *Berkeley Workshop*, page 125, 1981.

[43] Paul Grefen, Karl Aberer, Heiko Ludwig, and Yigal Hoffner. Crossflow: Cross-organizational workflow management for service outsourcing in dynamic virtual enterprises. *IEEE Data Engineering Bulletin*, 24(1):52–57, 2001.

[44] Paul Grefen, Barbara Pernici, and Gabriel Sanchez, editors. *Database Support for Workflow Management: The Wide Project*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[45] Vassos Hadzilacos. An operational model for database system reliability. In *PODS '83: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 244–257, New York, NY, USA, 1983. ACM Press.

[46] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[47] Theo Haerder and Kurt Rothermel. Concepts for transaction recovery in nested transactions. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 239–248, New York, NY, USA, 1987. ACM Press.

[48] Willem Jonker, Wim Nijenhuis, Zef Damen, Martin Verwijmeren, and Wijnand Derks. Workflow management systems and cross-organisational logistics. *Workshop on Cross-Organisational Workflow Management and Co-ordination*, 1999.

[49] Henry K. Korth and Gregory D. Speegle. Formal model of correctness without serializabilty. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 379–386, New York, NY, USA, 1988. ACM Press.

[50] Narayanan Krishmakumar and Amit Sheth. Specifying multi-system workflow applications in meteor. *Comp. Sc. Tech. Rep. TR-CS-02*, 1994.

[51] Vasudev Krishnamoorthy and Ming-Chien Shan. Virtual transaction model to support workflow applications. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 876–881, New York, NY, USA, 2000. ACM Press.

[52] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. A theory of relaxed atomicity (extended abstract). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 95–110, New York, NY, USA, 1991. ACM Press.

[53] Philip Lewis, Arthur J. Bernstein, and Michael Kifer. Database transaction processing - an application oriented approach. 2002.

[54] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[55] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[56] Nancy A. Lynch. Multilevel atomicity - a new correctness criterion for database concurrency control. *ACM Trans. Database Syst.*, 8(4):484–502, 1983.

[57] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. Relaxing serializability in multidatabase systems. *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 205–212, 1992.

[58] Sharad Mehrotra, Rajeev Rastogi, Abraham Silberschatz, and Henry F. Korth. A transaction model for multi-database systems. *Proceedings of the international conference on Distributed Computing Systems*, pages 56–63, 1992.

[59] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[60] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

[61] J. Eliot B. Moss. *Nested transactions: An Approach to Reliable Distributed Computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[62] J. Eliot B. Moss. Nested transactions: an introduction. pages 395–425, 1987.

[63] Christos Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., New York, NY, USA, 1986.

[64] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[65] Guy Pardon and Gustavo Alonso. Cheetah: a lightweight transaction server for plug-and-play internet data management. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 210–219, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[66] Andreas Reuter. Contracts: A means for extending control beyond transaction boundaries. *Proceedings of the 3rd International Workshop on High Performance Transaction Systems*, 1989.

[67] Andreas Reuter, Kerstin Schneider, and Friedemann Schwenkreis. Contracts revisited. *Advanced Transaction Models and Architectures*, pages 127–151, 1997.

[68] Kurt Rothermel and C. Mohan. Aries/nt: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the fifteenth international conference on Very Large Databases*, pages 337–346, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[69] Hans-Joerg Schek, Gerhard Weikum, and Haiyan Ye. Towards a unified theory of concurrency control and recovery. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 300–311, New York, NY, USA, 1993. ACM Press.

[70] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Joerg Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27(1):63–116, 2002.

[71] Friedemann Schwenkreis. Apricots - a prototype implementation of a contract system - management of the control flow and the communication system. *Proceedings of the Symposium on Reliable Distributed Systems*, pages 12–21, 1993.

[72] Friedemann Schwenkreis and Andreas Reuter. The impact of concurrency control on the programming model of contracts. *International Workshop on Advanced Transaction Models and Architectures (ATMA)*, 1996.

[73] Richard E. Stearns and Daniel J. Rosenkrantz. Distributed database concurrency controls using before-values. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 74–83, New York, NY, USA, 1981. ACM Press.

[74] Jari Veijalainen. *Transaction Concepts in Autonomous Database Environments*. PhD thesis, Munich, 1990.

[75] K. Vidyasankar. Generalized relative serializability. *COMAD98*, 1998.

[76] Radek Vingralek, Haiyan Hasse-Ye, Yuri Breitbart, and Hans-Joerg Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Theor. Comput. Sci.*, 190(2):363–396, 1998.

[77] Radek Vingralek, Haiyan Ye, Yuri Breitbart, and Hans-Joerg Schek. Unified transaction model for semantically rich operations. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 148–161, London, UK, 1995. Springer-Verlag.

[78] Jochem Vonk, Wijnand Derks, Paul Grefen, and Marjanka Koetsier. Cross-organisational transaction support for virtual enterprises. Eilat, 2000.

[79] Helmut Waechter and Andreas Reuter. The contract model. *Database Transaction Models, chapter 7*, pages 219–263, 1992.

[80] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, 1988.

[81] Gerhard Weikum. *Transaction Management in Database Systems with Layered Architectures*. PhD thesis, 1987.

[82] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.

[83] Gerhard Weikum and Hans-Joerg Schek. Architectural issues of transaction management in multi-layered systems. In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 454–465, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[84] Gerhard Weikum and Hans-Jorg Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.

[85] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[86] Jeroen Wijnands, Wijnand Derks, Sietse Dijkstra, and Willem Jonker. Experimenting cluster technology for massive cdr processing. *Databases in Telecommunications II, VLDB 2001 International Workshop DBTel, Lecture Notes in Computer Science 2209*, 2001.

[87] Mihalis Yannakakis. Serializability by locking. *J. ACM*, 31(2):227–244, 1984.

[88] Aidong Zhang, Marian Nodine, and Bharat Bhargava. Global scheduling for flexible transactions in heterogeneous distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 13(3):439–450, 2001.

[89] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 67–78, New York, NY, USA, 1994. ACM Press.

SIKS Dissertation Series

[1998-1] Johan van den Akker (CWI), *DEGAS - An Active, Temporal Database of Autonomous Objects*

[1998-2] Floris Wiesman (UM), *Information Retrieval by Graphically Browsing Meta-Information*

[1998-3] Ans Steuten (TUD), *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

[1998-4] Dennis Breuker (UM), *Memory versus Search in Games*

[1998-5] E.W.Oskamp (RUL), *Computerondersteuning bij Straftoemeting*

[1999-1] Mark Sloof (VU), *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products*

[1999-2] Rob Potharst (EUR), *Classification using decision trees and neural nets*

[1999-3] Don Beal (UM), *The Nature of Minimax Search*

[1999-4] Jacques Penders (UM), *The practical Art of Moving Physical Objects*

[1999-5] Aldo de Moor (KUB), *Empowering Communities: A Method for the Legitimate User-Driven Specification of* Network Information Systems

[1999-6] Niek J.E. Wijngaards (VU), *Re-design of compositional systems*

[1999-7] David Spelt (UT), *Verification support for object database design*

[1999-8] Jacques H.J. Lenting (UM), *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete* Reallocation.

[2000-1] Frank Niessink (VU), *Perspectives on Improving Software Maintenance*

[2000-2] Koen Holtman (TUE), *Prototyping of CMS Storage Management*

[2000-3] Carolien M.T. Metselaar (UvA), *Sociaal-organisatorische gevolgen van kennistechnologie; een* procesbenadering en actorperspectief.

[2000-4] Geert de Haan (VU), *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

[2000-5] Ruud van der Pol (UM), *Knowledge-based Query Formulation in Information Retrieval*

[2000-6] Rogier van Eijk (UU), *Programming Languages for Agent Communication*

[2000-7] Niels Peek (UU), *Decision-theoretic Planning of Clinical Patient Management*

[2000-8] Veerle Coup (EUR), *Sensitivity Analyis of Decision-Theoretic Networks*

[2000-9] Florian Waas (CWI), *Principles of Probabilistic Query Optimization*

[2000-10] Niels Nes (CWI), *Image Database Management System Design Considerations, Algorithms and Architecture*

[2000-11] Jonas Karlsson (CWI), *Scalable Distributed Data Structures for Database Management*

[2001-1] Silja Renooij (UU), *Qualitative Approaches to Quantifying Probabilistic Networks*

[2001-2] Koen Hindriks (UU), *Agent Programming Languages: Programming with Mental Models*

[2001-3] Maarten van Someren (UvA), *Learning as problem solving*

[2001-4] Evgueni Smirnov (UM), *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

[2001-5] Jacco van Ossenbruggen (VU), *Processing Structured Hypermedia: A Matter of Style*

[2001-6] Martijn van Welie (VU), *Task-based User Interface Design*

[2001-7] Bastiaan Schonhage (VU), *Diva: Architectural Perspectives on Information Visualization*

[2001-8] Pascal van Eck (VU), *A Compositional Semantic Structure for Multi-Agent Systems Dynamics.*

[2001-9] Pieter Jan 't Hoen (RUL), *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

[2001-10] Maarten Sierhuis (UvA), *Modeling and Simulating Work Practice BRAHMS: a multi-agent modeling and simulation language for work practice analysis and design*

[2001-11] Tom M. van Engers (VUA), *Knowledge Management: The Role of Mental Models in Business Systems Design*

[2002-01] Nico Lassing (VU), *Architecture-Level Modifiability Analysis*

[2002-02] Roelof van Zwol (UT), *Modelling and searching web-based document collections*

[2002-03] Henk Ernst Blok (UT), *Database Optimization Aspects for Information Retrieval*

[2002-04] Juan Roberto Castelo Valdueza (UU), *The Discrete Acyclic Digraph Markov Model in Data Mining*

[2002-05] Radu Serban (VU), *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*

[2002-06] Laurens Mommers (UL), *Applied legal epistemology; Building a knowledge-based ontology of the legal domain*

[2002-07] Peter Boncz (CWI), *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

[2002-08] Jaap Gordijn (VU), *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

[2002-09] Willem-Jan van den Heuvel(KUB), *Integrating Modern Business Applications with Objectified Legacy Systems*

[2002-10] Brian Sheppard (UM), *Towards Perfect Play of Scrabble*

[2002-11] Wouter C.A. Wijngaards (VU), *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

[2002-12] Albrecht Schmidt (UvA), *Processing XML in Database Systems*

[2002-13] Hongjing Wu (TUE), *A Reference Architecture for Adaptive Hypermedia Applications*

[2002-14] Wieke de Vries (UU), *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-*Agent Systems

[2002-15] Rik Eshuis (UT), *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

[2002-16] Pieter van Langen (VU), *The Anatomy of Design: Foundations, Models and Applications*

[2002-17] Stefan Manegold (UvA), *Understanding, Modeling, and Improving Main-Memory Database Performance*

[2003-01] Heiner Stuckenschmidt (VU), *Ontology-Based Information Sharing in Weakly Structured Environments*

[2003-02] Jan Broersen (VU), *Modal Action Logics for Reasoning About Reactive Systems*

[2003-03] Martijn Schuemie (TUD), *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

[2003-04] Milan Petkovic (UT), *Content-Based Video Retrieval Supported by Database Technology*

[2003-05] Jos Lehmann (UvA), *Causation in Artificial Intelligence and Law - A modelling approach*

[2003-06] Boris van Schooten (UT), *Development and specification of virtual environments*

[2003-07] Machiel Jansen (UvA), *Formal Explorations of Knowledge Intensive Tasks*

[2003-08] Yongping Ran (UM), *Repair Based Scheduling*

[2003-09] Rens Kortmann (UM), *The resolution of visually guided behaviour*

[2003-10] Andreas Lincke (UvT), *Electronic Business Negotiation: Some experimental studies on the interaction* between medium, innovation context and culture

[2003-11] Simon Keizer (UT), *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

[2003-12] Roeland Ordelman (UT), *Dutch speech recognition in multimedia information retrieval*

[2003-13] Jeroen Donkers (UM), *Nosce Hostem - Searching with Opponent Models*

[2003-14] Stijn Hoppenbrouwers (KUN), *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

[2003-15] Mathijs de Weerdt (TUD), *Plan Merging in Multi-Agent Systems*

[2003-16] Menzo Windhouwer (CWI), *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses*

[2003-17] David Jansen (UT), *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

[2003-18] Levente Kocsis (UM), *Learning Search Decisions*

[2004-01] Virginia Dignum (UU), *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

[2004-02] Lai Xu (UvT), *Monitoring Multi-party Contracts for E-business*

[2004-03] Perry Groot (VU), *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*

[2004-04] Chris van Aart (UvA), *Organizational Principles for Multi-Agent Architectures*

[2004-05] Viara Popova (EUR), *Knowledge discovery and monotonicity*

[2004-06] Bart-Jan Hommes (TUD), *The Evaluation of Business Process Modeling Techniques*

[2004-07] Elise Boltjes (UM), *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes*

[2004-08] Joop Verbeek (UM), *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politile gegevensuitwisseling en digitale expertise*

[2004-09] Martin Caminada (VU), *For the Sake of the Argument; explorations into argument-based reasoning*

[2004-10] Suzanne Kabel (UvA), *Knowledge-rich indexing of learning-objects*

[2004-11] Michel Klein (VU), *Change Management for Distributed Ontologies*

[2004-12] The Duy Bui (UT), *Creating emotions and facial expressions for embodied agents*

[2004-13] Wojciech Jamroga (UT), *Using Multiple Models of Reality: On Agents who Know how to Play*

[2004-14] Paul Harrenstein (UU), *Logic in Conflict. Logical Explorations in Strategic Equilibrium*

[2004-15] Arno Knobbe (UU), *Multi-Relational Data Mining*

[2004-16] Federico Divina (VU), *Hybrid Genetic Relational Search for Inductive Learning*

[2004-17] Mark Winands (UM), *Informed Search in Complex Games*

[2004-18] Vania Bessa Machado (UvA), *Supporting the Construction of Qualitative Knowledge Models*

[2004-19] Thijs Westerveld (UT), *Using generative probabilistic models for multimedia retrieval*

[2004-20] Madelon Evers (Nyenrode), *Learning from Design: facilitating multidisciplinary design teams*

[2005-01] Floor Verdenius (UvA), *Methodological Aspects of Designing Induction-Based Applications*

[2005-02] Erik van der Werf (UM), *AI techniques for the game of Go*

[2005-03] Franc Grootjen (RUN), *A Pragmatic Approach to the Conceptualisation of Language*

[2005-04] Nirvana Meratnia (UT), *Towards Database Support for Moving Object data*

[2005-05] Gabriel Infante-Lopez (UvA), *Two-Level Probabilistic Grammars for Natural Language Parsing*

[2005-06] Pieter Spronck (UM), *Adaptive Game AI*

[2005-07] Flavius Frasincar (TUE), *Hypermedia Presentation Generation for Semantic Web Information Systems*

[2005-08] Richard Vdovjak (TUE), *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

[2005-09] Jeen Broekstra (VU), *Storage, Querying and Inferencing for Semantic Web Languages*

[2005-10] Anders Bouwer (UvA), *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

[2005-11] Elth Ogston (VU), *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

[2005-12] Csaba Boer (EUR), *Distributed Simulation in Industry*

[2005-13] Fred Hamburg (UL), *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

[2005-14] Borys Omelayenko (VU), *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

[2005-15] Tibor Bosse (VU), *Analysis of the Dynamics of Cognitive Processes*

[2005-16] Joris Graaumans (UU), *Usability of XML Query Languages*

[2005-17] Boris Shishkov (TUD), *Software Specification Based on Re-usable Business Components*

[2005-18] Danielle Sent (UU), *Test-selection strategies for probabilistic networks*

[2005-19] Michel van Dartel (UM), *Situated Representation*

[2005-20] Cristina Coteanu (UL), *Cyber Consumer Law, State of the Art and Perspectives*