Department of Informatics, University of Zürich

BSc Thesis

Simultaneous Execution of Single- and Multi-Version Protocols with the Oshiya Debugger and Analyzer

Luis Schüller

Matrikelnummer: 10-734-085

Email: luis.schueller@uzh.ch

July 2, 2012

supervised by Prof. Dr. Michael H. Böhlen and Christian Tilgner

University of Zurich

Department of Informatics

Abstract

The Oshiya Debugger and Analyzer (ODA) is a tool for debugging, visualizing and comparing concurrency control techniques. Prior to this work ODA was limited to execute one single-version protocol concurrently. The goal of this thesis was to extend ODA, in order to support multi-version protocols as well as simultaneous protocol execution. Simultaneous protocol execution allows the user direct comparison of different protocols with regard to performance- and correctness-criteria. ODA simulates the execution of protocols over user-provided workloads. Additional features for the workload are introduced, to model sophisticated client behavior.

Zusammenfassung

Der 'Oshiya Debugger and Analyzer' (ODA) ist eine Applikation, die es erlaubt Datenbank-Protokolle zu analysieren und zu debuggen. Ursprünglich ermöglichte ODA Single-Versions Protkolle einzeln auszuführen. Ziel dieser Thesis war es Multiversions-Protokolle innerhalb der Applikation zu unterstützen, sowie mehrere Protokolle gleichzeitig auszuführen. Das simultane Ausführen mehere Protokolle ermöglicht dem Benutzer direkte Protokollvergleiche in Hinsicht auf Performanz und Korrektheit. Ausserdem werden zusätzliche Features vorgestellt, mit Hilfe derer realistisches Verhalten von Clients auf der Datenbank simuliert werden kann.

Contents

List of Figures

List of Tables

1 Introduction

The **O**shiya **D**ebugger and **A**nalyzer (ODA) is a tool for developing, visualizing and comparing Oshiya scheduling protocol implementations. ODA simulates the execution of protocols over user provided workloads, to give its users (database developers and researchers) insights into behavior and characteristics of scheduling protocols. ODA uses a declarative scheduling model, the so called **O**shiya **s**cheduling **m**odel (OSM), that performs scheduling iteratively in so-called *scheduling iterations*, which will be further described in Section 2.1. Hereby ODA facilitates the following key features:

Interactive Protocol Comparison ODA provides the possibility to execute multiple protocols simultaneously over the same workload. This feature is used to compare protocols, e.g., the strict Two-Phase Locking (SS2PL) and Snapshot Isolation (SI) protocols [TGB+11]. Interactive protocol comparison is the feature, the contributions of this thesis mainly focus on.

Navigational Debugging With navigational debugging ODA offers the user the feature to debug through the scheduling process backward and forward. The application shows the scheduling- and database state at each step [TGB+11].

Break and Analyze Queries ODA models breakpoints as break queries that are executed after each scheduler iteration. Scheduling stops when a break query returns a non-empty result which may be occurred through a constraint violation affected by a scheduling protocol. Analyze queries are used to identify matching tuples. The combination of break and analyze queries allows to easily detect and analyze errors in protocol executions and provides an understanding of how a protocol behaves for a certain workload [TGB+11].

Statistical Protocol Analysis ODA provides statistics about protocol executions, and it allows users to register new measures and customize how they are displayed, e.g., as a graph or as tabular data. Custom measures are modeled as statistic queries that access the scheduling state, database state, and results of break or analyze queries. ODA collects, aggregates, and visualizes the results of these queries over time [TGB+11].

1.1 Limitations Prior to this Work

The following limitations of ODA existed prior to this work.

Limited to Single-version Protocols Prior to this work ODA was limited to the execution of single-version protocols, e.g., SS2PL, because of technical limitations of the OSM within ODA and the data relation the OSM accesses on. The data relation did not allow to hold multiple versions of an item, as well as the OSM that could not process requests accessing a certain version of an item. Providing the user a wider spectrum of concurrency control techniques, e.g., multi-version protocols, enhances the application in regard to the main goal of ODA, which is developing, visualizing and comparing different scheduling protocols.

ODA Only Executes One Protocol Concurrently ODA provided the possibility to execute one protocol over a given workload concurrently. Thus, comparing scheduling protocols in ODA, e.g., with respect to correctness- and performance criteria, could only be done successively by executing multiple protocols and storing the produced data for a later comparison. This is very inconvenient for the user and was improved throughout this work.

Limitations in the Simulation of Client Behavior In order to analyze protocols it is fundamental to provide the user the possibility to simulate realistic scenarios for her database. Client behavior simulation is a complex problem that is very specific according to the system the user wants to simulate within ODA. The currently existing features are not sufficient for many scenarios to model realistic behavior patterns. E.g., ODA provides the user the possibility to consign read values to a write request, so that the value written is influenced by current state of the data relation. To model behavior, e.g., customers of a shopping system, it is not satisfying to just process the workload linearly, but to give the user the possibility to define special constraints that influence the workload at runtime. E.g., constraints defining a minimum quantity for an order, otherwise the requests transacting the order will be skipped.

1.2 Contributions

The following work describes three major enhancements to the application, that were developed throughout this bachelor thesis. The task of the thesis was to enable the support of multi-version protocols, and the simultaneous execution of multiple protocols in ODA. Parallel to these enhancements new requirements for the workload rose, so additional features to model a sophisticated client simulation were implemented.

Support of Multi-version Protocols ODA lets its potential users, e.g., database developers, investigate protocols with regard to correctness criteria, service-level agreements and performance requirements. Supporting multi-version protocols within ODA opens up a whole new spectrum of protocols to the user that are widespread in modern database systems. This contribution allows the development of application-specific multi-version protocols as a well as protocol comparisons that include multi-version protocols, as e.g., SS2PL and SI. In this work multi-version protocols are enabled through the following three adaptations:

- Enhancing the OSM to process requests accessing a special version of an item
- Enhancing the data relation, the OSM accesses on, to hold multiple tuples of an item
- Implementing a new *Executor*, the functional part of the application executing the scheduled requests on the data relation, for multi-version protocols

Simultaneous Protocol Execution The simultaneous execution of multiple protocols was not supported by ODA. Selecting multiple protocols, e.g., a multi- and single-version protocol, as well as a predefined workload simulating a scenario, enables protocol comparison to the user in way that to the best of our knowledge does not exist yet [TGB$^+$11]. ODA simultaneously executes these protocols on the workload and creates one schedule per protocol. These schedules can be analyzed by the application, e.g., in respect of correctness and performance. This allows direct comparison of schedules produced by different protocols.

Sophisticated Workloads A workload is the set of requests that will be scheduled by the protocols selected in ODA. It simulates the clients behavior on a database in terms of incoming requests. The simulation of realistic scenarios is fundamental for ODA, since it is the basis for all other functionalities, like the investigation of protocol behavior, or statistical analysis of different protocols. The application was enlarged by applying the following new features to a workload:

- **Conditional write-requests:** The value of write requests can change according to a predefined constraint.
- **Conditional read-requests:** According to a constraint defined in a read request several requests within a transaction can be skipped.
- **Delay Functionality:** Delays between request with a transaction can be molded within the workload.

This thesis is structured as following: In Chapter 2 fundamentals concerning the OSM and the application architecture of ODA will be explained to get an understanding of how ODA processes scheduling. Moreover, a brief introduction in the properties of single-version and multi-version protocols will be given. In Chapter 3 the concepts and implementation enabling multi-version protocols in ODA will be discussed. Chapter 4 goes into detail about the challenges and implementation of executing multiple protocols in ODA simultaneously. Chapter 5 introduces the workload functionalities prior to this work, as well as new functionalities that have been developed throughout this thesis.

2 Preliminaries

This chapter briefly describes the Oshiya scheduling model (OSM) that is used by ODA to process scheduling in Section 2.1. In Section 2.2 a rough overview of the architecture of ODA is given, since it will be needed in the ongoing paper. In Section 2.3 the differences of single-version and multi-version protocols are outlined, to better understand the enabling of multi-version protocols within the application explained in the next Chapter 3.

2.1 Oshiya Scheduling Model

The **O**shiya declarative **s**cheduling **m**odel (OSM) is a new approach to develop scheduling protocols in a declarative manner. Requests are modeled as tuples in so-called *scheduling relations*. The OSM maintains three of these scheduling relations, $\mathcal{R}, \mathcal{E}, \mathcal{H}$. $\mathcal{R}$ (pending requests) stores all the incoming requests waiting to be executed, $\mathcal{E}$ (executable requests) the scheduled requests to be executed, and $\mathcal{H}$ (relevant history) stores the scheduled requests as a relevant history. The term 'relevant' indicates that $\mathcal{H}$ does not store the whole history, but only these requests that might be relevant for future scheduling. A seven-step algorithm as shown in Fig 2.1 is iteratively processed in Oshiya indicated by the surrounding while-loop. A protocol is specified through the implementation of so-called *scheduling queries*. These scheduling queries are performed on the scheduling relations to forward the scheduling process [TGBK11]. For better understanding the seven steps of the algorithm will be briefly explained.

```
  $\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
  while true do begin
1   $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2   $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3
    $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4   $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5   $Execute(\mathcal{E})$;
6   $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7
    $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;
  end
```

Figure 2.1: Oshiya Scheduling Algorithm

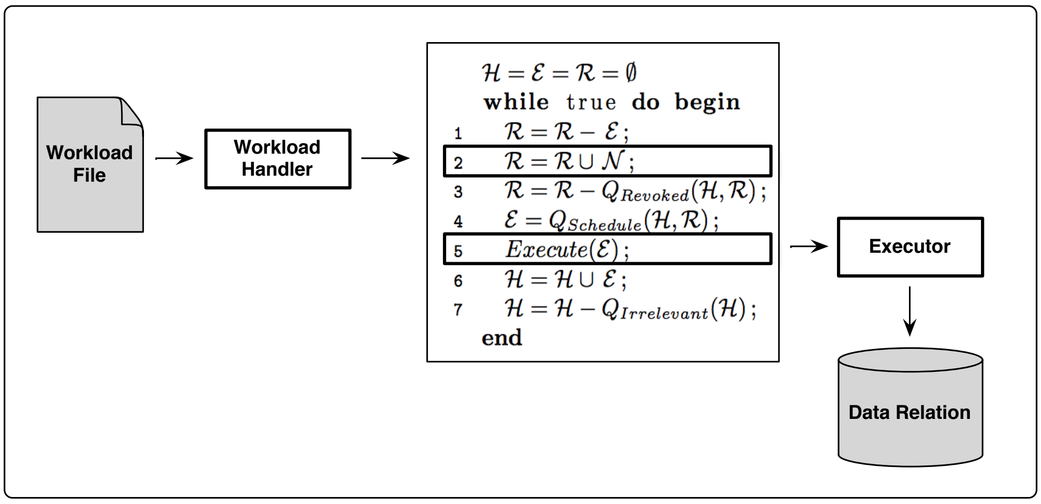
- In **step 1** all requests executed in the previous iteration of the scheduling algorithm carried by $\mathcal{E}$ will be removed from $\mathcal{R}$, since they are not pending anymore, but have been scheduled.
- **Step 2** is an ingoing interface of the OSM where new requests are inserted. Relation $\mathcal{N}$ maintains new requests and is merged with $\mathcal{R}$ that holds pending requests.
- In **step 3** $Q_{Revoked}$, one of the protocol specific scheduling queries, selects requests from $\mathcal{R}$ that cannot be scheduled, e.g., because of a deadlock. Selected requests will be removed from $\mathcal{R}$. To perform this task properly, $Q_{Revoked}$ needs additional information from $\mathcal{H}$ storing the relevant history.
- In **step 4** the second scheduling query $Q_{Schedule}$ is performed. It selects executable requests from $\mathcal{R}$ that will be inserted into $\mathcal{E}$. $Q_{Schedule}$ also needs to access $\mathcal{H}$ to determine the executable requests in $\mathcal{R}$.
- **Step 5** is the backend interface of the Oshiya scheduling model. All scheduled requests maintained in $\mathcal{E}$ are executed on the database, which is indicated by the function $Execute(\mathcal{E})$
- In **step 6** all executed request from $\mathcal{E}$ are added to the relevant request history $\mathcal{H}$.
- In **step 7** the third protocol-specific scheduling query $Q_{Irrelevant}$ is executed over $\mathcal{H}$. It selects all requests in $\mathcal{H}$ that are not further relevant for $Q_{Schedule}$ and $Q_{Revoked}$. These requests will be removed from $\mathcal{H}$ to slim $\mathcal{H}$ down and thus make $Q_{Revoked}$ and $Q_{Schedule}$ perform better.

2.2 Architecture of the Oshiya Demo Application

In this section the general architecture of ODA will be explained. ODA implements the OSM, which was described in the previous section. The application maintains the three scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$, and performs the algorithm of the OSM over these relations to process scheduling. Hereby, a protocol is defined through the three scheduling queries $Q_{Revoked}$, $Q_{Schedule}$, and $Q_{Irrelevant}$. A so-called *workload file* contains the set of requests to be scheduled, which are inserted into the OSM in step 2 by the so-called *Workload Handler*. The backend interface of OSM is the execution of the requests located in $\mathcal{E}$, in step 5 of the scheduling algorithm. The so-called *Executor* within ODA implements the indicated $Execute(\mathcal{E})$ function. The functionalities of the Workload Handler as well as the Executor will be explained more detailed in the following two subsections. Fig 2.2 gives an overview of the described architecture of ODA. Notice that the arrows indicated the flow of requests.

2.2.1 Workload Handler Functionalities

To simulate incoming requests the user predefines a workload stored in a workload file. According to the scheduling algorithm ODA schedules these requests by inserting them into the



```
$\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
while true do begin
1   $\mathcal{R} = \mathcal{R} - \mathcal{E}$ ;
2   $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$ ;
3   $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$ ;
4   $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$ ;
5   $Execute(\mathcal{E})$ ;
6   $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$ ;
7   $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$ ;
end
```

Figure 2.2: ODA Application Architecture

OSM over relation $\mathcal{N}$. The workload is a set of requests structured into transactions that are owned by a client. Example 1 introduces a workload consisting of two clients, Alice and Bob, that maintain one transaction each (t_1, t_2). Hereby $r_i(x)$ is a read request of transaction i on an item x and $w_i(x)$ a corresponding write request. c_i is the commit of transaction i. This notation will be used throughout this thesis.
The task of the Workload Handler within ODA is to manage the insertion of the requests defined in the workload file. ODA maintains only one request per transaction within its scheduling model at the same time. Whenever a request in $\mathcal{R}$ has been scheduled and moved to $\mathcal{E}$, e.g., $r_1(x)$, the next request within this transaction $r_1(y)$ is inserted into $\mathcal{R}$ by the Workload Handler.
Furthermore the Workload Handler has to cope with the following challenges:

Example 1 *Consider a shop system maintaining a relation Articles as shown in Table 2.1. ID is the primary key identifying the items (Articles) within the relation, whereas Qty is the number of available pieces of the specific item. There are two clients, Alice and Bob, accessing the $Articles$ relation within the shop system. Alice just browses through the shop without buying anything, which is modeled as multiple read-requests on several items (x, y, z) within the articles relation. Bob initiates a buying process of article x, which is modeled as a read-requests checking the actual available quantity of the item followed by a write-request decreasing the quantity of the item by the number of ordered pieces.*

$$\text{Alice: } t_1 :\ r_1(x)\ r_1(y)\ r_1(z)\ c_1$$

$$\text{Bob: } t_2 :\ r_2(x)\ w_2(x)\ c_2$$

$Articles$

ID	**Qty**
x	3
y	3
z	3

Table 2.1: Articles Relation

Navigational Backward Debugging ODA supports a navigational backward debugging feature. Whenever a user navigates the scheduling algorithm backwards, the Workload Handler removes all requests that have been inserted in the current scheduling iteration from $\mathcal{R}$. Taking Example 1 and a scheduling iteration i where $r_1(y)$ was inserted into $\mathcal{R}$. Whenever the user backwards i, the Workload Handler needs to identify that $r_1(y)$ had been inserted in i and remove it from $\mathcal{R}$.

Additional Workload Functionalities Within ODA the user cannot only define a static workload where values to be written by a write request are fixed, but also provides the functionality to manipulate the value to be written according to previously read values on the data relation. Consider that values written by a write request within ODA are always numeric values. This can be done by defining a variable marked by a '\$' character within the value field of a write request. Taking again Example 1 and Bob who wants to order one piece of article x. Bob first reads the available quantity of x, $r_2(x)$ and then subtracts 1 from the available quantity of x. This is modeled within the workload by the following notation:

$$\$1 - 1 \tag{2.1}$$

$\$1$ is a placeholder for the read value of the first request $r_2(x)$ of that transaction. The Workload Handler looks up the read value, inserts it into the placeholder, and solves the arithmetic term to get the actual value to be written by that request. So assuming $r_2(x)$ reads the value 3, $w_2(x)$ will write the value 2, since $3 - 1 = 2$. Workload functionalities had been further enlarged within this work, as explained in Chapter 5.

2.2.2 Executor Functionalities

The Executor is the functional part of the application that is connected to the backend interface of the OSM. The scheduling algorithm shown in Fig 2.1 indicates the functionality of an Executor by defining a function $Execute(\mathcal{E})$. In the step 5 of the algorithm, the Executor needs to handle the scheduled requests in $\mathcal{E}$ and executes them on the data relation, e.g., the introduced $Articles$ relation. Request can be read, write, commit, or abort requests. Detailed functionalities will be described in Chapter 3, which discusses the support of multi-version protocols in ODA.

2.3 Protocols

One contribution of the thesis is the support of multi-version protocols within ODA. This section briefly describes the differences between multi- and single-version protocols.

2.3.1 Single-Version Protocols

Single-version protocols maintain one tuple of every item x within a relation. Strict Two-Phase-Locking (SS2PL), as an example single-version protocol, handles concurrent requests on a tuple x within a relation using locks. Considering again Example 1, whenever Bob initiates a write-request $w_2(x)$ he needs to hold a write-lock on x. Alice trying to access x by a read-requests $r_1(x)$ needs a read-lock on x. SS2PL ensures that whenever a transaction, in this case t_2, holds a write-lock on a tuple, no other locks on this tuple may be assigned to other transactions. So concurrent reads and writes on the same item will be blocked [ENN10].

2.3.2 Multi-Version Protocols

Multi-version protocols hold multiple versions x_i of an item x. Snapshot Isolation (SI) is an example multi-version protocol that lets every write requests $w_j(x)$ create a new version x_j of item x that is only visible to other transactions after transaction j comitted (c_j). A read request $r_i(x)$ on the same object accesses the latest version of x, written by a transaction that committed before. Thus reads on an object x do not block writes on the same object and vice versa [TGBK11]. So whenever Bob initiates a write request $w_j(x)$ on x, a new version x_j will be create within the $Articles$ relation. Alice who wants to access x concurrently can access the previous version x_i.

3 Enabling Multi-Version Protocols

In this chapter the concepts and implementation of multi-version protocols within ODA will be explained. Multi-version protocols maintain multiple versions x_i of an item x. Each write request $w_i(x)$ creates a new version of item x, that is visible to other transactions after c_i. To identify a version of an item x a tuple (TA, Seq) is used, where TA is the transaction that created the version, and Seq the position of the request within the transaction, according to [TGBK11]. (TA,Seq) is a unique key for a version, since the transaction number TA is unique within the OSM of ODA, and Seq covers the case that a transaction does more than one write on an item. Enabling multi-version protocols within ODA requires three basic adaptions. First, the primary key of the data relation needs to be enhanced by the named version attributes (TA, Seq) to store multiple version of an item (Section 3.1). Second, the scheduling relations of the OSM need to be enhanced by the version attributes in order to process requests accessing a version x_i of an item x (Section 3.2). Third, an Executor executing requests scheduled by multi-version protocols needs to be implemented (Section 3.3).

3.1 Enhancements to the Data Relation

Considering the data relation $Articles(\underline{ID}, Qty)$ from Example 1. The underlining of $\underline{ID}$ indicates the primary key of the relation that uniquely identifies an item. To support multi-version protocols the data relation needs to store multiple versions x_i of an item x, so $\underline{ID}$ as primary key is no longer sufficient. As described above, the version identifier i is a composition of the transaction number TA, and the position Seq of the write request that created that version. The primary key $\underline{ID}$ is enlarged by the two attributes $\underline{CTA}, \underline{CSeq}$, where 'C' stands for 'create', to make clear this is the transaction that created the version as shown in 3.1. Since it might be also applicable to store the transaction that deleted an item, the data relation is also extend by the attributes $DTA, DSeq$, where 'D' stands for 'Delete' [TGBK11].

$$Articles(\underline{ID}, \underline{CTA}, \underline{CSeq}, DTA, DSeq, Qty) \tag{3.1}$$

Example 2 *Consider a relation Article maintained by a shop system supporting multi-version protocols, as shown in Table 3.1. Bob initiates a write request $w_2(x)$ on item x, subtracting one piece from the available quantity of x. Alice concurrently wants to access x through a read request $r_3(x)$. $w_2(x)$ creates a new version of item x, whereas $r_3(x)$ accesses a prior version of x, generated by a transaction t_1 that already committed.*

Articles

ID	CTA	CSeq	DTA	DSeq	Qty
x	1	1			3
x	2	1			2

Table 3.1: Articles Relation Supporting Multi-Version Protocols

3.2 Enhancements to the Oshiya Scheduling Model

As described in Section 2.1 the OSM models requests as tuples in the so-called scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$. $\mathcal{R}$ (pending requests) stores all the incoming requests waiting to be executed, $\mathcal{E}$ (executable requests) the scheduled requests to be executed, and $\mathcal{H}$ (relevant history) stores the scheduled requests as a relevant history. The schema of these relations processing single-version protocols is shown in 3.2. Notice that additional attributes that are not necessary for the actual scheduling process (e.g. Val the value to be written for write requests) are omitted for simplicity reasons.

$$\mathcal{R}(\underline{TA}, \underline{Seq}, Op, OID)\ \mathcal{E}(\underline{TA}, \underline{Seq}, Op, OID)\ \mathcal{H}(\underline{TA}, \underline{Seq}, Op, OID) \tag{3.2}$$

A request within these relations is uniquely identified by its transaction number TA and its position within that transaction Seq. Op indicates the type of the request, which can be a read, write, commit or abort. OID stores the item within the data relation the request accesses on.

To support multi-version protocols the schema of the scheduling relations needs to be adapted as following. A multi-version protocol enhances a pending read request $r(x)$ by the version of x when being scheduled, so being moved from $\mathcal{R}$ to $\mathcal{E}$. Thus, $\mathcal{E}$ only contains read requests of the form $r(x_i)$, why it is enlarged by a version identifier OTA, $OSeq$ that corresponds to the version attributes CTA, $CSeq$ in the data relation. Moreover, relation $\mathcal{H}$ is also enlarged by OTA, $OSeq$, since it stores the relevant history, so requests from $\mathcal{E}$. According to that, relations $\mathcal{E}$ and $\mathcal{H}$ of the OSM need to be adapted when processing multi-version protocols as shown in 3.3.

$$\mathcal{E}(\underline{TA}, \underline{Seq}, Op, OID, OTA, OSeq)\ \mathcal{H}(\underline{TA}, \underline{Seq}, Op, OID, OTA, OSeq) \tag{3.3}$$

Example 3 *Table 3.2 shows relation $\mathcal{E}$ after requests $r_3(x)$ and $w_2(x)$, introduced in Example 2 have been scheduled and thus, inserted into $\mathcal{E}$. As we can see the OTA and $OSeq$ are set to $(1, 1)$, indicating the version to be read by $r_3(x)$ corresponding to relation Articles in Table 3.1.*

3.3 Different Functionalities of the Executor

As described in Section 2.2 executable requests (reads, writes, commits, aborts) in relation $\mathcal{E}$ are executed on the data relation S through the Executor in step 5 of the scheduling algorithm of the OSM (Fig 2.1). Since ODA also provides backward debugging of protocols, the

$\mathcal{E}$

TA	Seq	Op	OID	OTA	OSeq
2	1	w	x		
3	1	r	x	1	1

Table 3.2: Relation $\mathcal{E}$ Maintaining Requests Scheduled by a Multi-Version Protocol

Executor also needs to revert the execution of requests besides the 'normal' forward execution. Executing requests scheduled by multi-version is different from executing single-version requests. The functionalities of both types of Executors will be described in this section.

Single-Version Executor A single-version Executor handles possible read, write, commit or abort requests as following:

Reads: A read-requests $r(x)$ is executed as a lookup of an item x located in S. The read item will be stored by the Executor in a relation *ReadValueMap* (RVM) that stores all read values read by read requests. This needs to be done, since the read values might be further used by the workload as we will see in Chapter 5, where sophisticated workload functionalities will be described. In the case of reverting a request $r(x)$ this stored item x in the RVM is removed again.

Writes: Handling write-requests in single-version protocols, the following two cases must be considered. If the item, the request accesses on, already exists within S, x needs to be updated by the value given in the request. Unlike when the item the request accesses on does not exist within S. In this case the Executor needs to create a new item x in S with the given value of the write request. Undoing a write request $w(x)$ the Executor restores the item x to the state before the execution of $w(x)$ respectively removes the item x if it was newly created by that request.

Commits: Commits do not have to be handled by the Executor since ODA shirks the concurrency control of the database system maintaining the data relation by auto-committing every single request.

Aborts: Since the Executor of ODA auto-commits every single request on the data relation of a transaction i, an abort a_i needs to be handled within the Executor by undoing all changes effected by i on items $x...y$. Moreover all read values of i in the RVM are removed. Undoing an abort a_i is the most challenging task the Executor has to face. All undone changes affected by i on $x...y$ need to be restored.

The described functionalities are summarized in the following Table 3.3.

Request	Forward	Backward
read	Insert x into RVM	Remove x from RVM
write	Update x in S / Insert x into S	Restore x in S / Remove x in S
commit	-	-
abort	Undo changes of i in S	Restore undone changes of i in S

Table 3.3: Single-Version Executor Functionalities

Multi-Version Executor Executors executing scheduled requests of multi-version protocols are facing different functional requirements compared to single-version Executors. Thus, a new Executor is introduced handling requests (read, write, commit, abort) scheduled by multi-version protocols as following:

Read: A read-request $r(x_i)$ is executed as a lookup of the version of an item x_i located in the data relation S. Parallel to single-version protocols the read item will be stored by the Executor in the RVM, since it might be further used by the workload for sophisticated workload functionalities. In the case of reverting a read-request $r(x_i)$ this stored item x_i in the RVM is removed again.

Write: A write-request $w_j(x)$ scheduled by a multi-version protocol is handled differently from the ones scheduled by single-version protocols. j uniquely identifies the transaction and the position within that transaction. $w_j(x)$ creates a new version x_j of an item x in S, whether a version of x already existed in S or not. Reverting $w_j(x)$ the created version x_j is removed from S again, so there does not have to be made any restoring of an overwritten value as in single-version Executors.

Commit: Parallel to single-version Executors, commits do not have to be handled in the multi-version Executor either, since both auto-commit every single request on S.

Abort: Handling an abort a_j, all effected changes on S by transaction j need to be undone. This is done by removing all versions of items $x_j...y_j$ created by j in S. Besides that, all stored values of items $x_i...y_i$ read by j are removed from the RVM. Accordingly undoing an abort a_j, through the backward debugging feature, all removed versions of items $x_j...y_j$ are restored as well as the stored values of items $x_i...y_i$ in the RVM.

The described functionalities are summarized in Table 3.4. The comparison of Table 3.3 and 3.4 clearly outlines the different requirements of a single- and multi-version Executor.

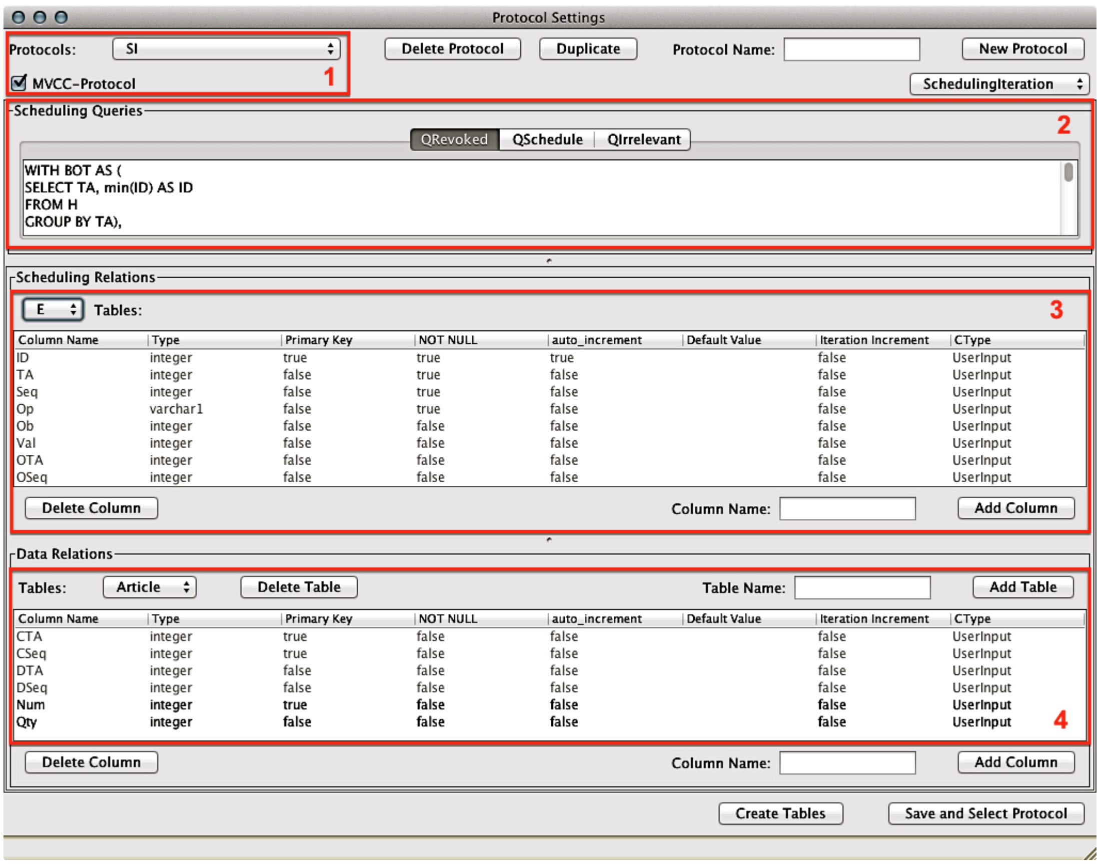
Request	**Forward**	**Backward**
read	Insert x_i into RVM	Remove x_i from RVM
write	Insert x_j into S	Remove x_j from S
commit	-	-
abort	Remove $x_j ... y_j$ from S	Restore $x_j ... y_j$ in S

Table 3.4: Multi-Version Executor Functionalities

3.4 Implementation

As described in the previous Sections 3.1-3.3 multi- and single-version protocols have different requirements on the OSM, the data relation, as well as on the Executor. Implementing the support of multi-version protocols in ODA is done by a clear distinction between these two types of protocols. Fig 3.1 shows the 'Protocol Settings' form of ODA, where protocols are developed and edited. Mark (1) shows a combo-box 'Protocols' where saved protocols are selected. The current selected protocol is 'SI', a multi-version protocol, indicated by the checkbox 'MVCC-Protocol' below. When developing a protocol the user clearly has to state, if the protocol is a multi-version protocol or not. According to that ODA automatically adjusts the the scheduling relations and data relation of the protocol by adding version attributes $(OTA, OSeq)$ to relation $\mathcal{E}$ and $\mathcal{H}$ mark (3), as well as $(CTA, CSeq, DTA, DSeq)$ to the data relation, in this case named 'Article' mark (4). The greying-out of some of the attributes in mark (3) and mark (4) indicates that these are system attributes that cannot be edited by the user, since they are compulsory for the scheduling process. In the section 'Scheduling Queries' mark (2) the protocol specific scheduling queries $Q_{Revoked}$, $Q_{Schedule}$ and $Q_{Irrelevant}$ are implemented.

According to whether the selected protocol is a single- or a multi-version protocol, the corresponding Executor is chosen by ODA. Both Executors implement the interface *IExecutor* as shown in Fig 3.2. The $Execute(\mathcal{E})$ function executes all requests in $\mathcal{E}$, whenever the scheduling algorithm of the OSM is in step 5. The undo() function is called, whenever the user reverts step 5 of the algorithm. Using the interface *IExecutor* additional Executors can be implemented in ODA within future work, for protocols that are not supported so far.



```
WITH BOT AS (
SELECT TA, min(ID) AS ID
FROM H
GROUP BY TA),
```

Column Name	Type	Primary Key	NOT NULL	auto_increment	Default Value	Iteration Increment	CType
ID	integer	true	true	true		false	UserInput
TA	integer	false	true	false		false	UserInput
Seq	integer	false	true	false		false	UserInput
Op	varchar1	false	true	false		false	UserInput
Ob	integer	false	false	false		false	UserInput
Val	integer	false	false	false		false	UserInput
OTA	integer	false	false	false		false	UserInput
OSeq	integer	false	false	false		false	UserInput

Column Name	Type	Primary Key	NOT NULL	auto_increment	Default Value	Iteration Increment	CType
CTA	integer	true	false	false		false	UserInput
CSeq	integer	true	false	false		false	UserInput
DTA	integer	false	false	false		false	UserInput
DSeq	integer	false	false	false		false	UserInput
Num	**integer**	**true**	**false**	**false**		**false**	**UserInput**
Qty	**integer**	**false**	**false**	**false**		**false**	**UserInput**

Figure 3.1: Protocol Settings Form

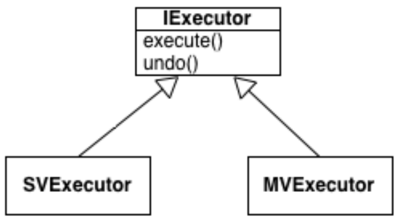


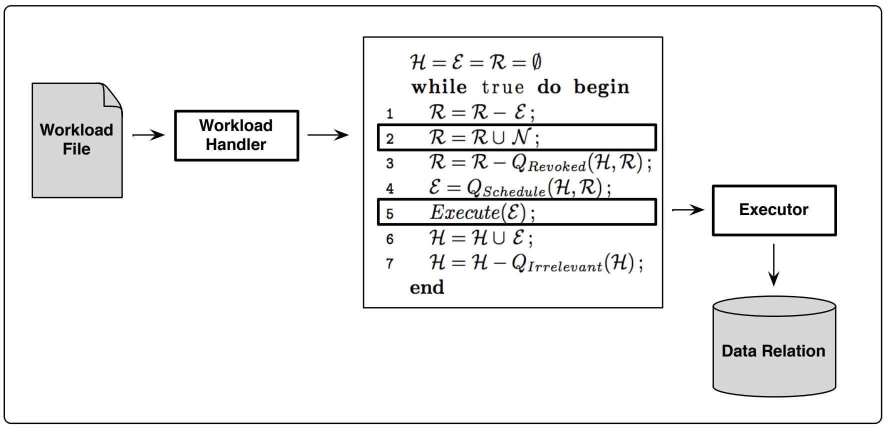
Figure 3.2: Executors in ODA

4 Enabling Simultaneous Protocol Execution

In this section the concepts and implementation supporting simultaneous execution of multiple protocols will be explained. First a quick overview of ODA executing a single protocol will be given. Afterwards the concepts and challenges behind the execution of multiple protocols will be introduced, as well as the implementation in ODA.

4.1 Executing a Single Protocol in ODA

Fig 4.1 shows the architecture of ODA running a single protocol as described in section 3.2. ODA implements the OSM, consisting of the scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$, and the seven-step scheduling algorithm that iteratively performs the protocol specific scheduling queries $Q_{Schedule}$, $Q_{Revoked}$, and $Q_{Irrelevant}$ on the scheduling relations. A predefined workload file contains the set of requests to be scheduled by the OSM. The Executor executes the requests on the data relation S. The insertion of the requests defined in the workload file into relation $\mathcal{R}$ is done by the Workload Handler. Whenever the user starts the scheduling process, ODA iteratively runs through the seven-step scheduling algorithm. In every iteration (step 2) the Workload Handler checks, if the previous requests had been scheduled and respectively inserts new requests into relation $\mathcal{R}$. The Executor implements the $Execute(\mathcal{E})$ function in step 5 of the algorithm and executes scheduled requests on the data relation as described in the previous Chapter 3.



```
   $\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
   while true do begin
1      $\mathcal{R} = \mathcal{R} - \mathcal{E}$ ;
2      $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$ ;
3      $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$ ;
4      $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$ ;
5      $Execute(\mathcal{E})$ ;
6      $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$ ;
7      $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$ ;
   end
```

Figure 4.1: ODA Architecture for Single Protocol Execution

4.2 Concepts Enabling Multiple Protocol Execution

Enabling simultaneous protocol execution within ODA the following concepts are central: First, ODA provides the possibility to let multiple protocols run parallel and independently. Thus, each protocol needs its own OSM consisting of the three scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$, the scheduling algorithm, and data relation S. Secondly, all previous functionalities are maintained within this extension, meaning the debugging mode, statistics etc. Third, to ensure comparability, the parallel running protocols schedule the same workload, which can become problematic since the workload cannot be standardized for all protocols, such as class based protocols that need an additional class attribute. An example class based protocol (Class Based 2PL) will be briefly explained in paragraph 'Running same Workload'. Executing multiple protocols **independently**, **simultaneous** but on the **same workload** results in the enhancement of the program architecture as shown in Fig 4.2. The application also allows to run only one protocol as prior to the enhancement, as a well as multiple protocols. This is done by generalizing the protocol specific application parts and multiply them by the number of concurrent selected protocols. These protocol specific application parts are the Workload Handler, inserting new requests from the workload file into the scheduling model, the OSM, consisting of the seven-step algorithm and the scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$, as well as the Executor, executing scheduled requests from the OSM on the data relation. The following paragraphs go more into detail concerning the challenges running the same workload for multiple protocols, as well as the synchronization and independency of multiple protocol execution.

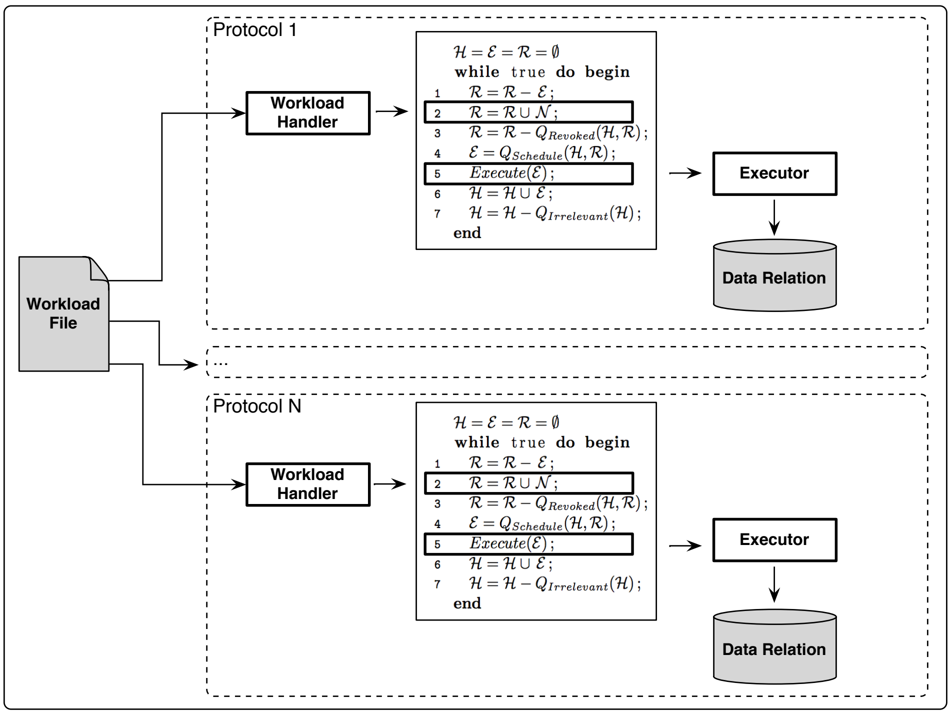

Protocol 1
Workload Handler
$\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
while true do begin
1 $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2 $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3 $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4 $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5 $Execute(\mathcal{E})$;
6 $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7 $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;
end
Executor
Data Relation
Workload File
...
Protocol N
Workload Handler
Executor
Data Relation

Figure 4.2: ODA Architecture for Multiple Protocol Execution

Running same Workload The workload is the set of requests to be scheduled by a protocol implemented in the OSM. In the OSM requests are modeled as tuples in the scheduling relations. In step 2 of the scheduling algorithm the Workload Handler inserts requests from the Workload File into relation $\mathcal{R}$, so the defined requests in the Workload File need to have the same schema than relation $\mathcal{R}$. Using the same Workload File for multiple protocols this can become problematic. It is conceivable that not all protocols have the same schema for incoming requests, respectively relation $\mathcal{R}$. E.g., Class-Based Two Phase-Locking (Class Based 2PL), where an additional class-attribute in incoming requests indicates the priority of a request, as Example 4 illustrates. Within this work, the first approach to simultaneously execute multiple protocols in ODA, the assumption was made that all simultaneously executed protocols need to have the same schema for incoming requests. So, e.g., Class Based 2PL and SS2PL cannot be run simultaneously, since they do not have the same schema for incoming requests.

Example 4 *Consider Alice and Bob both initiating a write request $w_1(x)$ resp. $w_2(x)$ concurrently to order an article x within the articles relation. Alice is a premium customer whereas Bob is a normal customer. The shop system uses a Class Based 2PL to process scheduling and to model the different customer types. Alice's request is of higher relevance since she is a premium customer (class 1), indicated by a class attribute within her incoming request $w_1^1(x)$. Bob however is a normal customer, so within his incoming request $w_2^2(x)$, the class attribute is set to 2 to indicate that his request is of lower relevance.*

Independency To ensure that multiple protocols do not interfere with each other, each protocols needs its own OSM, data relation, Workload Handler and Executor. This consequently means that each protocol needs independent scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$, where the scheduling algorithm performs on, as well as an independent data relation, where the scheduled requests are executed on. As described in Section 2.2 the OSM within ODA only maintains one request per transaction at the same time, so whenever a request had been scheduled, the next request within this transaction is inserted into the OSM. The scheduling of requests is protocol specific, so the insertion of requests into the OSM is also protocol specific. Thus, every protocol needs an independent Workload Handler that inserts requests from the Workload File into the OSM. Since every protocol maintains its own data relation, the execution of requests also needs to be done per protocol, so each protocol maintains its own Executor.

Synchronization During the scheduling process when the application runs iteratively through the seven-step scheduling algorithm, each step is performed parallel on all selected protocols. In other words ODA makes the scheduling algorithms wait for each other. Synchronization is fundamental, since it ensures the navigational debugging feature for multiple protocols, as well as stepwise protocol comparison.

4.3 Implementation

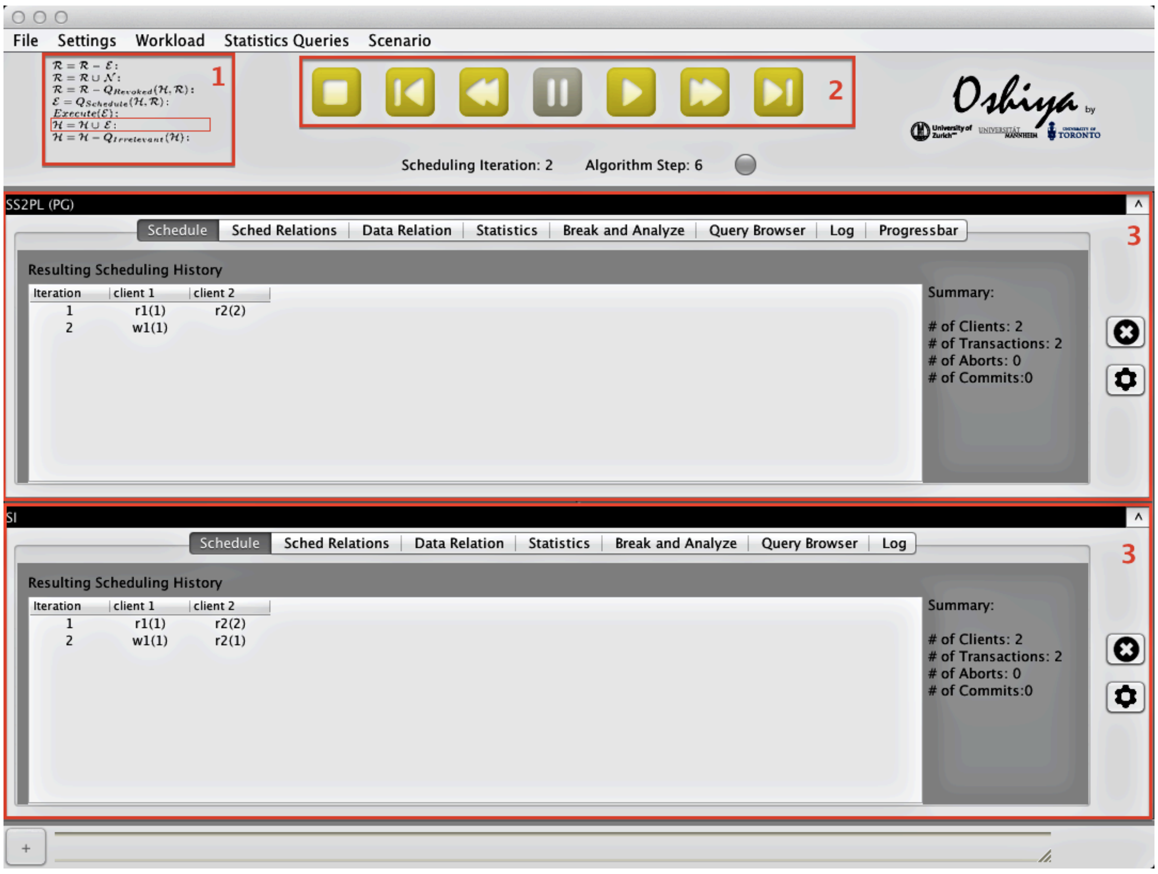
In this section the implementation of the pre-described concepts executing multiple protocols within ODA will be explained. First in 4.3.1 the GUI of ODA executing multiple protocols will be described, afterwards in 4.3.2 we will go into details on the adaptions of the core logic of the application.

4.3.1 GUI

For multiple protocol execution a new GUI concept was implemented. A strict distinction between protocol-specific and general controls and views needed to be done. Fig 4.3 shows the main frame of ODA running two protocols, SI and 2PL, currently paused in the second scheduling iteration in step 6.

General Controls The controls in mark (1) are the so-called *navigational controls*, where the user can play and pause the scheduling process of all selected protocols, as well as debug forward and backwards and reset the scheduling process. The labels below the navigational controls indicate the current scheduling iteration and step. The red rectangle in mark (2) frames the currently executed step in the scheduling algorithm of the OSM. In the 'Settings' item within the menu-bar the user sets the database, on which ODA performs the scheduling. The 'Workload' item lets the user import, generate, or save a workload file for execution. All these controls are general controls, meaning not protocol specific.

Protocol Panel The protocol panel in mark (3) was introduced to hold all protocol specific controls and views. It can be added generically to the main frame and consists of the following tabs. 'Schedule' shows the resulting schedule of the scheduling process so far, in this case after step 6 in iteration 2. Notice that the schedule is not shown in a total order, as ordinary known, but in a simplified manner. Requests executed in the same scheduling iteration are displayed as concurrently executed.
Furthermore the number of clients and transactions of the selected workload is shown, as well as the number of aborts and commits that resulted from the scheduling process so far. The 'Sched Relations' tab shows the state of the three scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$ of the OSM for the specific protocol. This tab is especially useful for the user to see which requests are selected by the scheduling queries $Q_{Schedule}$, $Q_{Revoked}$, and $Q_{Irrelevant}$. The tab 'Data Relation' shows the state of the data relation at each step of the scheduling process. The 'Statistics' tab shows the statistical analysis of a protocol, which can be individually configured by the user. The 'Progressbar' tab only exists for one protocol. It shows the amount of processed transactions in proportion to all transactions defined in the workload, for all selected protocols.



$\mathcal{R} = \mathcal{R} - \mathcal{E}$:
$\mathcal{R} = \mathcal{R} \cup \mathcal{N}$:
$\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$:
$\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$:
$Execute(\mathcal{E})$:
$\mathcal{H} = \mathcal{H} \cup \mathcal{E}$:
$\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$:

SS2PL (PG)

Resulting Scheduling History

Iteration	client 1	client 2
1	r1(1)	r2(2)
2	w1(1)	

Summary:

# of Clients: 2
# of Transactions: 2
# of Aborts: 0
# of Commits:0

SI

Resulting Scheduling History

Iteration	client 1	client 2
1	r1(1)	r2(2)
2	w1(1)	r2(1)

Summary:

# of Clients: 2
# of Transactions: 2
# of Aborts: 0
# of Commits:0

Figure 4.3: ODA-GUI Executing Multiple Protocols

4.3.2 Core Adaptions

This subsection explains adaptions to the core logic of the application that had to be made in order to enable simultaneous protocol execution.
Following the pre-described concept, shown in Fig 4.2, the protocol specific program logic parts consisting of the Workload Handler, the OSM, and the Executor are multiplied by the number of protocols selected. Moreover every protocol needs an autonomous database schema, where its scheduling relations $\mathcal{R}, \mathcal{E}, \mathcal{H}$ and data relation is located on. Fig 4.4 shows an class diagram of the relevant part in ODA implementing synchronous protocol execution. The *Oshiya Controller* is the main controller that launches the application. The Oshiya Controller maintains a thread *Algo*, performing the seven-step algorithm of the OSM. The functions within Algo correspond to the navigational controls mark (1) in Fig 4.3. The class *Basis Functions* implements the seven step algorithm of the OSM, within the function *forwardStep(int:step)*. Since ODA also provides the feature of backward debugging Basis Functions also needs to implement the undoing of the seven-step scheduling algorithm, which is implemented in function *backwardStep(int:step)*. Each selected protocol maintains its own OSM, so the application holds an instance of Basis Functions for each selected protocol, indicated by the '*' cardinality within the class diagram. Moreover, each instance of Basis Functions

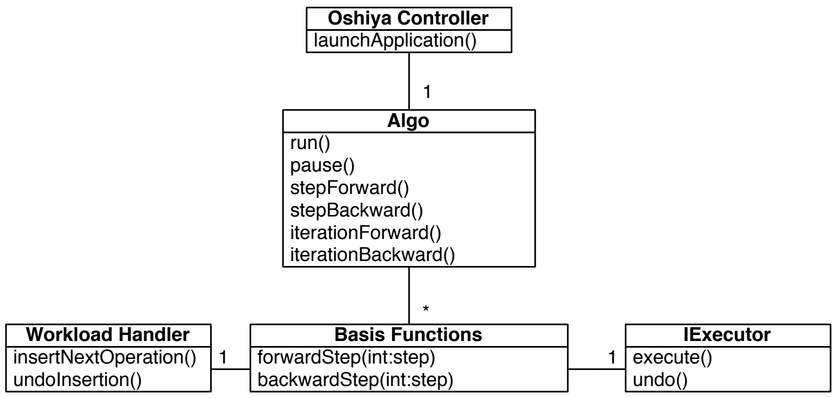


Figure 4.4: Class Diagram of ODA Showing Relevant Classes for Synchronous Protocol Execution

maintains an instance of the Workload Handler, as well as the Executor.
Whenever Basis Functions executes step 2 of the scheduling algorithm it asks for new requests calling the function *insertNextOperation()* on the Workload Handler, corresponding to the application architecture in Fig 4.2. In the case of undoing step 2 through the backward debugging feature the method *undoInsertion()* is called. It removes the previously inserted requests from the OSM, respectively relation $\mathcal{R}$.
The Executor is either an instance of a multi-version Executor or a single-version Executor, as shown in Fig 3.2, corresponding to whether Basis Functions implements the OSM of a single- or multi-version protocol. Function *execute()* is called in every step 5 of the scheduling algorithm and executes the scheduled requests in $\mathcal{E}$ on the data relation. Whenever a user backwards step 5 of the scheduling algorithm the function *undo()* of the Executor is called to undo the executed requests. The detailed functionalities of the two types of Executors as well as the functions *execute()* and *undo()* are described in Section 3.3.

5 Sophisticated Workload Functionalities

The simulation of realistic scenarios is fundamental for ODA, since it is the basis for all other functionalities like the investigation of protocol behavior or the statistical analysis of different protocols. A workload is the set of requests to be scheduled by the selected protocols. Thus, it defines the scenario, for which the user wants to simulate the behavior of different protocol implementations. The goal of this contribution is to provide the user functionalities to model realistic workloads that did not exists prior to this work. For instance, a workload of customers accessing a shopping system. The customer browses through the shop, reads several articles, and finally makes some orders. Hereby, the browsing through the shop is modeled by read requests on an relation *Articles* within the shops database, whereas the buying process is modeled as write request on relation *Articles*, subtracting the ordered pieces from the available quantity of the article. Section 5.1 explains the concepts of the following new workload features, whereas Section 5.2 describes the implementation.
To realistically model the browsing through the shop, delays between the read request are inserted to model the time a customer watches an article until accessing another one, respectively decides to order it. The delay functionality within workloads will be described in Section 5.1.1. Considering a customer, who wants to order a certain amount x of an article. Whenever the available quantity of that article is smaller than x, the user at least wants to order all remaining pieces. This is modeled using conditional write statements, which will be introduced in Section 5.1.2. Whenever the available amount of an article is 0, the ordering of an article is skipped, this is modeled using conditional read request, which can also be called jump constraints, explained in Section 5.1.3.

5.1 Concepts

5.1.1 Delay Functionality

The OSM only maintains one request per client as explained in Section 2.2. Whenever the previous request of a client has been scheduled by the OSM, the Workload Handler jumps further to the next request defined in the workload and inserts it into relation $\mathcal{R}$. To model delays between requests so called *delay requests* are introduced within the workload. Delay requests are pseudo-requests, that are not inserted into the OSM. Whenever the next request the Workload Handler wants to insert is a delay request, it does not insert it, but waits for the next iteration to jump to next request within the workload. Thus, the insertion of the next 'real' request is delayed by one iteration.

ODA does not only provide the possibility to model one delay request between two 'real' requests, but also multiple. Thus, it is also possible to model a delay for multiple scheduling iterations. This approach of enabling delays in between requests has the advantage that the original functionality of the Workload Handler inserting requests does not have to be modified, but only the handling of the newly introduced delay requests has to be implemented.

Example 5 *Considering Alice browsing through a web shop, modeled as multiple read request on a relation Articles. We model the time Alice watches an article until clicking further to the next article by adding delays in between the read requests as shown in Fig 5.1.*

$$Alice : r_1(x)\; d\; r_1(y)\; d\; d\; r_1(z)\; d\; d\; d\; c_1$$

Figure 5.1: Delays in Between Requests

5.1.2 Conditional Write Expressions

The idea of conditional write statements is to define a condition that manipulates the value to be written by write request. Prior to this work, there was the possibility to manipulate the value to be written according to a previously read value, as described in Section 2.2, using a placeholder within the value field of a write request. For instance the following expression models such a placeholder:

$$\$1 - 1 \tag{5.1}$$

$\$1$ is a placeholder for the read value of the first request within that transaction t. The Workload Handler looks up the value read by transaction t, inserts it into the placeholder and solves the arithmetic term to get the actual value to be written by that request. This concept was enlarged to enable conditions within write requests introducing the following syntax:

$$[Condition], [Then], [Else] \tag{5.2}$$

Example 6 *Bob wants to order three pieces of an article x. In the case that less than three pieces are available he at least wants to order all available pieces. So he defines a maximum order quantity and not a fix quantity. This can be modeled using Expression 5.2 within the write request $w(x)$ of his order. The whole transaction of his order is shown in Fig 5.2.*

$$Bob : r_1(x)\; w_1(x)\; c_1$$

Figure 5.2: Bob Ordering Article x

For instance '$\$1 \geq 3, \$1 - 3, 0$' models a conditional write request according to Example 6. The expression is separated into three parts by comma characters. The first part '$\$1 \geq 3$' is the condition, the second part '$\$1 - 3$' the value to be written if the condition is true, and the third part '0' the value to be written if the condition is false. The values to be written can be arithmetic terms referencing prior read values or fix values. So in this case, whenever the read value $\$1$ of the first request r_1x within that transaction 1 is bigger than 3, the write request subtracts 3 from the value of x, else it writes the value 0.

5.1.3 Conditional Read Expressions/Jump Functionality

The idea behind conditional read requests is to provide the user a syntax, that allows jumps within the workload, according to a defined condition. The syntax of conditional read requests is shown in Expression 5.3, whereas the condition always refers to the value read by the request.

$$[Condition], [Then] \tag{5.3}$$

Example 7 *Bob wants to order one piece of an article x. In the case that the available quantity of x is smaller than 1, in other terms x is sold out, the write request updating the quantity of x should be skipped. This can be modeled by using Expression 5.3 within the read request. Considering Bob's transaction shown in Fig 5.2, whenever $r_1(x)$ reads a value less than 1 the next request inserted by the Workload Handler will not be $w_1(x)$ but c_1.*

'$< 1, 2$' is an example read constraint for Example 7. The expression is structured into two parts separated by a comma character. The first part '< 1' defines the condition whereas the second part '2' indicates the jump size that will be applied by the Workload Handler, whenever the condition is true. So in this case when the value read by $r_1(x)$ is smaller than 1 the Workload Handler will not jump one request further, so to $w_1(x)$ in the next scheduling iteration, but will skip the next request and jump two requests further, so to c_1.

5.2 Implementation

This section describes the implementation of the named sophisticated workload functionalities. First, the general implementation of the Workload Handler is explained. Afterwards the implementation of the described functionalities is explained.

5.2.1 General Implementation of Workload Handler

Fig 5.3 shows a more detailed view of the application architecture of ODA focusing on the Workload Handler. The bold arrows within the figure indicate the flow of requests, whereas the dotted arrows indicate the data flow. The Workload Handler maintains two relations *Workload* and *ReadValueMap* that have been omitted before for simplicity reasons. The following two paragraphs explain the functionality and purpose of these relations.

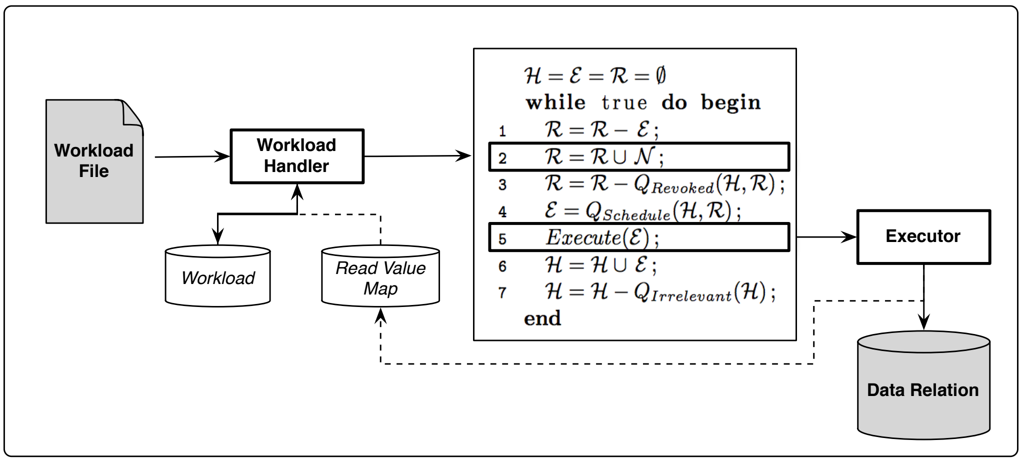

Workload File
Workload Handler
Workload
Read Value Map
$\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
while true do begin
1 $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2 $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3 $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4 $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5 $Execute(\mathcal{E})$;
6 $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7 $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;
end
Executor
Data Relation

Figure 5.3: Detailed View of Workload Handler Architecture

Workload Relation The Workload Handler maintains a relation *Workload*. Whenever a protocol and a workload file has been selected by the user, the requests defined in the workload file are inserted as tuples into relation Workload. Holding the predefined requests as tuples in a relation is advantageous, since they will be also inserted into the OSM as tuples. The schema of relation Workload differs depending on the protocol selected, since protocols can have a different schemas for incoming requests as explained in Section 4.2. Expression 5.4 shows the minimal schema of relation Workload that is used, e.g., by SS2PL and SI.

$$Workload(\underline{CID}, \underline{TA}, \underline{Seq}, Op, Ob, Val) \tag{5.4}$$

Requests within the relation Workload are uniquely identified by an unique id of the client (CID), the transaction number (TA) that is unique within the client id, and the position of the request within that transaction (Seq). Attribute Op identifies the operation of the request, which can be a read, write, or commit. ODA does not support aborts within a workload. Aborts can only be generated by the scheduler, i.e. the OSM. Attribute Ob identifies the object within the data relation the request accesses. This attribute is only set for read and write request, since commits do not access an object within the data relation. The Val field carries the value to be written by a write request, this can be a fix value, e.g. 0, as well as an arithmetic expression as shown in Expression 5.1. The relation Workload maintaining the workload introduced in Example 8 is shown in Table 5.1. Notice that Alice's client id (CID) is 1 whereas Bob's client id (CID) is 2 and item x is identified as object 1 and y as object 2.

Example 8 *Consider Alice and Bob both ordering one piece of an article x and y within one transaction each. The requests on a data relation that stores article items is shown in Fig. 5.4. First, a read request checks the available quantity of x, then a write request subtracts 1 of the available quantity of the article, and finally the transaction commits. In a second request the same is done for the second article y.*

$$Alice : r_1(x)\ w_1(x)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

$$Bob : r_1(x)\ w_1(x)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

Figure 5.4: Alice and Bob Ordering two Different Articles Each

$Workload$

	CID	TA	Seq	Op	Ob	Val	
$P_{Alice} \rightarrow$	1	1	1	r	1		$*$
	1	1	2	w	1	$1-1	$*$
	1	1	3	c			$*$
	1	2	1	r	2		
	1	2	2	w	2	$-1	
	1	2	3	c			
$P_{Bob} \rightarrow$	2	1	1	r	1		
	2	1	2	w	1	$1-1	
	2	1	3	c			
	2	2	1	r	2		
	2	2	2	w	2	$-1	
	2	2	3	c			

Table 5.1: Workload Relation Including a Pointer for Alice and Bob

The Workload Handler maintains a pointer for each client (P_{Alice} and P_{Bob}) pointing on the request that is currently processed in the OSM. In Table 5.1 the pointer of Alice and Bob are both pointing on the first request within the first transaction $r_1(x)$. Thus, request $r_1(x)$ of Alice and Bob are currently processed in the OSM.
For every iteration i of the scheduling algorithm, the Workload Handler checks in step 2 if the current request of a client has been scheduled, i.e. moved into relation $\mathcal{E}$, in iteration $i-1$. If so, the Workload Handler moves the pointer of that client further to the next request within the workload relation and inserts it into relation $\mathcal{R}$.
Corresponding to the forward moving of the pointers, the Workload Handler moves the pointers backward in the case the user makes use of the backward debugging feature and reverts step 2 of a scheduling iteration.

According to the schema of relation Workload shown in expression 5.4 the schema of relation $\mathcal{R}$ is the following:

$$\mathcal{R}(\underline{TA}, \underline{Seq}, Op, Ob, Val) \tag{5.5}$$

Notice that requests in relation $\mathcal{R}$ are uniquely identified by the transaction number TA and position of the request within that transaction Seq, whereas TA, Seq is not unique within relation Workload. When inserting request of a transaction from relation Workload into $\mathcal{R}$, the Workload Handler needs to assign a new unique transaction number for the requests within that transaction before inserting them into $\mathcal{R}$.

Read Value Map Relation $ReadValueMap$ is accessed by the Workload Handler and the Executor. It stores all values read by read requests. It has the following schema, whereas TA, Seq uniquely identifies the request and Val stores the read value:

$$ReadValueMap(\underline{TA}, \underline{Seq}, Val) \tag{5.6}$$

Whenever a read requests is executed on the data relation, the Executor inserts the read value into the ReadValueMap as mentioned in Section 2.2. Relation ReadValueMap is used for the arithmetic expressions that can be defined within the value of a write request. Consider Alice's first transaction in Example 8 $r_1(x)$ $w_1(x)$ c_1 and the corresponding tuples in the workload relation in Table 5.1 marked with a '*'. An arithmetic expression '$\$1 - 1$' is set as value for the write request $w_1(x)$. The placeholder $\$1$ references the value read by the previous read request $r_1(x)$. Whenever the Workload Handler inserts $w_1(x)$ into the OSM, it has to solve the arithmetic expression before. It looks up the value read by $r_1(x)$ in relation ReadValueMap, inserts the value into the placeholder $\$1$, and solves term to get the actual value to be written by $w_1(x)$. Considering $r_1(x)$ read the value 5, 5 is inserted for the placeholder $\$1$ and the term $5 - 1$ is solved by the Workload Handler. So the value written by $w_1(x)$ is 4.

5.2.2 Implementation of the new Sophisticated Workload Functionalities

This subsection describes the implementation of the previous explained sophisticated workload functionalities in Section 5.1. Example 9 visualizes the utilization of these functionalities, which are the delay functionality, as well as conditional read and write expressions.

Example 9 *Consider Alice browsing through a web shop and finally deciding for an item x to buy. This is modeled in two transactions shown in Fig 5.5. The first transaction t_1, the browsing through the shop, are read requests accessing multiple articles (x, y, z) on relation Articles. To model the time Alice watches an article until clicking further to the next article, delays are inserted between the read requests. The second transaction t_2 models the buying process of article x. $r_2(x)$ is a conditional read request, that let's ODA jump directly to c_2, in the case the available quantity of x is smaller than 1. $w_2(x)$ is a conditional write request. Alice wants to order three pieces of x, whenever the available quantity of x is smaller than 3, she at least wants to order all remaining pieces. Table 5.2 shows this workload as tuples in relation Workload. Notice that x is identified as object 1, y as object 2 and z as object 3.*

$$t_1 : r_1(x)\ d\ r_1(y)\ d\ d\ r_1(z)\ d\ d\ d\ c_1$$
$$t_2 : r_2(x)\ w_2(x)\ c_2$$

Figure 5.5: Alice First Browsing Through the Shop Then Buying Article x

$Workload$

	<u>CID</u>	<u>TA</u>	<u>Seq</u>	Op	Ob	Val
$P_{Alice} \rightarrow$	1	1	1	r	1	
	1	1	2	d		
	1	1	3	r	2	
	1	1	4	d		
	1	1	5	d		
	1	1	6	r	3	
	1	1	7	d		
	1	1	8	d		
	1	1	9	d		
	1	1	10	c		
	1	1	1	r	1	<1,2
	1	1	2	w	1	\$1≥3,\$1-3,0
	1	1	3	c		

Table 5.2: Workload Relation Including Delays and Conditional Writes and Reads

Implementation of Delay Functionality As described in Section 5.1.1 ODA models delays as 'pseudo requests' in between 'real requests'. Delay requests can be defined in the workload file, and thus are inserted into the workload relation. Table 5.2 shows the workload relation maintaining the workload of Example 9. The workload is processed request by request moving the pointer (P_{Alice}) further as explained in the previous section. Whenever the pointer points on a delay, no request will be inserted into the OSM.
Pointers pointing on a 'real' request (read, write, commit) are only forwarded, whenever the request has been scheduled. Pointers pointing on a delay are forwarded in every scheduling iteration. So $r_1(y)$ is delayed one scheduling iteration, $r_1(z)$ two scheduling iterations, and c_1 three scheduling iterations.

Implementation of Conditional Write Expressions Enabling conditional write expressions within in ODA is an enlargement of the existing placeholder functionality using relation ReadValueMap. The previously read values are inserted into the placeholders. The Workload Handler checks if the condition is true and replaces the conditional expression by the right value to be written by the write request, according to whether the condition has been true or not.
Considering Alice ordering three pieces of an article x as introduced in Example 9. In the case that there are less than three pieces available, she at least wants to order the remaining pieces. This is modeled using a conditional write expression '$\$1 \geq 3, \$1 - 3, 0$' inserted into the Val field of the write request $w_2(x)$ within the workload relation. Assuming the value read by $r_1(x)$ was 4, the Workload Handler inserts 4 into the placeholders $\$1$ and checks the condition $4 \geq 3$. In this case the condition is true, so the whole conditional expression will be replaced by the value defined for the condition being true $4 - 3 = 1$. So the value written by $w_1(x)$ is 1.

Implementation of Conditional Read Expressions/Jump Functionality Whenever a conditional read request has been executed in scheduling iteration i, the Workload Handler checks in scheduling iteration $i+1$, if the defined condition is true applying the value read by that request. According to the correctness of the conditional expression the pointer within the workload relation is moved. Conditions within read requests are defined in the Val field of the workload relation. This is the simplest approach, since the attribute Val has not been used for read request before anyway.
Taking Example 9, where Alice wants to order three pieces of an article x. In the case that the available quantity of x is smaller than 1, in other terms x is sold out, the write request $w_2(x)$ updating the quantity of x should be skipped. This is modeled using the conditional read expression '$< 1, 2$' inserted into the Val field of the read request $r_2(x)$. Assuming $r_2(x)$ has been executed in scheduling iteration i and read an available quantity of 0 for article x, $w_2(x)$ is skipped. Before inserting the next request in $i + 1$ the Workload Handler inserts the read value into the defined condition of $r_2(x)$ and checks it for correctness. $0 < 1 \rightarrow true$. Whenever the condition is true, the Workload Handler moves the pointer as many steps as specified in the conditional read statement, in this case 2. So the next request inserted into the OSM in $i + 1$ is not $w_2(x)$ but c_2.

6 Conclusion and Future Work

The Oshiya Debugger and Analyzer (ODA) is a tool developed for database developers and researchers to develop, debug and analyze Oshiya protocol implementations with respect to performance- and correctness criteria executing user-provided workloads. In this work ODA was enlarged by three major enhancements: The support of multi-version protocols, the simultaneous protocol execution, and the sophisticated workload functionalities, to enable the simulation of realistic client behavior.

In order to enable multi-version protocols, the backend relation (data relation), on which requests scheduled by the Oshiya scheduling model (OSM) are executed on was adapted to hold multiple versions of a data item, as required by multi-version protocols. Requests scheduled by multi-version protocols access a certain version of a data item, thus the Oshiya scheduling model was enlarged to process such requests. Since requests scheduled by multi-version protocols are executed differently on the underlying backend relation than requests scheduled by single-version protocols, a new Executor was implemented. The support of multi-version protocols opens up a whole new spectrum of concurrency control techniques to the users of ODA, that are widespread in modern database systems. The analysis of multi-version protocols is especially interesting, since there exist many different multi-version protocol implementations with different properties concerning correctness and performance.

Through the simultaneous protocol execution feature ODA provides the user the opportunity to generate schedules for multiple protocols concurrently. In combination with the navigational debugging feature, that lets the user debug the process of schedule generation, this facilitates detailed investigation and comparison of different protocol behaviors.

In order to make expressive analyzations of protocols, realistic workload functionalities are important. The following functionalities within a workload were implemented, to give the user the possibility to define workloads that simulate a sophisticated client behavior, e.g. to model a customer on a web shop: So-called delay requests were introduced, which can be inserted in between 'normal' requests in a workload, e.g., to model the time a customer has a look on an item within the shop until accessing another one. Introducing conditional requests, conditions within the workload can be defined that dynamically adjusts the workload according to the state of the backend relation. This can be used to model, e.g., a minimum or maximum order quantity of a customer.

Future Work ODA does not support simultaneous execution for all types of protocols. Concurrent protocol execution is restricted by the workload, i.e. the schema of the incoming requests. Enlarging ODA to the extend that, e.g., Class based 2PL and 2PL can be executed concurrently, would enable concurrent protocol comparison to a wider spectrum of protocols.

To promote the distribution of ODA, it might be reasonable to make the application web-enabled within future work.

Bibliography

[ENN10] Ramez Elmasri, Shamkant Navathe, and Shamkant Navathe. *Fundamentals of Database Systems*. 2010.

[TGB$^+$11] Christian Tilgner, Boris Glavic, Michael H Böhlen, Carl-Christian Kanne, Patrick Leibundgut, and Luis Schüller. Debugging, visualizing, and comparing scheduling protocols. (0), 2011.

[TGBK11] Christian Tilgner, Boris Glavic, Michael H. Böhlen, and Carl-Christian Kanne. Declarative Serializable Snapshot Isolation. pages 170–184, 2011.