Fachhochschule Nordwestschweiz

IP9 - Algoria: Verteiltes Klassenzimmer

Master Thesis

University of Applied Sciences Northwestern Switzerland - Computer Sciences

Studierender:	Reto Frey
Experte:	Conrad Pomm
Betreuender Dozent:	Prof. Dr. Christoph Stamm

Revision:	1.0
Datum:	8. Februar 2013

Zusammenfassung / Abstract

Zusammenfassung

Algoria Worksheet ist ein Programm, welches Studierenden und Dozierenden die Möglichkeit bietet, den Computer im Algorithmen und Datenstrukturen Unterricht sinnvoll und effektiv einzusetzen. Dazu behilft es sich einer interaktiven Zeichenfläche, welche aus gezeichneten Figuren Datenstrukturen erkennen und auswerten kann. Zusätzliche Funktionen sind das Stellen von Aufgaben und das automatische Überprüfen der gezeichneten Lösungen. Das Lösen von Aufgaben umfasst nicht nur das Zeichnen einer geforderten Datenstruktur, sondern auch die Beantwortung von Multiple Choice Fragen. Während das Überprüfen von Multiple Choice Antworten einfach ist, ist für die Überprüfung von Datenstruktur-Lösungen mit der geforderten Musterlösung eine Metrik entwickelt worden, welche eine Ähnlichkeit zwischen Musterlösung und der abgegebenen Lösung angibt.

In den vorausgegangenen Projektarbeiten wurde das Programm als Einzelplatzanwendung erweitert. Zusätzlich wurde der Grundstein für das kollaborative Bearbeiten einer Aufgabe in *algoria Worksheet* gelegt. Auf dieser Grundlage aufbauend legt diese Arbeit das Hauptaugenmerk auf die Möglichkeit der Realisierung eines verteilten Klassenzimmers. Die angebotenen Funktionalitäten umfassen das gleichzeitige Bearbeiten einer Aufgabe. Zusätzlich wird die Abhängigkeit vom lokalen Netzwerk gelöst und der Weg zu einer standortunabhängigen Kommunikation geebnet.

Für das gleichzeitige Bearbeiten einer Aufgabe liegt eine, auf dem Grundstein der vergangenen Projektarbeiten aufgebaute, Implementierung vor. Es werden konkrete Operationen definiert und deren Implementation besprochen. Mithilfe dieser Operationen kann ein Benutzer seine Änderungen an einer Aufgabe propagieren. Damit es keine Inkonsistenzen geben kann, die durch überschneidende Änderungen entstehen, ist ein Synchronisationsmechanismus auf der Basis von Operation Serialization implementiert worden. Damit eine Kollaboration auch unabhängig von einem gemeinsamen lokalen Netzwerk funktioniert, werden ein Internet-Server-Service und dessen Implementation präsentiert.

Abstract

Algoria Worksheet is an application which provides the ability to use the computer in the algorithm and data structure lectures in a useful and effective way to students as well as lectures. To achieve that it uses an interactive drawing board which is able to recognize and analyze data structures from drawn figures. Additionally, the application provides the functionality to create worksheets and the automatic correction of solution to these worksheets. The solving of tasks within a worksheet does not only contain the drawing of an asked data structure but also the answering of multiple choice questions. The correction of a multiple choice task is relatively easy to achieve. For the marking of drawing-based tasks a metric has been developed. This metric describes the similarity of the given sample solution to the solution a student provides when answering a question.

During the previous reports the application has been extended as a single-user system. Additionally the foundation for collaborative working was laid. Based on this foundation this reports main areas cover the development of a distributed classroom. This classroom includes functionalities such as concurrent editing of an exercise. Additionally the need of being in the same local area network is being removed. This leads to a communication which is independent from the location of the individual partners.

When changes coincide, there is the possibility of inconsistencies being caused. ,To prevent these from happening, a synchronization mechanism based on operation serialization has been implemented. To enable collaboration independent from a shared local area network a stand-alone service server and its implementation are being presented.

Inhaltsverzeichnis

1. Disposition

Die Disposition gibt in vier Seiten einen Überblick über die Ausgangslage und die Anforderungen des Projektes. In der Ausgangslage werden die bereits bestehenden Softwareteile vorgestellt, was den Einstieg in die Thematik und das Verständnis erleichtern soll. Die Anforderungsbeschreibung, welche danach beschrieben wird, enthält drei konkrete Fragestellungen, sowie die daraus abgeleiteten Anforderungen an die zu erweiternde Applikation.

1.1. Ausgangslage

In den folgenden Abschnitten wird die Ausgangslage dieses Projektes beschrieben. Da es sich um ein Fortsetzungsprojekt handelt, werden die Arbeiten, welche in den voraus gegangenen Projekten geleistet wurden, beschrieben. Das erste Projekt, welches in dieser Projektreihe steht, ist die Entwicklung einer interaktiven Zeichenfläche. Dies wird unter 1.1.1 behandelt. Das darauf aufbauende Programm *Algoria Worksheet* wird im Kapitel 1.1.2 beschrieben.

1.1.1. Algoria

Algoria ist eine interaktive Zeichenfläche, welche sich mit Vorteil mit einem Tablet-Computer bedienen lässt. Sie erlaubt das einfache Zeichnen von Datenstrukturen, welche durch die *Algoria*-Engine erkannt werden. Für diesen Schritt werden zuerst die gezeichneten Striche getrennt verarbeitet. Somit entsteht ein Bild aus einzelnen Strichen, welche nun weiter analysiert werden. Anhand von so genannten Ladder-Definitionen wird nun versucht, eine beschriebene Form zu finden. Solche Formen können z.B. Pfeile sein. Hier wird beschrieben, wie drei Striche zueinander stehen (Winkel zwischen den Strichen) und wo sie sich schneiden sollen (jeweils an einem Ende der jeweiligen Striche). Eine andere Definition wäre, wie ein Array gezeichnet werden soll. Hier werden die bereits erwähnten Attribute so gesetzt, dass sich 4 Striche zu einem Rechteck vereinen. Wenn die Applikation eine Datenstruktur erkannt hat, wird diese im Speicher nachgebildet.

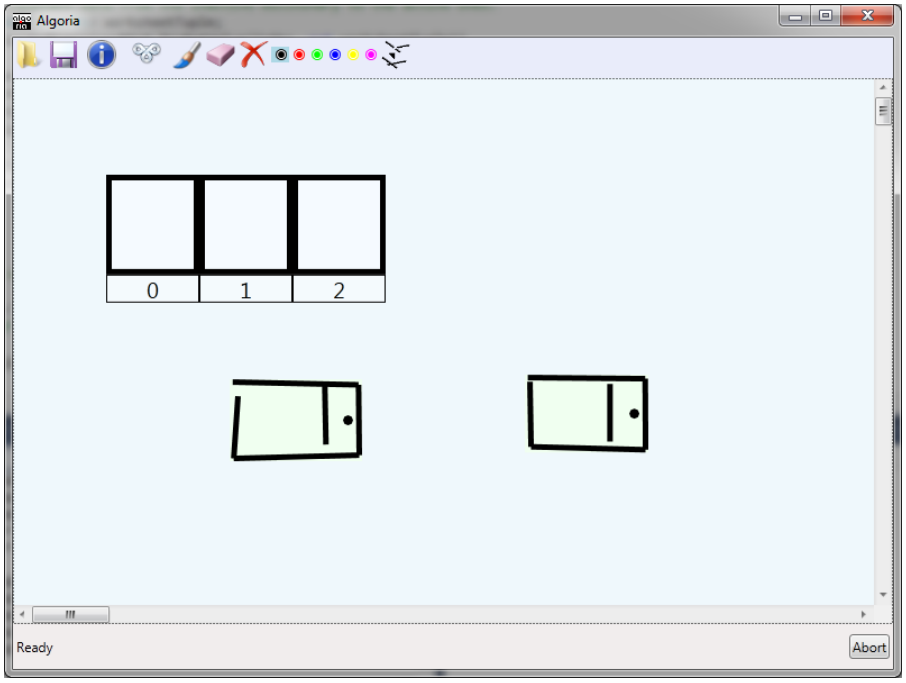


Abbildung 1.1.: Algoria

Die Abbildung 1.1 zeigt einen Screenshot von *Algoria*. In der Zeichenfläche sind bereits ein Array sowie zwei Linked-List Elemente erkannt worden. Weitere Funktionen, welche *Algoria* bietet, sind die folgenden:

- Vordefinierte Algorithmen auf den erkannten Datenstrukturen ausführen
- Verschieben von Datenstrukturen
- Ändern von Datenstrukturen (Hinzufügen und Löschen von Elementen innerhalb einer Datenstruktur)
- Manipulieren von Werten innerhalb einer Datenstruktur
- Speichern und Laden von Zeichenflächen

1.1.2. Algoria Worksheet

Algoria Worksheet wurde während der gleichnamigen Bachelor Thesis[1] geplant und entwickelt. Ziel dieses Programmes ist es, den Algorithmen und Datenstrukturen Unterricht dahingehend zu bereichern, dass der Computer eine sinnvolle Rolle übernehmen kann. So soll es Studierende während und neben dem Unterricht unterstützen. Dozierende haben die Möglichkeit, Aufgabenblätter, bestehend aus mehreren Aufgaben, zu erstellen. Diese Aufgaben können zum einen vom Typ Multiple Choice sein, zum anderen kann der Dozierende eine Musterlösung in Form von gezeichneten Datenstrukturen eingeben. Beim zweiten Typ müssen Studierende so nah wie möglich an die geforderte Musterlösung kommen. Damit Studierende möglichst schnell eine Rückmeldung zu ihrer Lösung erhalten, wurde eine automatische Lösungsprüfung in die Applikation integriert. Diese Rückmeldung besteht darin, dass dem Studierenden mitgeteilt wird, zu wie viel Prozent sich seine Lösung mit der Musterlösung deckt. Für die Berechnung dieses Prozentwertes wurde eine Metrik für die Messung der Ähnlichkeit zwischen zwei Datenstrukturen entwickelt. Die Metrik zählt die notwendigen Aktionen, welche notwendig sind, um aus einer gegebenen Datenstruktur (Studierendenlösung) eine Gesuchte (Musterlösung) zu machen. Mögliche Aktionen sind das Hinzufügen von zusätzlichen Elementen oder das Ändern eines Wertes in einem Element. Jede dieser Aktionen hat ein Gewicht. Das heisst, dass das Fehlen einer Datenstruktur schlimmer ist als wenn nur ein Wert in der Datenstruktur falsch ist. Mithilfe dieser Schritte können den Studierenden Tipps gegeben werden, wie sie ihre Lösung weiter verbessern können.

Für beide Aufgabentypen kann ein Aufgabentext mit Anweisungen für die Aufgabe eingegeben werden. Für das Einlesen von Datenstrukturen verwendet *Algoria Worksheet* die interaktive Zeichenfläche, welche bereits in *Algoria* realisiert wurde. Da die Anforderungen von Dozierenden und Studierenden an das Programm sehr verschieden sind, wurden zwei unterschiedlichen Versionen des Programmes entwickelt, die Studierenden- und die Admin-Version.

Die Abbildung 1.2 zeigt *Algoria Worksheet* in der Admin-Version mit einem erstellten Arbeitsblatt. Das Arbeitsblatt enthält zum jetzigen Zeitpunkt vier verschiedene Aufgaben. Mithilfe der linken Seitenliste kann, analog zu Power Point, zwischen den verschiedenen Aufgaben gewechselt werden. Die Seitenliste verfügt über Drag & Drop Funktionalitäten, um die Aufgabenreihenfolge verändern zu können. Multiple Choice Antworten können im unten aufklappbaren Bereich erfasst werden.

[1] http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P6_2011_Algoria_Worksheet.pdf[Frey11]

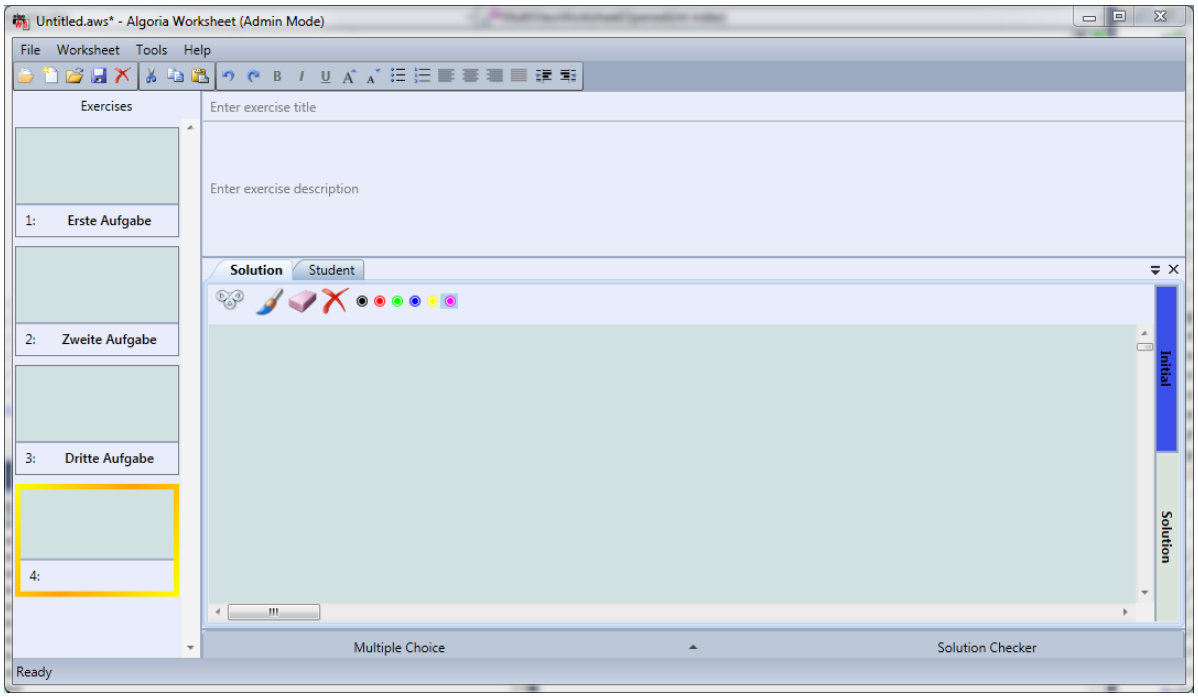


Abbildung 1.2.: Algoria Worksheet

Nebst den bereits beschriebenen Arbeiten während der Bachelor Thesis wurde die Applikation im Verlauf der Projektarbeit 7 mit dem Titel „P7: Algoria – Persönliche Lernumgebung“[2] weiterentwickelt. Hauptaugenmerk der Projektarbeit war die Verbesserung der automatischen Lösungskorrektur. Vor der Projektarbeit 7 wurde die Metrik mit fest eingestellten Gewichtungen für jeweils notwendige Aktionen erstellt. Nach der Projektarbeit haben Dozierende die Möglichkeit pro Datenstrukturtyp, welcher in einer Musterlösung vorhanden ist, Gewichtungen für einzelne Aktionen zu setzen. Eine weitere Änderung an der Lösungsprüfung ist das Verhalten bei mehreren Datenstrukturen in der Musterlösung und in der Studierendenlösung. Die Paarung zwischen den Datenstrukturen der Muster- und Studierendenlösung wird so erstellt, dass die resultierende Summe aus allen Metriken der gewählten Paare möglichst klein ist. Das Problem des Findens einer optimalen Paarung wird durch das Assignment Problem[3] beschrieben. Lösungsalgorithmen für dieses Problem können somit auch für das Finden von Paarungen zwischen Datenstrukturen angewendet werden.

Die darauffolgende Projektarbeit 8 mit dem Titel „P8: Algoria - Kollaboration im Schulzimmer“[4] hat die Applikation zusätzlich erweitert. Im Mittelpunkt der Arbeit stand die kollaborative Zusammenarbeit zwischen Studierenden und Dozierenden im Klassenzimmer. Zum einen ist es einem Dozierenden ermöglicht worden, mehrere Arbeitsblätter parallel darzustellen, ohne dazu eine weitere Instanz des Programmes zu öffnen. Zum anderen wurde die Applikation mit der Fähigkeit ausgestattet, Arbeitsblätter über das Netzwerk zu verteilen. Basierend auf dieser Netzwerkfähigkeit wurde ein Grundgerüst errichtet, welches die gemeinsame Bearbeitung einer Aufgabe ermöglichen soll.

[2] http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P7_2012_Algoria_Lernumgebung.pdf [Frey12a]
[3] http://www.me.utexas.edu/~jensen/models/network/net9.html [Jens12]
[4] http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P8_2012_Algoria_Lernumgebung.pdf [Frey12a]

1.2. Anforderungsbeschreibung

In [Frey12a] ist ein Grundgerüst entstanden, welches es erlaubt, eine Kollaboration im Klassenzimmer zu betreiben. Die gemeinsame Bearbeitung einer Aufgabe ist konzeptionell betrachtet worden. Um Synchronisationsproblemen vorzubeugen, sind verschiedene Ideen zur Sicherstellung der Konsistenz betrachtet und verglichen worden. In dieser Arbeit liegt das Hauptaugenmerk auf der Synchronisation der gleichzeitigen Bearbeitung einer Aufgabe. Zusätzlich sind, durch die Fixierung auf das physikalische Klassenzimmer, Abhängigkeiten vom lokalen Netzwerk entstanden. Diese Abhängigkeiten sollen aufgelöst werden. Die folgenden Fragen stammen aus der Aufgabenstellung[5] oder wurden aus ihr generiert. Während des Projektes sollen diese Fragen beantwortet werden.

- Wie realisieren Sie eine kollaborative Lernumgebung mit Synchronisationprimitiven aus der Basis von *Algoria Worksheet*?
- Wie erreichen Sie eine standortunabhängige Kollaboration, sodass von überall her an der Lösung einer Aufgabe mitgearbeitet werden kann?
- Wie kann erreicht werden, dass kurze Verbindungsunterbrüche zu keinen grösseren Einbussen in der Handhabung führen?

Diese Fragen ziehen konkrete Anforderungen an *Algoria Worksheet* nach sich. Die Tabelle 1.1 listet diese auf. Zusätzlich wird angegeben, welches Kapitel dieses Berichtes sich mit der Anforderung befasst.

Anforderung	Behandelndes Kapitel
Studierende können gleichzeitig an einer Aufgabe arbeiten. Mithilfe eines Synchronisationsparadigmas wird die Sicherstellung der Konsistenz gewährleistet.	Kapitel 2, Seite 8
Die Mitarbeit beim Lösen einer Aufgabe ist standortunabhängig möglich.	Kapitel 3, Seite 27
Wenn es zu einem Unterbruch in der Verbindung zum Server der Kollaborationssitzung kommt, ist es möglich, vereinfacht wieder der Sitzung beizutreten.	Kapitel 4, Seite 29

Tabelle 1.1.: Anforderungen an *Algoria Worksheet*

Nebst der Beantwortung der Fragen und der Umsetzung der Anforderungen soll während der Arbeit ein zum Thema passendes wissenschaftliches Paper erstellt werden. Dieses ist in Kapitel 2 eingefügt.

[5] Siehe Anhang A, Seite 36

2. Kollaborative Bearbeitung

Damit ein Dokument gemeinsam bearbeitet werden kann, bedarf es einiger Grundlagen. Zum einen ist es notwendig den gemeinsamen Zugriff gewähren zu können, zum anderen muss sichergestellt werden, dass der Zugriff synchronisiert wird. In den nächsten 8 Seiten wird ein, zu diesem Thema erstelltes, wissenschaftliches Paper eingefügt. Diese Arbeit beschreibt den Synchronisationsmechanismus Operation Serialization. Zudem gibt das Paper einen Einblick, wie Operation Serialization in Algoria angewendet werden kann und welche Voraussetzungen dafür gegeben sein müssen. Um den vorgestellten Synchronisationsmechanismus in *Algoria Worksheet* integrieren zu können wird die Implementation dieser Voraussetzungen in Kapitel 2.1 präsentiert. Abgeschlossen wird dieser Teil der Arbeit mit der Diskussion der Implementation des Synchronisationsmechanismus in *Algoria Worksheet* in Kapitel 2.2.

Operation Serialization in Algoria

Reto Frey and Christoph Stamm
University of Applied Sciences Northwestern Switzerland
reto.frey@students.fhnw.ch / christoph.stamm@fhnw.ch

***Abstract* — Interactive learning applications should allow concurrent processing done by different users in an intuitive and natural manner. To maintain consistency they need a concurrency control technique. In this paper we describe the problematic of concurrent editing of drawings and present our approach based on Operation Serialization in the context of an interactive drawing canvas that recognizes sketches of data structures. To find possible conflicts we built a conflict matrix which allows a fast check. The implementation of Operation Serialization in Algoria allows a concurrent editing of a drawing canvas without losing consistency. To prevent an unnecessary inflation of the history buffer we extended the range of existing conflicts with a same-effect-conflict.**

Keywords* — *Operation Serialization, consistency maintainance, educational collaboration, interactive drawing canvas, Algoria, Algoria Worksheet

I. INTRODUCTION

In an educational setting the computer became more important over the last years. For a long period of time, slides were presented with the help of an overhead projector and made up an essential part of education. Presentations played on a computer, shown with the help of a beamer started to steadily occupy this place. Today nearly every student owns a notebook and most have it with them when attending lectures. Algoria [9] is an application which originates in the algorithm and data structure classes during the study as computer scientist. This application is specifically designed to be used with tablet-pc's and works with inputs being made either by finger, by pen or by mouse. Algoria offers an interactive drawing canvas which is able to recognize sketches of data structures like arrays, lists or trees. For this recognition a set of geometrical figures is needed that describes the appearance of the different data structures. If a drawing on the canvas is being recognized as a data structure, a corresponding instance of this data structure is being created in the memory. These data structures can be modified by hand or by animated algorithms.

The application Algoria Worksheet is an extension of Algoria and is also used mainly in an educational setting [4]. Algoria Worksheets consist of multiple exercises where each exercise has an interactive drawing canvas available. To work on an exercise in an interactive worksheet the student is mostly asked to draw a data structure. When creating the exercises, teachers not only write an instruction text but also save a sketch of the sample solution. Based on this sample solution Algoria Worksheet is able to determine if a given answer is correct. Students can use this application to do their homework, recapitulate newly learned topics concerning data structures or algorithms, or prepare themselves for exams.

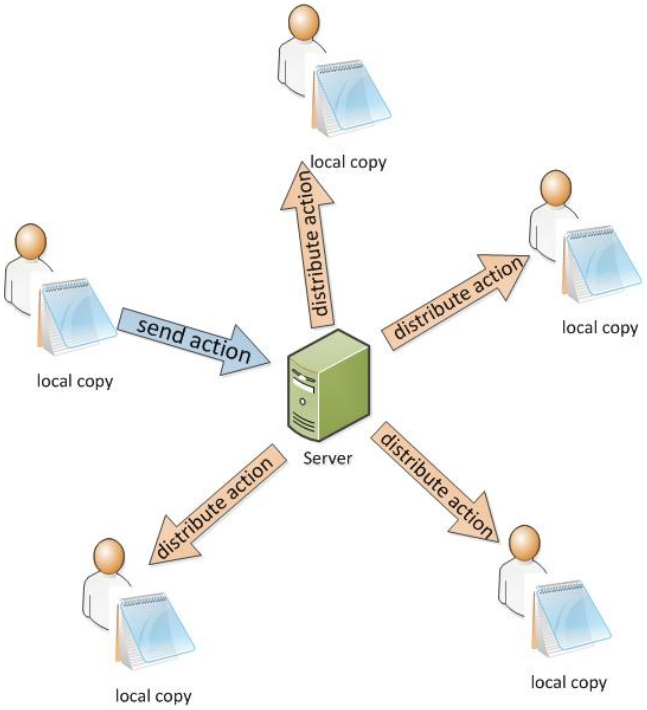


Figure 1: Collaboration infrastructure

When students want to work collaboratively, they sit together at a table, stand in front of a whiteboard, or they are spread over the world in case of using collaborative software [2][5][7][8][10]. They modify the content of a sheet of paper or draw sketches to a whiteboard concurrently. However, when they edit an electronic document, due to the limitations in access to the input devices of a notebook only one student is actively working. The other students are working passively by giving instructions or discussing. A network framework integrated in Algoria Worksheet allows multiple students to work concurrently on one exercise while sitting on the own personal computer. As shown in Figure 1, every user has its own local version of the worksheet. When a student edits the drawing canvas, the changes made are propagated to all other users taking part in the collaboration session. The network framework also allows the transfer of a worksheet. Thanks to the widely spread availability of fast internet connections, this collaboration does not stop when students leave the classroom.

II. Problematic of Concurrent Operations

Including a network framework into Algoria Worksheet and adding the possibility to concurrently edit a document transforms the application from originally being a single user system to a real-time collaborative editing system. This transformation contains the problematic of losing consistency when multiple users concurrently edit a document.

In the following we describe the problem of concurrent operations. In all the examples we discuss the same scenario. Two users at two different sites work on the same document at the same time. When collaborative work is done, this can lead to situations with or without conflicts. Situations with conflicts are defined by the following equations [3]:

Sites: $S_1, S_2,\ S_1 \neq S_2$
Operations: O_1 generated at S_1, O_2 generated at S_2
Situation without conflict: $O_2 \circ O_1 = O_1 \circ O_2$
Situation with conflict: $O_2 \circ O_1 \neq O_1 \circ O_2$

The usage of the ∘-sign indicates that there is a composition consisting of two operations. This usage is also being introduced in [3]. According to the definition $O_2 \circ O_1$ means that O_1 is being executed first and the result is being used as input for operation O_2. For the definition of a conflict-situation to be true, another assumption must apply: When the operation O_2 is being generated at S_2, operation O_1 must not have been arrived at S_2. The operations must overlap each other. Figure 2 shows the situation in a sequence diagram:

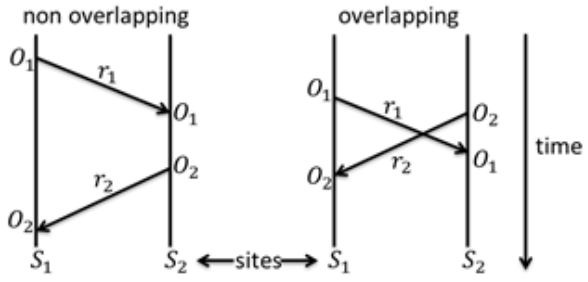


Figure 2: a) Non-overlapping and
b) overlapping operations from [3]

The non-overlapping case is no threat to the consistency. Before the generation of operation O_2 at S_2, the first operation O_1 coming from S_1 is already executed. The user at site S_2 has the same modified version of the document as the user at site S_1. When two operations overlap, the two users manipulate the same state at the same time. If the two operations result in a conflicting situation, this bares the problem of losing consistency as the order of execution is different at S_1 from site S_2.

There are multiple techniques that address the problem and help maintaining consistency either through resolving of conflicts or through preventing conflicts to happen in the first place. Three of these techniques are briefly presented in the following three sections.

A. Locking

The locking approach [1] restricts the access to a given object. If one user owns the lock for an object, all other users are not able to do any modifications to it. Concurrent modification in a document is only possible when users lock different object. After the lock has been returned, other users are allowed to lock the object and perform their modifications. This is the single one approach presented here that prevents conflicts from happening so there is no need to resolve them. If only one user can modify the document at a time the possibility of two concurrent edit actions is being removed. This can be done in different granularities from locking the whole canvas down to the locking of one data structure. Also a locking of a region is a possibility. Even if this strategy removes all concerns on consistency problems it has a major drawback: Without regard to the granularity, locking affects the responsiveness of the system.

B. Operation Transformation

A technique that does not build upon locking is operation transformation [11]. In this approach the main idea is to transform the incoming operation that it does not cause a conflict anymore.

$O_2 \circ O_1 \neq O_1 \circ O_2$
$O_1^T = T(O_1, O_2), O_2^T = T(O_2, O_1)$
$O_2^T \circ O_1 = O_1^T \circ O_2$

The core function of this approach is the transformation function T. The transformed action is of the same type but the parameters have been altered. Compared to the locking approach operation transformation has the advantage that it does not affect the responsiveness of the system. The disadvantage is that finding the transformation function is hard.

To convey the idea of operation transformation we examine the situation when two operations O_1 and O_2 concurrently edit an array. In Figure 3 an array is shown where the range marked in blue indicates which elements are affected when moving the element at position 1 to position 3. The red arrow marks the insertion point of the insert operation at index 1.

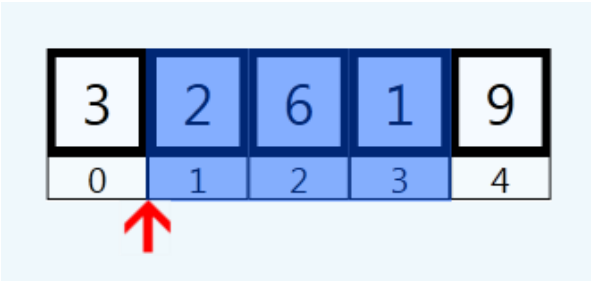


Figure 3: Concurrent insert and move operation

In this example O_1 denotes the move operation, O_2 labels the insertion operation. If O_2 is executed first, the move operation needs to be modified in order to still move the correct element. After 2 is executed the elements behind

position 1 are all moved back by one position. Therefore both, the index of the element to move and the target index of O_2 need to be incremented by one. If O_1 = move(1,3) then O_1^T = move(2,4). In this context move(1,3) means the element at position 1 is moved to position 3.

C. Operation Serialization

Another idea that does not lock and therefore does not affect the responsiveness is Operation Serialization [6]. The basic idea behind this approach is to find a happened-before relationship between all operations that have been executed. Based on these relationships one tries to find a linear order of operations where the relationships are not being broken. When a new operation arrives at a site, it is being inserted into the list of operations. The position of the insertion is being determined by the happened-before relationships the newly arrived operation has. The main part of Operation Serialization is a method that inserts an arrived operation into a list of already executed operations (history buffer).

Current history buffer: $L_t = [O_0, O_1, O_2]$
Newly arrived operation: O_{new}
Insert function: $L_{t+1} = \mathrm{I}(L_t, O_{new})$
so that $O_{new} \in L_{t+1}$

In [6] the determination of the insert position is being reduced to a graph-problem. To determine the position, two directed graphs are constructed and updated during the collaborative editing process. These two graphs are called real-conflict-graph (G_{RC}) and serialization-graph (G_S), respectively. Every operation being made locally or received is represented through a node in both graphs. When a new operation arrives, it is added to G_{RC} and G_S and compared to each operation in the history buffer. Whether there are edges between two nodes is depending on the type of conflict they have. After topologically sorting G_S a new history buffer which contains the newly arrived operation is resulting. H denotes the operation coming from the history buffer, O is the operation to be inserted into the history buffer. [6] describes four types of conflicts:

- *No conflict:* No order between H and O exists therefore there is no edge in G_{RC} or G_S($O \circ H = H \circ O$);
- *Right order conflict (C_{right}):* O has to be executed before H. The edge goes from O to H;
- *Reverse order conflict ($C_{reverse}$):* H has to be executed before O. The edge goes from H to O;
- *Realconflict (C_{real}):* H and O cannot be executed concurrently. The direction of the edge depends on the priority of each operation. The priority of the operations is being left to the implementation.

If two operations are concurrent (overlapping) they have an edge in G_S when either a C_{right} or a $C_{reverse}$ conflict between them exists. In case of a C_{real} conflict the edge is added to G_{RC}. When they are not concurrent, it is being checked whether one operation depends on or if one operation precedes another operation. Operation O_2 is said to be dependent on O_1 if O_2 needs the outcome of O_1 to work correctly, and O_1 precedes O_2, if O_2 is being generated after O_1 has been executed. For each depends-on and precedes relation a corresponding edge is added to G_S.

After all necessary edges are added all nodes with incoming edges in the G_{RC} are being deleted. Also every operation that depends on an operation that has been deleted is being removed. After that the nodes of G_S are topologically sorted and stored in L_{t+1}. O_{new} is an element of L_{t+1}. If O_{new} is not inserted at the end of L_{t+1}, each operation located behind O_{new} has to be undone and O_{new} can be executed. Then all operations behind O_{new} must be redone

III. Operation Serialization in Algoria

In the next section we describe the process that runs through when a user edits the interactive drawing canvas in Algoria during a collaboration session, to identify all steps that are needed in order to maintain consistency. Then we present the most important parts that have been implemented in Algoria to allow Operation Serialization to work.

A. Edit operation propagation sequence

The sequence diagram in Figure 4 describes the necessary steps that are executed when a user edits the interactive drawing canvas. The scenario includes three different sites where each site is working on the same document.

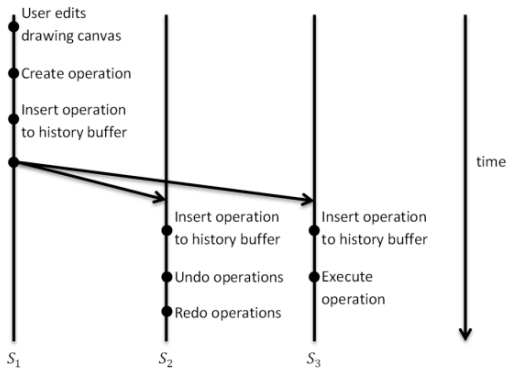


Figure 4: Edit operation propagation

The user at site S_1 performs a change in the local version of the document. Then an operation representing that change is created and inserted into the history buffer by applying the Operation Serialization algorithm. The same operation is sent across the network to all other users taking part in the collaboration session and each of them inserts the received operation into his or her local history buffer. S_2 performs then the necessary undo and redo operations, while at S_3 the received operation is inserted at the end of the history buffer, and therefore no undo or redo is required at S_3.

B. Defining Operations

The step immediately after a change has been made to the drawing canvas is the creation of an operation that can be inserted into the history buffer. There are lots of operations

implemented in Algoria for this purpose. Even though all these operations represent a different change made to the drawing canvas they all share a common basis:

- *Redo method:* The redo method is being used to execute the change;
- *Undo method:* The undo method is being used to undo the change;
- *Target Guid*: A globally unique identifier which defines the target of the operation.

In Algoria two different groups of operations exist. The first group consists of operations that are not depending on the type of data structure the operation modifies (TIO), and the second groups operations are type dependent (TDO). The most interesting operations in the first group are:

- *AddDatastructure(ds):* Adds data structure *ds* to the drawing canvas;
- *MoveDatastructure(guid, pos1, pos2):* Moves the data structure specified by guid from position *pos1* to *pos2*;
- *RemoveDatastructure(guid):* Removes the data structure specified by *guid*.

In the second group the operations target a specific type of data structure. Typical operations like adding, inserting, removing and value-setting of an element are available in all different data structures.

Array

- *MoveElement(guid, from, to):* Moves the element at position *from* to position *to* in the array specified by *guid*.
- *SwapElements(guid, a, b):* Swaps the element at position *a* with the element at position *b* in the array specified by *guid*.

List

- *MoveElement(guid, from, to):* Moves the element at position *from* to position *to* in the list specified by *guid*.
- *MergeLists(guidA, guidB):* Merges the list specified by *guidA* with the list specified by *guidB*. The resulting list is again specified by *guidA*.

Tree

- *RotateNodes(guid, root, pivot):* Rotates node *pivot* around node *root* in the tree specified by *guid*. The position of *pivot* relative to *root* determines whether this is a right or a left rotation.
- *PruneBranch(guid, node):*Prunes the branch with root *node* in the tree specified by *guid*.

After the creation of an operation object, this directly gets inserted into the history buffer and then sent to all other sites of the collaboration session. Therefore every operation needs to be serializable. When inserting a new operation into the history buffer two directed graphs G_{RC} and G_S are being used to determine the position of the new operation.

C. Building the Serialization and Real Conflict Graphs

To show the process of building up G_{RC} and G_S we use the example of two sites S_1 and S_2 editing an array concurrently. To do so each site generates operations. At site S_1 the Operations O_1 and O_2 are created whereas O_3 and O_4 are generated at S_2. O_1 denotes an operation setting the value of the element with index 4 to '3', O_2 moves the element at postition 2 to position 4, O_3 deletes the element at position 1 and O_4 swaps the elements at positions 4 and 6. We examine the construction of both graphs at S_1.

First we specify the initial state the two graphs have before the new operations are generated. In Figure 5 this state of both graphs is shown. The operation O_0 is already in the history buffer and therefore in the graphs. G_{RC} does not contain any edges so no real conflicts exist.

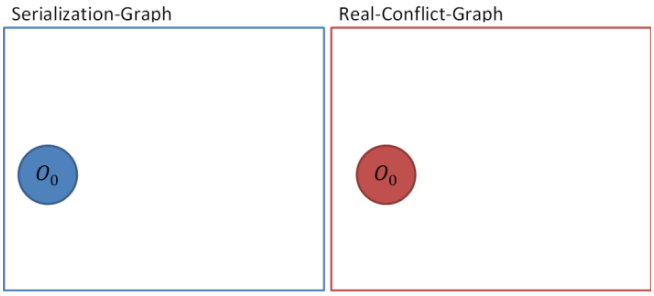


Figure 5: Initial state of G_S and G_{RC}

The insertion of O_1 and O_2 does not yield a problem because they are generated locally at site S_1. Figure 6 shows the state of G_S and G_{RC} after the integration process.

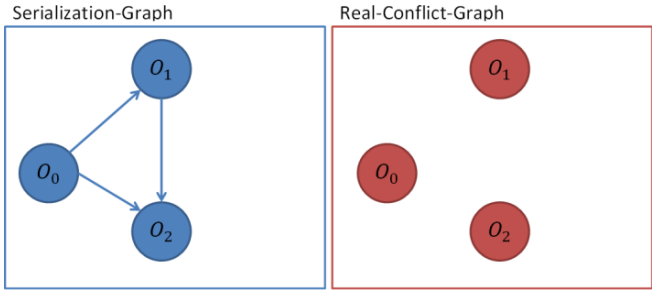


Figure 6: G_S and G_{RC} after adding O_1 and O_2

Because O_0 was already executed when O_1 is created, O_0 precedes O_1 and the two nodes are connected through an edge. In contrast to G_S G_{RC} does still not contain an edge. Therefore, no C_{real} exists at this point.

When O_3 and O_4 are received, they are also added to G_S and G_{RC}. In Figure 7 both graphs contain the received operations. According to the edge from O_4 to O_2 in G_{RC} these two operations cause a C_{real} conflict.

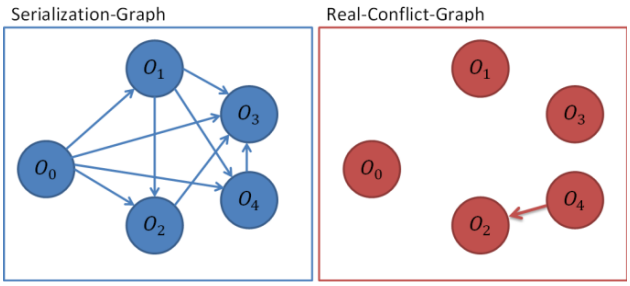


Figure 7: G_S and G_{RC} after adding O_3 and O_4

Next we present how the comparison of two operations is being done in Algoria to check for the existence of conflicts.

D. Creating the conflict matrix

According to [6] there are four possible outcomes if two operations are checked for the existence of a conflict between. One outcome is that no conflict exists and therefore no edge has to be added between the two operations in either graph. If a conflict exists, the type of the conflict defines the direction of the edge and the graph where the edge has to be added. Since Algoria describes two main groups of operations we examine operations coming from the first in combination with the second group. In all the following tables H denotes the operation coming from the history buffer whereas O represents the operation to integrate into the history buffer. Additionally one condition must hold true: H and O target the data structure.

A comparison of three TIO operations with the group of TDO is shown in Table 1. Conflicts between a data structure type independent operation and one out of the group of operations that depend on the type of data structure are always the same. What operation represents the data structure type dependend group is not an issue.

Table 1: Conflicts between data structure type independent and data structure type dependent (TDO) operations

O \ H	AddDs	MoveDs	RemoveDs	TDO
AddDs	C_{real}	C_{right}	C_{right}	C_{right}
MoveDs	$C_{reverse}$	C_{real}	C_{right}	No
RemoveDs	$C_{reverse}$	$C_{reverse}$	C_{real}	$C_{reverse}$
TDO	$C_{reverse}$	No	C_{right}	Table 2

Additionally Table 1 shows the conflicts that appear when the operations in the type independend group are compared. Except for the target of the operation other parameters of the operation do not influence the resulting conflict. The comparison between operations in the TDO group is described in Table 2.

After the two main groups of operations have been compared we now expand the data structure type dependent operations into groups for each type they target. In Table 2 we can see the comparison between operations in the TIO group and operations targeting an array, a list or a tree. With the exception of the TIO group which is described in Table 1 operations compared from different groups never result in a conflict.

Table 2: Conflicts between operation groups

O \ H	TIO	Array	List	Tree
TIO	Table 1	Table 1	Table 1	Table 1
Array	Table 1	Table 3	No	No
List	Table 1	No	…	No
Tree	Table 1	No	No	…

To fully complete the conflict matrix we now show a small part of the table that compares two array-modifying operations. For list and tree operations a similar table exists. In opposition to the comparisons we have shown in the previous tables now the type of conflict is not only dependent on the type of operation but also on the parameters used. Table 3 shows a pullout of the comparison matrix of the data structure array.

Table 3: Conflicts between array operations

O \ H	Insert	Move	Swap
Insert	C_{right} / $C_{reverse}$ / C_{real}	$C_{reverse}$ / No	$C_{reverse}$ / No
Move	C_{right} / No	C_{real} / No	$C_{reverse}$ / C_{real} / No
Swap	C_{right} / No	C_{right} / C_{real} / No	C_{real} / No

When two operations in Table 3 are compared there are multiple types of conflicts that are possible. To explain how the correct type is determined we present two different situations and illustrate what condition must be fulfilled for which type of conflict. For both examples we have the same setup containing two sites S_1 and S_2. The two combinations of operations are Move- Insert and Move-Swap.

We first examine the case at site S_1 where O is a move operation and H represents the insertion of a new element into the array. Whether this leads to a C_{right} conflict or to no conflict at all depends on the parameter of these two operations. Let us look at an example of this combination. Figure 8 shows an array containing five elements that is the target of both operations.

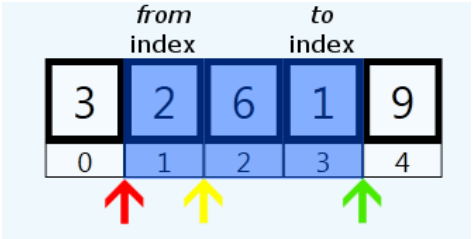


Figure 8: Possible insertion points for a new element relative to a range affected by a move operation.

A user at site S_2 generates the operation to move the element at index 1 to index 3, so all elements in between of these two indices will change their positions as well. There are three possible insertion positions relative to the range R defined by the move operation:

- *Red arrow*: The insertion point is in front of R. After a new element has been inserted into the array both the *from*-index and the *to*-index of the move operation point to new elements. Therefore, the move operation has to be executed before the insert operation. The result of this is a C_{right} conflict;
- *Yellow arrow*: The insertion point is in R. When the new element is inserted between the *from*-index and the *to*-index, one of those indexes will not point at the element which was initially selected for the move operation. Hence, the move operation has to be executed before the insert. The result here is also a C_{right} conflict;
- *Green arrow*: The insertion point is behind R. No conflict results as the equation $O \circ H = H \circ O$ is fulfilled.

We now change the site on which we encounter the conflict between the previous move and insert operation. At site S_2 O represents the insertion of a new element into the array and H denotes a move operation. When these two operations are compared at this site the resulting conflict changes. According to Table 3 a $C_{reverse}$ conflict can happen. Again this depends on the position of the insertion relative to the range the move operation uses. In this case when the insertion point is in front of (red arrow) or within (yellow arrow) the range of the move operation, the result would be a $C_{reverse}$. If the insertion point is behind the range of the move operation, no conflict results.

In contrast to the preceding example the combination of a swap and a move operation can lead to two different conflicts. In this case at site S_1 O is a move operation and H represents a swap of two elements. In Figure 9 the state of an array can be seen before either O or H are executed.

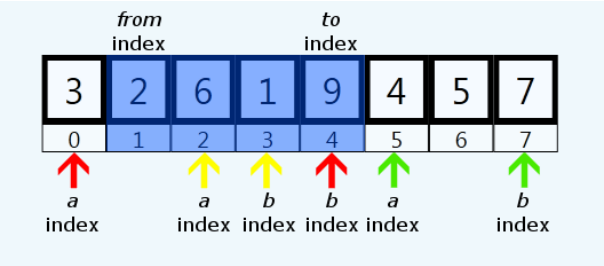


Figure 9: Possible swap indices relative to a range affected by a move operation.

Again three different scenarios are possible:

- *Red arrows:* One of the two indexes of the swap operation is inside the range of the move operation whereas the other is outside of this range. This leads to a C_{real} because it is not possible to combine the desired outcome of both operations. The same result is achieved when both swap indexes are inside the range of the move operation but either the *a*-index or the *b*-index is equal to either the *from*-index or the *to*-index;
- *Yellow arrows:* Both swap indexes are fully inside the range of the move operation, meaning that neither the *a*-index nor the *b*-index is equal to the *from*-index or the *to*-index. In this case the swap operation needs to be performed first. If the move operation was executed before swapping the elements, the targets of the *a*-index and of the *b*-index would be different. Therefore a $C_{reverse}$ conflict is being generated;
- *Green arrows:* Both targets of the swap operation are outside of the range of the move operation, whether they are in front of or behind the range does not matter. The order of execution does not change the result, therefore no conflict is generated;

In addition to the four types of conflict described in [6] we describe a new type of conflict, the same-effect-conflict (C_{se}). Since during the integration of a new operation a check with every operation in the history buffer is performed, the size of the history buffer becomes crucial. The C_{SE} conflict allows the discarding of operations with the same effect to prevent an unnecessary inflation of the history buffer.

There are two different possibilities for two operations have the same effect. The first one is very intuitive. Two operations are said to cause the same effect if they have the same type, are concurrently generated, overlap and all operation parameters are equal. An example of this is given by the operations O_1 and O_2:

O_1 = Move(3, 5, guid)
O_2 = Move(3, 5, guid)

The other way two operations can have the same effect is when they have the same result. Let us analyze a move and a swap operation.

O_3 = Move(1, 2, guid)
O_4 = Swap(1, 2, guid)

Here the type of the operations is different. Moving an element by only one position is equivalent to a swap of two neighbor elements in an array. When two operations cause a C_{se}, conflict the operation that has been received can be discarded and does not need to be integrated to the history buffer. In fact if two operations are causing the same effect, there are situations where the discarding of the received operation does not only prevent the inflation of the history buffer but is mandatory. I.e. if the swap operation is executed twice, the original state is being restored and the initially planned change is without any effect.

E. Real Conflict Removal and Topological Sort

In case of a C_{real}, the next step is to resolve this conflict. To achieve this, all nodes containing an ingoing edge in G_{RC} are being removed in both graphs. In addition every node that has a depends-on relationship to a removed node is also removed. In Figure 10 we see the result of the resolution of the C_{real} conflict in Figure 7. The node representing O_2 is deleted because the swap operation O_2 had higher priority. This

priority can be defined for each type of operation. In this example the priorities are randomly defined.

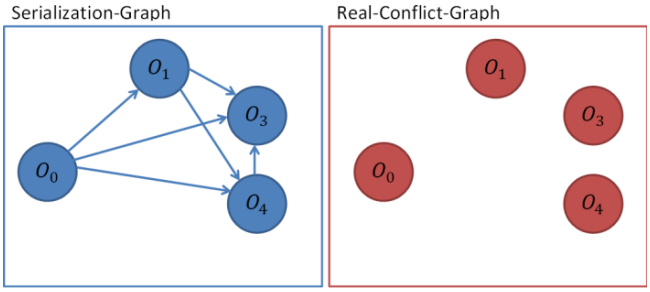


Figure 10: G_{RC} and G_S after resolving real conflicts

To obtain a sorted list of operations we topologically sort G_S. If multiple nodes have an indegree of 0, the operation with the earliest appearance in the current history buffer is added next into the sorted list.

F. *Undo / Redo*

After the topological sort the operation O_{new} is inserted in the history buffer at the correct position so that consistency is ensured. If O_{new} was added at the end of the history buffer, it can be safely executed. If not, all operations placed behind O_{new} need to be undone. Then O_{new} can be executed and all undone operations can be redone. This can lead to a big list of operations that have to be undone and redone immediately.

Algoria allows the visualization of multiple animated algorithms. The execution of an algorithm can cause the generation of a large number of animated operations. Because these operations are generated in a large quantity a slow execution of this undo/redo function is resulting. Also the duration of an animation can be long so an undo and redo of multiple animated operations is time consuming. We counteract this behavior by only starting an animation when it is desired. When a student starts a sorting algorithm, this is being shown with the animation. But instead of adding every change the algorithm causes in the data structure into the history buffer an AlgorithmExecuted operation is generated. This contains all sub-operations the algorithm causes. When undoing and redoing this type of operation, this operations allows a normal mode, which includes animations and a fast mode that causes immediate effect without any animation.

IV. RESULTS

After discussing all parts of the Operation Serialization algorithm we describe a scenario of a collaborative manipulation of an array in Algoria. The scenario includes two editing sites S_1 and S_2, both starting in the same initial state. Three operations which edit the same array are used:

- O_0 = Move(2, 4)
- O_1 = SetValue(4, '3')
- O_2 = Delete(1)

S_1 generates O_0, S_2 concurrently generates O_1 and O_2. In Figure 11 the different states displayed at sites S_1 and S_2 are shown in the left and right column, respectively. The history buffer is also shown below the individual arrays.

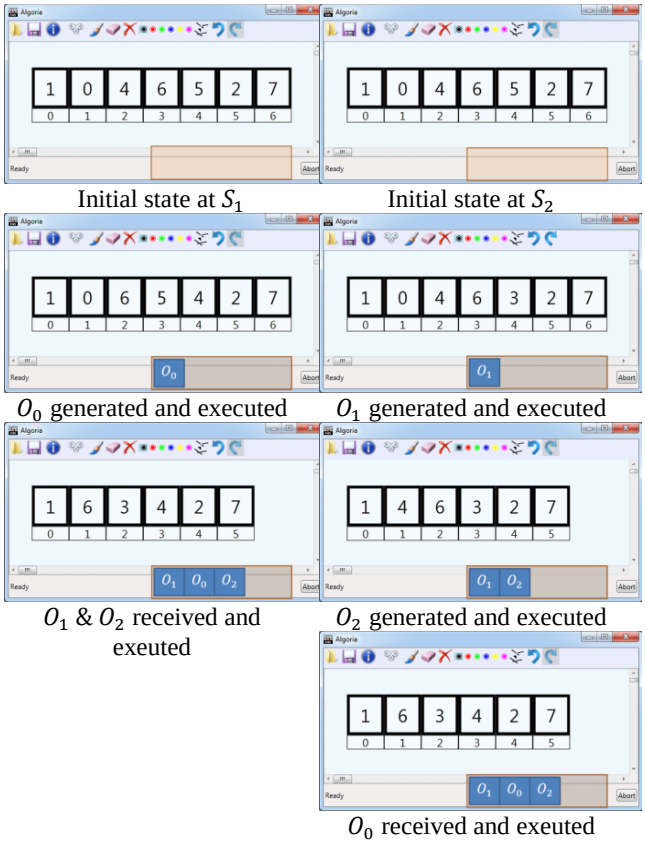


Figure 11: Concurrent editing of an array in Algoria

In the left column we see that the move operation has been completed, then O_1 and O_2 are received, integrated to the history buffer and executed. Additionally, the right column shows first the manipulating of the value at index 4 and then the deletion of the element at index 0. Then O_0 is received, inserted in the history buffer and executed.

What cannot be seen at both sites is the resolving of the conflicts. Figure 12 therefore visualizes the work being done in the background.

Figure 12: Visualization of operations being executed during concurrent manipulations.

Elements that have changed are marked in red. First we discuss S_1 on the left site. After O_o has been executed, O_1 arrives. Between O_0 and O_1 a C_{right} conflict exists so O_1 is placed in front of O_0 in the history buffer. O_0 is being undone, then O_1 is executed and O_0 is being redone. Now O_2 arrives and both operations in the history buffer have a $C_{reverse}$ conflict with the received operation. Because of that, O_2 is being inserted at the end of the history buffer and can be executed without undoing any other operation. At site S_2 the local operations are being executed and then O_0 is received. The comparison of O_2 with O_0 yields a C_{right} conflict and O_0 is being inserted between O_1 and O_2 in the history buffer. Therefore O_2 needs to be undone, O_0 is executed and O_2 is being redone. Now both sites have the same array and consistency is ensured.

Operation Serialization has been implemented into Algoria Worksheet based on the observations of this paper. Multiple students can work on a worksheet concurrently and edit the same drawing canvas at the same time. With the help of a static conflict matrix conflicts are found between operations. Possible extensions of the collaboration framework in Algoria Worksheet are a dynamic conflict checker based on annotations or a more efficient way to detect overlapping operations.

REFERENCES

[1] Chen, D. and Sun, C. *Optional Instant Locking in Distributed Collaborative Graphics Editing Systems*. In Proceedings of *International Conference on Parallel and Dirstributed Systems* (ICPADS 2011), P. 109-116, 2001

[2] Codoxware Website. [Online]. Available: http://www.codoxware.com, February 2013

[3] Ellis, C.A. and Gibbs, S.J. *Concurrency Control in Groupware Systems*. In Proceedings of ACM SIGMOD *Conference on Management of Data* (SIGMOD '89), P. 399 – 407, 1989.

[4] Frey, R. Zogg, K. *IP6 Algoria Worksheet*, *Bachelor Thesis*, 2011 (http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P6_2011_Algoria_Worksheet.pdf)

[5] Google Docs Website. [Online]. Available: https://docs.google.com, February 2013

[6] Ignat, C.-L. and Norrie, M.C. *Draw-Together: Graphical Editor for Collaborative Drawing*. In Proceedings of *International Conference on Computer Supported Cooperative Work* (CSCW'06), P. 269 – 278, 2006

[7] Lautamäki, J. Nieminen, A. Koskinen, J. Aho, T. Mikkonen, T. Englund, M. *CoRED: browser-based Collaborative Real-time Editor for Java web applications.* In Proceedings of *ACM 2012 conference on Computer Supported Cooperative Work* (CSCW '12), Pages 1307-1316

[8] Lucidchart Website. [Online]. Available: https://www.lucidchart.com, February 2013

[9] Schweizer, R. Stamm, C. Walti B. *Algoria: Tablet-PC Anwendung für den Informatikunterricht. IMVS Fokus Report*, P. 18 – 26, 2010 (http://www.fhnw.ch/technik/imvs/publikationen/artikel-2010/algoria-tablet-pc-anwendung-fuer-den-informatikunterricht)

[10] VSAnywhere Website. [Online]. Available: https://vsanywhere.com/default.aspx, February 2013

[11] Weihai Yu, *Constant-Time Operation Transformation and Integration for Collaborative Editing.* In Proceedings of *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing* (CollaborateCom'11), P. 258 – 267, 2011

2.1. Kollaborationsvoraussetzung in Algoria Worksheet

Durch die genaue Beschreibung von Operation Serialization haben sich einige Voraussetzungen herauskristallisiert, welche in Algoria Worksheet gegeben sein müssen. Wie diese Voraussetzungen aussehen, wird in Kapitel 2.1.1 analysiert. Danach wird auf einige Implementierungsdetails eingegangen.

2.1.1. Analyse / Konzeption der Implementation

Für die Integrierung eines Synchronisationsalgorithmus müssen drei Voraussetzungen gegeben sein.

- Operationen müssen definiert werden, welche eine Änderung an einer Zeichnungsfläche beschreiben.
- Es muss ein Verlauf der gemachten Änderungen gespeichert werden.
- Die gemachten Operationen müssen über eine Netzwerkverbindung zu allen Teilnehmern der Kollaborationssitzung verteilt werden.

In [Frey12b] ist *Algoria Worksheet* bereits für das kollaborative Bearbeiten einer Aufgabe vorbereitet worden. Dazu gehört auch die Implementierung eines Netzwerkframeworks für das Versenden von Arbeitsblättern und das Propagieren von Änderungen an einer Aufgabe. Um die Änderungen propagieren zu können, sind bereits zwei Operationen testweise implementiert worden. Es handelt sich um das Hinzufügen und das Löschen einer Datenstruktur. Vollkommen neu ist nur die Anforderung, eine Liste mit dem Verlauf der gemachten Änderungen nachzuführen. Diese Voraussetzung hat Einflüsse sowohl auf die Operationen, als auch an das Netzwerkframework. Daher wird das Führen einer Verlaufs-Liste zuerst betrachtet.

Für die Konzeption einer Verlaufsliste muss zuerst geklärt werden, wo diese zur Verfügung gestellt wird und welche Operationen diese berücksichtigen soll. Während einer Kollaborationssitzung bearbeiten mehrere Benutzer in einer lokalen Instanz von *Algoria Worksheet* ein Arbeitsblatt. Die Verlaufsliste könnte somit alle Änderungen in einer *Algoria Worksheet*-Instanz einschliessen. Ein Arbeitsblatt besteht jedoch aus mehreren Aufgaben, welche jeweils über eine Zeichenfläche verfügen. Solange jedoch Benutzer nicht an derselben Aufgabe arbeiten, ist eine Synchronisierung des Zugriffs nicht notwendig. Somit ist eine globale Verlaufsliste pro Instanz nicht notwendig und die Einbindung pro Aufgabe bietet sich an. Da der einzige editierbare Teil einer Aufgabe die interaktive Zeichenfläche *Algorias* beinhaltet, kann die Verlaufsliste direkt in *Algoria* integriert werden. In *Algoria* wurde das Konzept einer Verlaufsliste bereits teilweise realisiert. Wie in Abbildung 2.1 zu sehen, wird diese Liste jedoch pro Datenstruktur geführt.

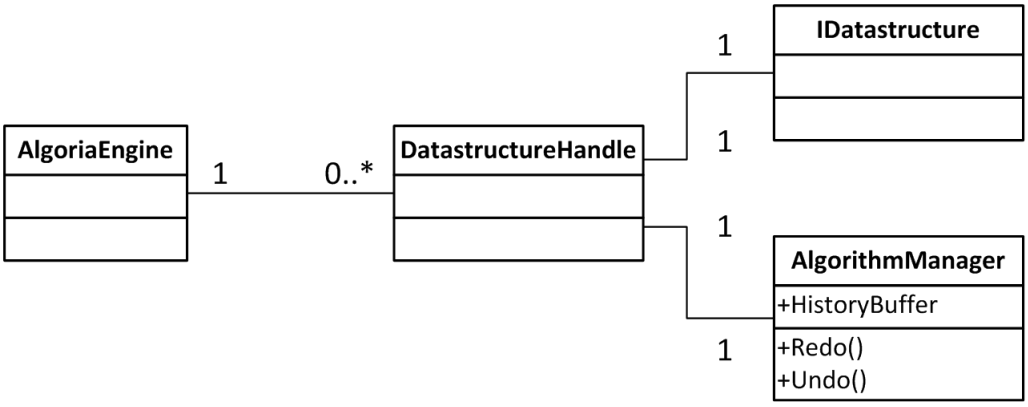


Abbildung 2.1.: Klassendiagramm mit ursprünglicher Verlaufslistenposition

Die *Algoria*-Engine verfügt via DatastructureHandles über eine Verbindung zu den Datenstrukturen in der Zeichenfläche. Jede dieser Datenstrukturen verfügt über einen AlgorithmManager, welcher eine Verlaufsliste der Datenstruktur führt. Diese Verlaufsliste ist nicht öffentlich zugänglich und neue Änderungen werden direkt von der Datenstruktur dem AlgorithmManager gemeldet. Operationen, welche ein Hinzufügen oder das Löschen einer Datenstruktur zur Folge haben, können in so einem Modell nicht realisiert werden. Beim Löschen einer Datenstruktur wird auch der AlgorithmManager terminiert. Daher ist eine Verlaufsliste notwendig, welche alle Änderungen an der Zeichenfläche beinhaltet.

Die Neugestaltung der Verlaufsliste beeinflusst auch die Operationen, welche in dieser Liste geführt werden. Für die Realisierung des Operation-Serialization-Algorithmus müssen die Folgen einer Operation in der Verlaufsliste rückgängig gemacht und wiederhergestellt werden können. Die bereits in *Algoria* implementierten Operationen sind eng an die jeweilige Datenstruktur gekoppelt. Diese Kopplung wird jedoch gelöst, wenn die Verlaufsliste nicht mehr in der Datenstruktur geführt wird. Somit muss eine Operation neu über die folgenden Informationen verfügen:

- Undo Methode
  Die Undo Methode macht die Änderungen einer Operation wieder rückgängig.
- Redo Methode
  Wenn eine Operation mit der Undo Methode rückgängig gemacht wurde, kann mithilfe der Redo Methode die Änderung wiederhergestellt werden.
- Datenstruktur-Guid
  Der 'globally unique identifier' einer Datenstruktur identifiziert das Ziel der Operation.
- Parameter der Operation
  Die Parameter enthalten zusätzliche Informationen, welche für das Ausführen der Operation notwendig sind.

Wenn eine Operation über die aufgeführten Informationen verfügt, kann sie theoretisch an die anderen Teilnehmer einer Kollaborationssitzung versendet werden. Dazu muss aber eine weitere Voraussetzung gegeben sein: Die zu versendenden Operationen müssen mithilfe der Windows Communication Foundation (WCF)[1] versendet werden können. Objekte, welche in WCF versendet werden, dürfen jedoch keine Funktionalität beinhalten. Solchen DataContract-Klassen ist es nur erlaubt, serialisierbare Felder zu enthalten. Somit müssen alle Informationen, welche eine Operation benötigt um die gewünschte Funktion erfüllen zu können, in ein Parameter-Objekt verpackt werden.

[1] http://msdn.microsoft.com/en-us/library/ms731082.aspx[Micr12]

Das Netzwerkframework, welches in [Frey12b] implementiert wurde, verfügt über ein sehr überschaubar gehaltenes Interface. Für das Versenden von Operationen – Anfragen eines neuen Arbeitsblattes oder Propagieren von Änderungen – gibt es eine Methode, die SendAction benannt ist. Mit dem Einbinden des Versendens von Operationen aus der Verlaufsliste muss auch die SendAction Methode überdacht werden. Es ist ersichtlich, dass es zwei Arten von Operationen gibt, welche versendet werden müssen. Zum einen sind dies administrative Aufgaben (das Anfordern eines Arbeitsblattes und die Antwort darauf, sowie eine Meldung, welche das Herunterfahren des Servers mitteilt). Die andere Gruppe besteht aus Operationen, welche eine Zeichenfläche verändern. Da diese Gruppen nur geringfügig Gemeinsamkeiten aufweisen, ist es sinnvoll, für jede der Gruppen eine Methode auf dem Server anzubieten. Dies erweitert auch das Callback-Interface auf der Clientseite um eine Methode.

2.1.2. Ausführungen zur Implementation

Nach der Analyse der Anforderungen, die Operation Serialization an *Algoria Worksheet* stellt, wird im Folgenden die Implementation besprochen. Hierbei liegt der Fokus auf der Verlaufsliste, den Operationen und der Kollaborationsinfrastruktur.

Verlaufsliste

Das Ziel, eine einzige Verlaufsliste in *Algoria* zu führen, führt zum Verlagern der Verantwortlichkeit über diese Liste. Bisher wurde die Liste in der Datenstruktur vom AlgorithmManager verwaltet (siehe Abbildung 2.1). Abbildung 2.2 zeigt, dass die *Algoria*-Engine auch über eine Verlaufsliste verfügt. Die Verlaufsliste des AlgorithmManagers wird jedoch nicht gelöscht. Der Grund dafür ist, dass Operationen in der globalen Verlaufsliste ohne Animation ablaufen sollen. Wird jedoch ein Algorithmus visualisiert, könnten mithilfe dieser AlgorithmManager-Verlaufsliste die Zwischenschritte animiert dargestellt werden.

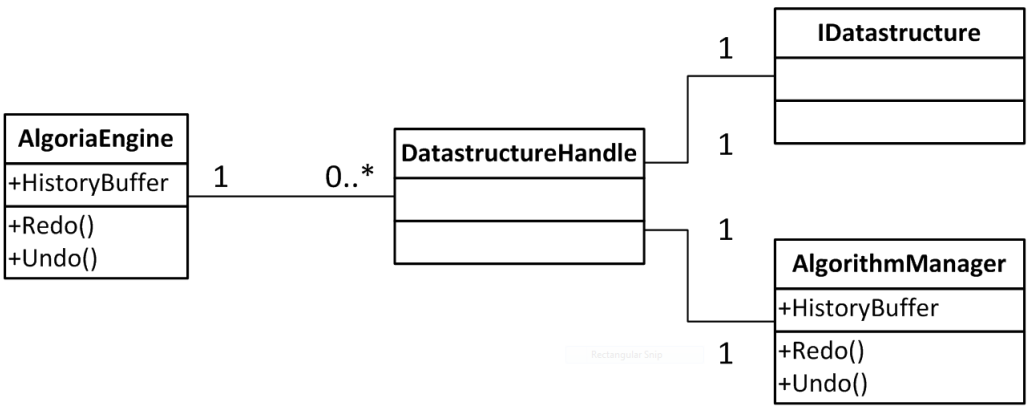


Abbildung 2.2.: Neue Position der Verlaufsliste.

Die Generierung einer Operation wird in der Datenstruktur selbst durchgeführt. Wie in der Abbildung 2.2 allerdings zu sehen ist, verfügt nur die Algoria-Engine über eine Referenz zur Datenstruktur. Damit trotzdem Operationen der *Algoria*-Engine gemeldet werden können, übergibt die *Algoria*-Engine der Datenstruktur bei der Instanziierung ein Delegate-Objekt. Mithilfe von diesem Delegate-Objekt kann eine Datenstruktur eine neue Operation in die Verlaufsliste eintragen, ohne eine Referenz zur *Algoria*-Engine zu haben. Somit wird eine zu enge Kopplung zwischen Datenstruktur und *Algoria*-Engine verhindert. Die *Algoria*-Engine ihrerseits ermöglicht den Zugriff auf die Verlaufsliste mithilfe einer ReadOnlyObservableCollection. Durch die Verwendung dieser Collection kann sichergestellt wer-

den, dass Zugriff nur lesender Natur ist und den Inhalt der Liste nicht verändert werden kann. Für Klassen, die sich dafür interessieren, ob eine neue Operation der Verlaufsliste hinzugefügt wurde, wird zusätzlich ein UndoRedoPushed-Event angeboten. Die beschriebenen Implementierungen sind in der Klasse AlgoriaEngine (siehe AlgoriaEngine.UndoRedo.cs) zu finden. Damit die Verlaufsliste visuell dargestellt werden kann, verfügt Algoria über einen UndoRedoStackVisualizer, der in Abbildung 2.3 zu sehen ist. Zurzeit dient der Visualizer vor allem einem Entwickler, der den Status der Verlaufsliste visualisiert haben möchte. In Zukunft ist es denkbar, diese Anzeige zu verwenden um mehrere Schritte rückgängig zu machen. Ein Beispiel dafür ist bereits in vielen Microsoft Produkten zu sehen. Es wird mit dem kleinen Dreieck neben dem Undo-Icon die Möglichkeit gegeben, die Verlaufsliste anzuzeigen und mehrere Undo-Schritte gleichzeitig auszuführen.

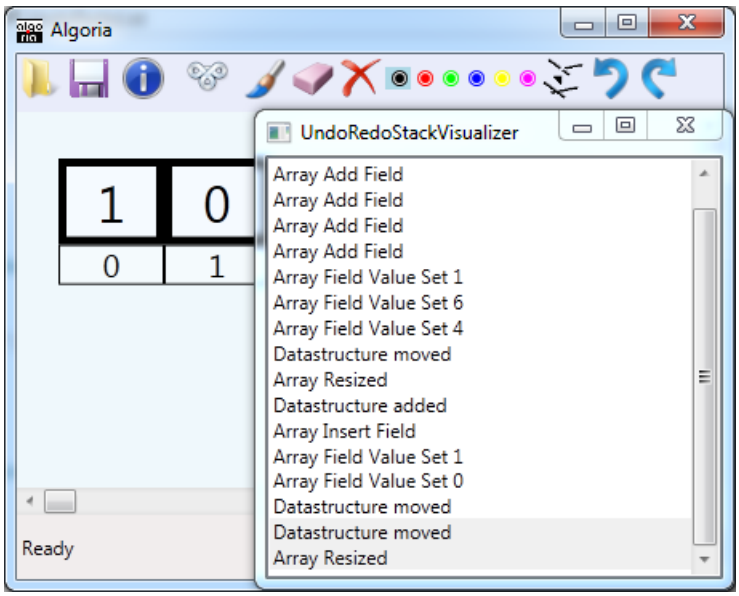


Abbildung 2.3.: Visualisierung der Verlaufsliste mit dem UndoRedoStackVisualizer

Operationen

Die Analyse der zu implementierenden Operationen hat bereits die Anforderungen an eine Operation aufgezeigt. Alle Operationen haben eine gemeinsame Basis, die abstrakte ParamDelegatedUndoRedo Klasse (siehe DelegatedDoUndo.cs). WCF erlaubt es nicht, Objekte mit Funktionalität zu übertragen. Aus diesem Grund muss die Instanz der Operationsklasse auf der Empfängerseite erneut erzeugt werden können und darf keinerlei Parameter, welche in der Undo oder Redo Methode verwendet werden, enthalten. Alle notwendigen Informationen müssen demnach der Undo / Redo Methode als Parameter direkt übergeben werden. Der Codeausschnitt 2.1 zeigt, stellvertretend für Undo und Redo, die Signatur der Undo Methode.

```
1 void Undo(IAlgoria engine, UndoRedoParams undoRedoParams)
```

Codeausschnitt 2.1: Undo Signatur

Diese Signatur wird wie folgt interpretiert. Die Undo Operation wird mit den Parametern aus dem UndoRedoParams Objekt ausgeführt. Das Ziel der Undo Methode ist es, das Ziel der Operation in der übergebenen IAlgoria Instanz zu finden und die entsprechende Änderung rückgängig zu machen. Jede implementierte Operation verfügt über eine dazugehörige Erweiterung der UndoRedoParams Klasse. Beispielsweise gibt es für die ParamArraySwapUndoRedo Klasse die ArraySwapUndoRedoParams Klasse. Wird eine Operationsklasse instanziiert, erhält diese alle notwendigen Informationen um ein Undo oder ein Redo auszuführen. Diese Informationen werden in der dazugehörigen UndoRedoParams

Erweiterung gespeichert. Die UndoRedoParams Klasse definiert die folgenden Felder:

- string UndoRedoType
- string UndoRedoName
- Guid DatastructureGuid

Wichtig für die Ausführung der Undo bzw. Redo Methode ist hier die Guid der Datenstruktur, welche das Ziel der Operation ist. Mithilfe dieser Guid kann die korrespondierende Datenstruktur in der *Algoria*-Engine gefunden werden. Das UndoRedoName Feld wird im UndoRedoStackVisualizer verwendet um die Operation auflisten zu können. UndoRedoType wird in der Implementierung der Kollaborationsinfrastruktur verwendet und im dazugehörigen Abschnitt erläutert.

Infrastruktur

Die Gruppierung der Operationen, welche über das Netzwerk verteilt werden, hat einen Einfluss auf das ServiceContract-Interface des Kollaborationsservers. Die Funktionalität der bisherigen SendAction-Methode wird durch zwei neue Methoden übernommen. Diese zwei Methodensignaturen sind im Codeausschnitt 2.2 zu sehen.

```
1 [OperationContract]
2 void SendAdminAction(ICollaborationAdminAction adminAction);
3
4 [OperationContract]
5 void SendModifyAction(UndoRedoParams undoRedoParams, Guid sender,
6     Guid exercise);
```

Codeausschnitt 2.2: ServiceContract Methoden für das Versenden von Operationen

Die erste Methode ist unverändert im Vergleich zur bisherigen Implementation. Der Umfang der Implementation des ICollaborationAdminAction-Interfaces hat sich jedoch verringert, da Veränderungen an der Zeichenfläche neu mithilfe von SendModifyAction propagiert werden können. Mit dieser neuen Methode können die, durch die Instanziierung des Operationsobjektes erzeugten, UndoRedoParams-Objekte direkt versendet werden. Da die UndoRedoParams Klasse primär für die lokale Verwendung in der Verlaufsliste entwickelt wurde, verfügt sie über gewisse Felder nicht. So müssen der Absender der Operation (clientGuid) und der Guid der betroffenen Aufgabe zusätzlich angegeben werden. Das Callback-Interface muss die dazugehörigen ReceiveAdminAction, bzw. ReceiveModifyAction anbieten, damit die Operationen vom Server zum Client weitergegeben werden können.

Wird eine neue Operation in die Verlaufsliste eingefügt, muss diese an den Server der Kollaborationssitzung weitergeleitet werden. Die SendModifyAction Methode verlangt für diesen Service ein Objekt vom Typ UndoRedoParams. Dieses Parameter-Objekt kann direkt vom Operationsobjekt abgefragt werden. Die Basisklasse aller Modify-Operationen (ParamDelegatedUndoRedo) definiert dafür ein abstraktes Feld mit dem Namen UndoRedoParams. Somit muss der Client keine Vergleiche anstellen um den Typ der Operation festzustellen und danach das dazugehörige Parameterobjekt zu instanziieren. Damit eine solche Folge von Vergleichen auch beim Empfangen einer Modify-Operation nicht notwendig wird, verfügt jede Erweiterung der UndoRedoParams Klasse über das Feld UndoRedoType. Dieses Feld enthält den Assembly-qualified-Name der zum Parameterobjekt gehörenden Operationsklasse. Mithilfe dieser Information kann beim Empfangen ein neues Operationsobjekt erzeugt werden. Der Codeausschnitt 2.3 zeigt diese Erzeugung.

```
1 var undoRedoType = Type.GetType(undoRedoParams.UndoRedoType);
2 var undoRedo = Activator.CreateInstance(undoRedoType, undoRedoParams) as
    ParamDelegatedUndoRedo;
```

Codeausschnitt 2.3: Erzeugen eines neuen Operationsobjektes auf Clientseite

Bei dem so erzeugten Operationsobjekt kann durch den Aufruf der Redo Methode die erwünschte Manipulation vollzogen werden.

2.2. Operation Serialization in Algoria Worksheet

Mit einer soliden Grundlage bezüglich der Voraussetzungen von Operation Serialization wird im Folgenden die Implementation des Synchronisierungsalgorithmus beschrieben. Diese Beschreibung ist in drei Teilstücke unterteilt. Im ersten Teil wird eine Implementierung von Operation Serialization konzeptionell betrachtet und geplant. Darauf folgen Beschreibungen zur tatsächlichen Implementierung.

2.2.1. Analyse / Konzeption der Implementierung

Die Implementation des Operation Serialization Algorithmus basiert auf Pseudocode, welcher in [Igna06] beschrieben wird. Zwei Punkte müssen hier explizit betrachtet werden:

- Wie wird der Konflikttyp zweier Operationen bestimmt?
- Wie werden überlappende Operationen gefunden?

Für die Bestimmung des Typs eines Konfliktes werden immer zwei Operationen miteinander verglichen. Basierend auf den Operationstypen kann entweder mit oder ohne Betrachtung der jeweiligen Parameter der resultierende Konflikttyp bestimmt werden. Im Anhang B ist ein Vergleich aller implementierten Operationen zu finden. Ist eine Zelle mit mehreren Farben ausgefüllt, bedeutet dies, dass der Konflikttyp von den jeweiligen Operationsparametern abhängig ist. Unter welchen Voraussetzungen der jeweilige Konflikt zu Stande kommt ist in den zwei Seiten nach der Konfliktmatrix als Pseudocode aufgeführt. Diese Vergleichsfunktionalität muss in Algoria Worksheet integriert werden.

Die Detektion einer Überschneidung von zwei Operationen kann auf mehrere Arten angegangen werden. Zwei Möglichkeiten werden im Folgenden betrachtet. Zum einen ist es denkbar, die Zeit als Messgrösse zu wählen. Ein anderer Ansatz ist die Verwendung einer Versionsnummer für den Nachweis einer Überschneidung.

Zeitstempel

In der Abbildung 2.4 werden drei mögliche Sequenzen aufgezeigt, in welchen das Auffinden einer Überschneidung mithilfe von Zeitstempeln gemacht werden soll. Es wird immer die Situation der Instanz bei S_1 betrachtet.

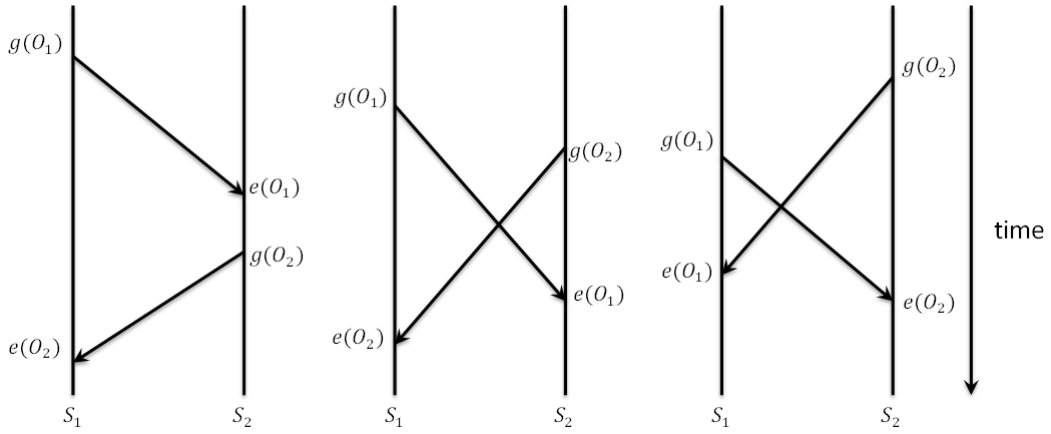


Abbildung 2.4.: Konfliktdetektion mit Zeitstempel

Die Sequenz auf der linken Seite zeigt den Fall, in welchem kein Potential für einen Konflikt entsteht, die Operationen schneiden sich nicht. Damit dies gilt, muss O_1 bei S_2 bereits ausgeführt worden sein, bevor O_2 generiert wird. Im mittleren Diagramm überschneiden sich die Operationen. O_2 wird bei S_2 generiert, bevor O_1 ausgeführt werden konnte. Daraus lässt sich ableiten, dass im folgenden Fall eine Überschneidung vorliegt:

$$t_{g(O_1)} < t_{e(O_2)} \quad \& \quad t_{g_{O_2}} < t_{e(O_1)}$$

Für die Bestimmung des Konfliktes braucht es demzufolge die Information über alle vier Zeitstempel. In der rechten Sequenz wird ein zusätzlicher Fall aufgezeigt. Hier schneiden sich zwar die Operationen, es werden jedoch nur drei Zeitstempel benötigt um diesen Konflikt erkennen zu können. Durch die Generierung von O_2 nach dem Entstehen von O_1 kann beim Eintreffen von O_2 bei S_1 zweifelsfrei eine Übereinstimmung detektiert werden. In der mittleren Sequenz ist eine Erkennung mit drei Zeitstempeln auch möglich, allerdings nur bei S_2. Verallgemeinert kann demzufolge gesagt werden, dass die Site, welche die Operation später generiert, den Konflikt mit drei Zeitstempeln erkennen kann. Zwei der Zeitstempel werden lokal generiert. Dabei handelt es sich um den Zeitpunkt der Generierung der lokalen Operation und um den Ausführungszeitpunkt der empfangenen Änderung. Der Zeitstempel des Generierungszeitpunktes der erhaltenen Operation kann im zu versendenden Objekt gespeichert werden und ist somit leicht zu bestimmen. Der Zeitpunkt der Ausführung einer lokal generierten Operation bei einem Kommunikationspartner ist schwieriger festzustellen. Es müsste jede erhaltene Operation mit einem Zeitstempel der Ausführung beantwortet werden. In Abbildung 2.4 ist zu sehen, dass im rechten Beispiel die Konfliktdetektion bei S_1 mit drei Zeitstempeln möglich ist. Gleichwohl benötigt S_2 die Information über den Ausführungszeitpunkt von O_2 bei S_1. Dies führt zur Schlussfolgerung, dass in jedem Fall die Bestimmung einer Überlappung mittels drei vorhandener Zeitstempel möglich ist. Dies gilt jedoch nur für eine der Seiten. Die Gegenüberliegende benötigt alle vier für das Feststellen der Überschneidung.

Versionsnummer

Die Verwendung eines globalen Versionszählers bildet eine Alternative zu Zeitstempeln. Die Abbildung 2.5 zeigt zwei mögliche Szenarien, bei welchen mithilfe der Versionsnummer der Konflikt detektiert werden soll. Zusätzlich zu denjenigen in 2.4 ist in diesen Beispielen eine weitere Zeitlinie enthalten. Zwischen den beiden *Algoira Worksheet* Instanzen S_1 und S_2 ist der Kollaborationsserver aufgeführt, welcher die Operationen einer Instanz an alle anderen verteilt. Die beiden Kästchen unter den Sequenzdiagrammen zeigen jeweils die basierende (orange) und die erzeugende (blau) Version der Operation. Die grundlegende Idee ist, dass S_1 und S_2 bei Version 0 starten. Ändert ein Benutzer etwas an der

Zeichenfläche, sendet *Algoira Worksheet* eine Operation, welche die Änderung repräsentiert. Dieser Operation wird noch die Versionsnummer hinzugefügt, die zum Zeitpunkt der Generierung lokal aktuell ist. Beim Erhalt einer Operation von einer *Algoira Worksheet* Instanz inkrementiert der Server seinen lokalen Versionszähler und sendet die generierte Nummer an den Sender der Operation zurück. Diese erhaltene Version ist somit lokal aktuell. Bevor der Server die Operation jedoch weiter verteilt, wird dem Operationsobjekt die Information hinzugefügt, welche Version diese verursacht hat.

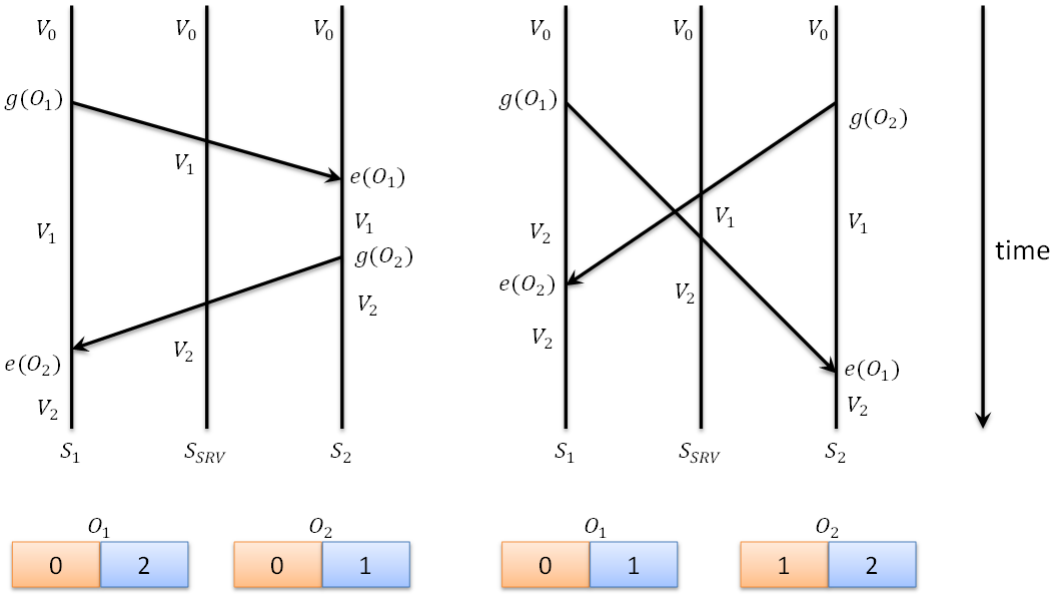


Abbildung 2.5.: Konfliktdetektion mit Versionsnummern

Im linken Fall ist erneut eine überschneidungsfreie Sequenz gezeigt. S_1 generiert O_1. Zu diesem Zeitpunkt ist bei S_1 die Version 0 aktuell. Mit dem Senden der Operation an den Server erhält S_1 die Information, dass O_1 die Version 1 ausgelöst hat. Somit hat S_1 die Version 1 und O_1 enthält zusätzlich die Information, dass die Operation Version 1 erzeugt hat. S_2 erhält O_1 zum Zeitpunkt der lokal aktuellen Version 0. Da O_1 auch auf Version 0 beruht, kann die Operation problemlos ausgeführt werden. Auch S_2 inkrementiert die lokale Versionsnummer auf 1. Da O_2 erst nach dem Ausführen von O_1 generiert wird, basiert O_2 auf Version 1. Beim Versenden an den Server wird die Versionsnummer nochmals inkrementiert. Da auf beiden Seiten jeweils die Ausgangsversion der Operation mit der lokal gültigen Versionsnummer übereinstimmt, ist keine Überschneidung vorhanden.
Das Beispiel auf der rechten Seite birgt jedoch eine Überschneidung. Beide Sites generieren zur annähernd selben Zeit die jeweiligen Operationen. Lokal aktuell ist sowohl bei S_1 als auch bei S_2 Version 0. Somit basieren O_1 und O_2 auf dieser Versionsnummer. Von S_1 wird O_1 zum Server versendet. Da jedoch O_2 bereits auf dem Server eingetroffen ist, wurde die globale Versionsnummer bereits auf 1 inkrementiert. Somit erzeugt O_1 Version 2. Bei S_1 wird demzufolge die lokale Versionsnummer von 0 direkt auf 2 erhöht. Beim Empfang von O_2 bei S_1 ist in der empfangenen Operation zu sehen, dass diese die Version 1 erzeugt hat. Somit muss die lokale Versionsnummer nicht angepasst werden, nachdem O_2 ausgeführt wurde. Da jedoch beide Versionen O_1 und O_2 auf derselben Version basieren, ist eine Überschneidung vorhanden. Im Allgemeinen überlappen sich zwei Operationen wenn:

$$|\{b(O_1), \ldots, c(O_1)\} \cap \{b(O_2), \ldots, c(O_2)\}| > 1$$

$b()$ bezeichnet hier die Basisversion, auf welcher die Operation generiert wurde. $c()$ gibt die Version an, welche durch die Operation erzeugt wurde. Die gegebene Definition schliesst den gezeigten Fall $(b(O_1) == b(O_2))$ mit ein.

Entscheidung

Der Vergleich der zwei Methoden zur Feststellung von überlappenden Operationen liefert den Schluss, dass sowohl die Verwendung von Zeitstempeln, als auch eine Implementation basierend auf Versionsnummern Nachteile haben. Für Zeitstempel bedeutet dies, dass immer einer der Kommunikationspartner den Ausführungszeitpunkt seiner Operation beim Gegenüber in Erfahrung bringen muss. Diese Anfrage benötigt zusätzliche Zeit, in welcher die empfangene Aktion nicht lokal ausgeführt werden kann. Damit die Operation in die Verlaufsliste eingeführt werden kann, muss die Information, ob es sich um eine überlappende Operation handelt, vorhanden sein. In der Zeit, in welcher auf den Zeitstempel der Ausführung gewartet werden muss, ist auch das Einfügen anderer Operationen blockiert. Dieses Problem tritt jedoch auch bei Versionsnummern auf. Die Antwort vom Server, welche Versionsnummer durch Operation erzeugt worden ist, benötigt für die Übertragung ebenfalls Zeit. Solange diese Nummer jedoch nicht bekannt ist, kann eine neu empfangene Operation nicht in die Verlaufsliste eingefügt werden. Wird in dieser Zeit eine Operation empfangen ist diese solange blockiert, bis die Information vom Server beim Client eintrifft. Zusätzlich führt die Protokollierung von Versionsnummern zu zusätzlicher Logik in der Server-Instanz. Da dieser Server möglichst leichtgewichtig gehalten werden soll, spricht dies gegen eine Implementation auf der Basis von Versionsnummern. Auch wenn der Entscheid für eine Verwendung von Zeitstempeln fällt, bringen diese eine zusätzliche Hürde. Nicht alle Kollaborationsteilnehmern haben dieselbe Zeit auf dem System eingestellt. Somit muss eine Synchronisierung dieser Zeit durchgeführt werden. Für dieses Problem sind in [Jae09] und [Kope87] Lösungen beschrieben.

2.2.2. Ausführungen zur Implementation

Collaboration Operation Serializer Architektur

Für die Bestimmung eines Konflikttyps ist bereits die Konfliktmatrix vorgestellt worden. Da es vorstellbar ist, dass dieses Konzept in einer späteren Version ausgetauscht wird, muss die Architektur entsprechend gewählt werden. Für die Realisierung von Operation Serialization in *Algoria Worksheet* wurde die folgende Klassenhierarchie aufgebaut.

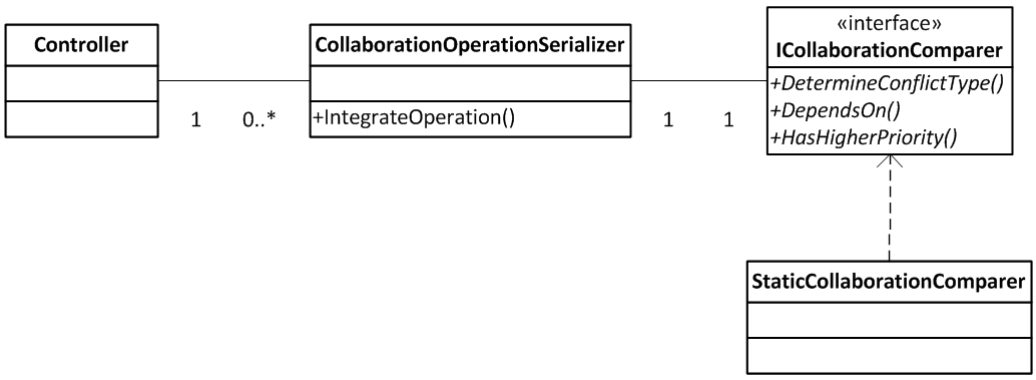


Abbildung 2.6.: Operation Serialization Klassenhierarchie

Wie in Abbildung 2.6 zu sehen, kann ein Controller (diese Klasse administriert die ganze Applikation) über mehrere Instanzen eines CollaborationOperationSerializers verfügen. Der Grund dafür ist, dass pro synchronisierte Zeichenfläche ein Real-Conflict- und Serialization-Graph geführt werden muss. Somit übernimmt eine CollaborationOperationSerializer-Instanz das Integrieren einer neuen Operation in die Verlaufsliste und führt gegebenenfalls auch Undo- bzw. Redo-Aktionen aus. Das Resultat der Integration ist die neuste Version der Verlaufsliste. Zusätzlich testet der CollaborationOperationSerializer auf das Vorhandensein einer Precedes-Beziehung und gibt an, ob zwei Operationen sich

überlappen. Mithilfe eines ICollaborationComparers werden Konflikttyp, Prioritäten und DependsOn Beziehungen erkannt. Die Abstraktion als Interface dient der einfacheren Austauschbarkeit des Comparers. Implementiert ist dieses Interface im StaticCollaborationComparer. Dieser bedient sich der Konfliktmatrix für die Konflikttypbestimmung.

Überlappende Operationen

Wie in der Analyse beschrieben, wird eine Überlappung mithilfe von Zeitstempeln festgestellt. Dies bedarf einiger Anpassungen sowohl am Contract-Interface des Kollaborationsservers als auch der versendeten Operationsklassen und dem Client einer Kollaboration. Die abstrakte Basisklasse UndoRedoParams wird mit drei Properties erweitert. Es handelt sich hierbei um zwei Zeitstempel (jeweils einen für den Generierungs- und für den Ausfühungszeitpunkt) und ein Guid, welcher die Operation zweifelsfrei identifiziert. Das Contract-Interface verfügt neu über eine Methode SendAcknowledgement(operationGuid, senderGuid, timeStamp). Mithilfe dieser Methode kann ein Client melden, dass er die Operation mit der Guid *operationGuid* zum Zeitpunkt *timeStramp* ausgeführt hat. Der Kollaborationsserver leitet diese Meldung dann zu dem Client weiter, welcher die entsprechende Operation generiert hat. Ein Client bekommt also für jede generierte Operation eine Empfangsmeldung. Für die Verwaltung der Empfangsmeldungen wird eine Tabelle geführt. Für jede versendete Operation wird vermerkt, von welchen Kollaborationspartnern bereits eine Empfangsbestätigung eingegangen ist. Bevor eine Operation in die Verlaufsliste integriert wird, versucht die Methode TryOperationIntegration (Codeausschnitt 2.4) sicherzustellen, dass eine Überlappung von Operationen erkannt werden kann.

```
1  FUNCTION TryOperationIntegration(Operation o)
2  IF o wurde lokal generiert
3      Einfuegen in die Verlaufsliste
4  ELSE IF Empfangsbestaetigungsliste enthaelt den Absender von o
5      IF vom Absender fehlt eine Empfangsbestaetigung
6          Warte eine Zeit T
7          Forciertes Einfuegen in die Verlaufsliste
8      ELSE
9          Einfuegen in die Verlaufsliste
10     END IF
11 ELSE
12     Einfuegen in die Verlaufsliste
13 END IF
14 END FUNCTION
```

Codeausschnitt 2.4: Pseudo Code TryOperationIntegration Methode

Sollte eine Empfangsbestätigung für den Sender der zu integrierenden Operation noch nicht erhalten worden sein, wird eine Zeit T gewartet. Dies ist notwendig, da die Übertragung der Empfangsbestätigung eine gewisse Zeit dauern kann. Ist auch nach dem zusätzlichen Abwarten die Bestätigung noch nicht eingetroffen, wird davon ausgegangen, dass keine Überschneidung besteht und die Operation wird in die Verlaufsliste eingefügt. Forciert bedeutet in diesem Zusammenhang, dass beim Versuch einen Konflikt zu detektieren von keiner Überschneidung ausgegangen wird.

3. Internet Service

Damit Benutzer nicht nur in einem gemeinsamen Netzwerk kollaborativ zusammenarbeiten können, wird im Folgenden die Möglichkeit einer standortunabhängigen Kollaborationssitzung besprochen.

3.1. Analyse / Konzeption der Implementierung

Eine Kommunikation unabhängig vom Standort der Kommunikationspartner kann durch ein Verlagern des Kollaborationsservices ins Internet erreicht werden. Dieser Service muss dieselben Funktionen umfassen wie derjenige, welcher in [Frey12b] beschrieben wurde. Bei der Entwicklung des lokalen Kollaborationsservers ist darauf geachtet worden, dass dieser nicht eng an *Algoria Worksheet* gekoppelt wird und eine spätere Auslagerung in eine externe Serverapplikation problemlos möglich ist. Es bietet sich an, diese Voraussetzung zu nutzen. Durch die Implementation der standortabhängigen Kollaboration mithilfe von WCF ist die Entwicklung des Internet Services unter Verwendung desselben Frameworks naheliegend.

Die Grundidee hinter dem Internet Server ist, dass eine Kollaborationsserver-Instanz beantragt werden kann. Die Benutzer verwenden dann die bestehende Implementation dieses Servers mit dem Unterschied, dass der Initiator einer Sitzung nicht mehr selbst Host für den Server ist. Damit dies erreicht werden kann, ist ein neuer Service erforderlich, welcher sich um die Bereitstellung und Administration von Kollaborationsservern kümmert. Im Gegensatz zum standortabhängigen Kollaborationsserver soll dieser Service nicht von einer C# Applikation selbst zur Verfügung gestellt, sondern mithilfe der Internet Information Services (IIS) angeboten werden. Dies hat den Vorteil, dass somit die Verwaltung der Ressourcen durch IIS geschieht. Das Verteilen eines Updates für den Internet-Service erleichtert sich durch den Einsatz von IIS zusätzlich. Wird die Version auf dem Server mit einer neuen überschrieben, ist bei der nächsten Verwendung des Services automatisch die aktuellste geladen.

Die Verschiebung des Kollaborationsservers auf einen Server, welcher Zugang zum Internet hat, bringt jedoch nicht nur Vorteile. Damit es einem Benutzer von *Algoria Worksheet* möglich ist, auf einer Maschine mehrere Instanzen des Kollaborationsservers parallel laufen lassen zu können, ist eine Möglichkeit zur Unterscheidung implementiert worden. Diese basiert darauf, dass jeder Server sein Kollaborationsservice über einen anderen Port anbietet. Wird ein neuer Kollaborationsserver lokal gestartet, sucht dieser zuerst nach einem freien Port im Intervall zwischen 13000 und 13100. Mithilfe dieser Information können zwei, auf derselben Maschine laufende, Kollaborationsserver unterschieden werden. Diese Art der Unterscheidung ist bei einem, im Internet laufenden, Server aus Sicherheitsgründen nicht möglich. Da auf dem Server jedoch mehrere Instanzen des Kollaborationsservers parallel laufen können, muss eine andere Methode zur Unterscheidung eingebaut werden. Die typische Adresse eines WCF Services lautet „Protokoll:/Servername:Port/Servicename“. Fest vorgegeben sind in dieser Adresse drei von vier Angaben nämlich Protokoll, Servername und Port. Somit kann nur der Servicename zur Unterscheidung zweier Services verwendet werden. Mithilfe einer Zufallszahl wird der Servicename erweitert, somit wird aus „Servicename“ neu „Servicename/Zufallszahl“. Diese Lösung hat eine zusätzliche Einschränkung, welche betrachtet werden muss. Wird ein Service auf einem Port veröffentlicht, ist dieser Port reserviert und kann von keinem weiteren Service mehr verwendet werden. Um das gemeinsame Verwenden eines Ports zu ermöglichen, gibt es in WCF den Port Sharing Service. Läuft dieser Windows-Service auf einer Maschine, ist es möglich, mehrere Services über einen Port laufen zu lassen. Der Port Sharing Service kümmert sich um die Weiterleitung des Netzwerkverkehrs zum dazugehörigen Kollaborationsserver.

3.2. Ausführungen zur Implementation

Die Infrastruktur für eine Kollaboration im lokalen Netzwerk ändert sich nicht. Für die Verwendung eines Kollaborationsservers im Internet sind einige Änderungen von Nöten. In Abbildung 3.1 ist die Infrastruktur einer Kollaborationssitzung im Internet zu sehen. Gelb eingefärbt ist die*Algoria Worksheet*-Instanz, welche die Sitzung initiiert. Grün hinterlegt sind die restlichen Kommunikationspartner und der blaue Kasten repräsentiert den Internet Service. Im Vergleich zur standortabhängigen Infrastruktur (siehe [Frey12b], Abbildung 3.1, Seite 20) verfügt der Initiator der Sitzung über keinen Algoria Collaboration Service Host, sondern lediglich über einen Server-Proxy für diesen Service Host. Damit eine neue Instanz eines Algoria Collaboration Service Hosts gestartet werden kann, verwendet der Initiator einen Algoria Collaboration Service Server-Proxy. Nachdem die Serviceadresse des Algoria Collaboraion Service-Hosts bekannt ist, wird der Algoria Collaboration Service Server clientseitig für die Kollaborationssitzung nicht mehr verwendet.

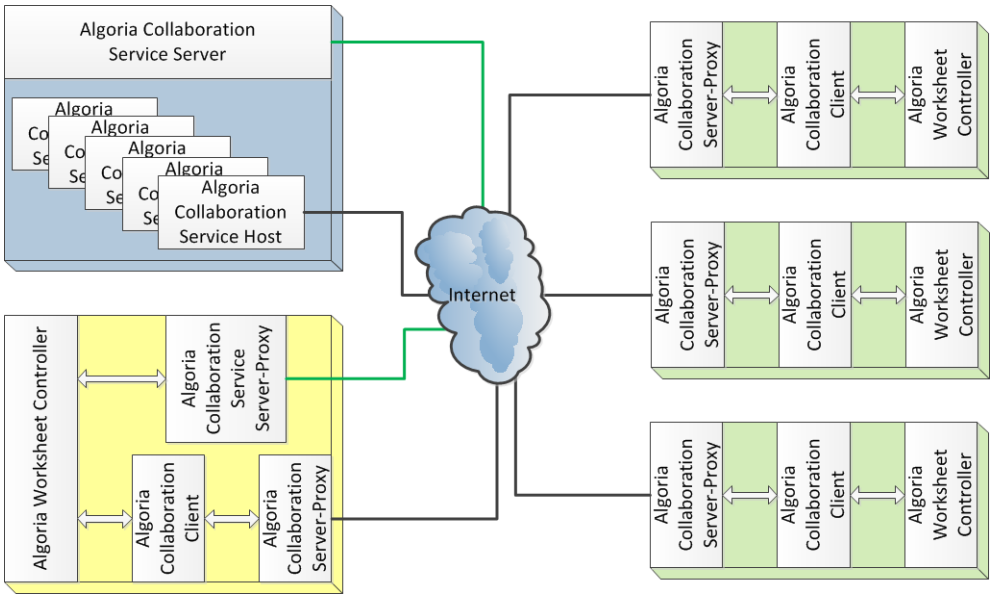


Abbildung 3.1.: Aufbau einer netzwerkunabhängigen Kollaboration

Da es sich beim Algoria Collaboration Service Server um einen WCF Service handelt, verfügt dieser über ein Service Contract-Interface. Diese ist in der Datei ICollaborationServiceContract.cs zu finden und definiert die Methoden, welche mithilfe eines Proxy-Objekts aufrufbar sein sollen. Das Interface ist schmal und verfügt nur über zwei Methoden. Zum einen ist es möglich, eine Liste aller laufenden Kollaborationssitzungen abzufragen, zum anderen kann eine neue Sitzung gestartet werden. Letztere liefert als Resultat die Service Adresse der erzeugten Sitzung. Mithilfe dieser Adresse können sowohl Initiator als auch alle übrigen Teilnehmer der Kollaborationssitzung eine Verbindung zum Kollaborationsserver aufnehmen. Die serverseitige Implementierung dieses Interfaces ist in der CollaborationServiceContract-Klasse (siehe CollaborationServiceContract.cs) zu finden. Auffällig ist, dass die Implementation des Service-Contracts über eine Annotation verfügt. Mithilfe der ServiceBehavior-Annotation kann unter anderem bestimmt werden, in welchem Kontextmodus der Service gestartet wird. Da der Kontext jeweils für jeden neuen Aufruf des Collaboration Service Servers beibehalten werden muss, wird der InstanceContextMode auf single gesetzt.

Für die Verwendung des Port Sharing Services muss nicht nur der entsprechende Windows-Service laufen. Beim Erstellen der Bindings für die Verbindung muss zusätzlich das PortSharingEnabled-Feld auf wahr gesetzt werden.

Soll der Algoria Collaboration Service Server aktualisiert werden, kann für das Verteilen der neuen Version die Anleitung in Anhang C verwendet werden.

4. Verbindungsunterbruch

Verliert ein Benutzer während einer Kollaborationssitzung die Verbindung soll es möglich sein, wieder in die Sitzung einzusteigen. Ob der Verbindungsverlust die Folge eines technischen Problems ist, oder weil der Benutzer den Standort wechseln muss und währenddessen keine Verbindung hat, ist hier nicht relevant. Im Folgenden wird beschrieben, wie das Wiedereintreten in eine Kollaborationssitzung ermöglicht wird.

4.1. Analyse / Konzeption der Implementierung

Für das Ermöglichen eines Wiedereinsteiges in eine Kollaborationssitzung, müssen einige Informationen vorhanden sein. Es handelt sich dabei um die Folgenden:

- Um welches Arbeitsblatt handelt es sich?
- Um welche Version des Arbeitsblattes handelt es sich?
- Unter welcher Guid nahm der Benutzer an der Kollaborationssitzung teil?
- Welche Änderungen hat der Benutzer während der Abwesenheit verpasst?

Damit sichergestellt werden kann, dass Benutzer nicht versehentlich in eine Sitzung wiedereintreten können, welcher sie zuvor gar nicht beigewohnt haben, müssen die Benutzer mit der vorherigen Client-Guid der Sitzung beitreten. Zusätzlich müssen die Arbeitsblätter und die darin enthaltenen Aufgaben übereinstimmen. Ist dies gegeben, können Benutzer wieder an der Sitzung teilnehmen. Hat sich während der Abwesenheit eines Teilnehmenden nichts am Arbeitsblatt geändert, müssen keine weiteren Vorkehrungen getroffen werden und ein Weiterarbeiten ist möglich. Sind jedoch Änderungen vollzogen worden, muss das Arbeitsblatt des Teilnehmenden auf den neusten Stand gebracht werden. Hier sind zwei Wege vorstellbar: die lokale Version wird mit der aktuellen des Initiators einer Kollaborationssitzung überschrieben oder die verpassten Änderungen müssen nachträglich ausgeführt werden. Am vielversprechendsten ist eine Kombination aus beiden Ansätzen. Hierzu wird beim Wiedereintritt in eine Kollaborationssitzung abgefragt, wie viele Änderungen verpasst wurden. Handelt es sich dabei um eine kleine Anzahl, werden die korrespondierenden Operationen übertragen und ausgeführt. Überschreitet die Zahl der verpassten Änderungen einen Schwellwert, wird das gesamte Arbeitsblatt vom Initiator in der aktuellen Version heruntergeladen.

4.2. Ausführungen zur Implementation

Für die Implementierung der Wiedereintrittsfunktionalität müssen sowohl client- als auch serverseitig Erweiterungen gemacht werden. Clientseitig handelt es sich primär um die Bewahrung der in der Analyse beschriebenen Informationen. Für den Fall, dass ein Benutzer nur kurz die Verbindung zum Server verliert, sind die relevanten Daten in der laufenden *Algoria Worksheet* Instanz vorhanden. Somit ist ein Speichern nicht notwendig. Wechselt aber ein Teilnehmender einer Kollaborationssitzung seinen Standort, ist es sehr wahrscheinlich, dass der Computer heruntergefahren wird und somit die notwendigen Informationen verloren gehen. Aus diesem Grund enthält die Klasse WorksheetDs (WorksheetDs.cs), welche ein Arbeitsblatt repräsentiert, zwei neue Felder. Das erste Feld enthält den Guid des Arbeitsblattes. Im zweiten wird der Guid, mit welchem der Teilnehmende zuletzt in einer Kollaborationssitzung involviert war, gespeichert. Solange ein Arbeitsblatt nur von Studierenden bearbeitet wird, ändert sich der Guid nicht. Wird jedoch durch einen Dozierenden eine Aufgabe im Arbeitsblatt gelöscht oder neu hinzugefügt, wird ein neuer Guid erzeugt. Dies hat den Grund, dass wenn Studierende kollaborativ arbeiten wollen und nicht dieselben Aufgaben in einen Arbeitsblatt

haben, eine Zusammenarbeit unmöglich ist. Das Verändern der Reihenfolg von Aufgaben in einem Arbeitsblatt hat jedoch keinen Einfluss auf die Guid. Die Identifikation einer Aufgabe als Ziel für eine Operation geschieht wiederum mit einem Guid, welcher pro Aufgabe verschieden ist. Somit können Aufgaben beliebig geordnet sein und ein kollaboratives Arbeiten ist trotzdem möglich. Tritt ein Benutzer einer Kollaborationssitzung bei, in welcher er zuvor nicht involviert war, wird der Guid des Clients neu generiert. Versucht ein Benutzer jedoch in eine bestehende Sitzung wiedereinzutreten, wird der zuvor gespeicherte Guid an den Kollaborationsserver versandt. Basierend auf diesem Guid kann der Server bestimmen, ob der Benutzer bereits einmal an der laufenden Kollaborationssitzung teilgenommen hat.

Für die Bestimmung, ob ein Benutzer schon einmal an einer spezifischen Kollaborationssitzung teilgenommen hat, speichert der Server nach dem Anmelden eines Clients den entsprechenden Guid. Durch die Erweiterung des CollaborationContract-Interfaces (ICollaborationContract.cs) ist es nun für einen Client möglich, abzufragen, ob der Server bereits eine Verbindung zu einem Client mit der gegebenen Guid hatte. Hierbei handelt es sich um die HasHistoryForClient(Guid clientGuid)-Methode. Wenn sich ein Client beim Server mit der Methode Subscribe(..., Guid clientGuid) anmeldet, wird in der neuen Version ein SubscribeAnswer-Objekt (SubscribeAnswer.cs) zurückgegeben. Anhand von dieser Antwort kann der Client sehen, ob er bereits einmal bei diesem Server angemeldet war. Falls ja, enthält dieses Objekt eine Liste von allfällig verpassten Operationen oder gibt an, dass zu viele Änderungen verpasst wurden. Ist dies der Fall oder hat ein Client noch nie in der betreffenden Kollaborationssitzung teilgenommen, wird die aktuelle Version beim Initiator der Sitzung abgeholt. Damit der Server ein solches SubscribeAnswer-Objekt erstellen kann, wird eine Liste aller verteilen Operationen geführt. Meldet sich ein Client neu beim Server an, wird fortan protokolliert, welcher Client welche Operation erhalten hat. Meldet sich eine Client-Instanz vom Server ab oder verliert die Verbindung auf andere Weise, wird die Position der letzten erhaltenen Operation gespeichert. Verbindet sich der gleiche Client erneut, können alle Operationen hinter der gespeicherten Position dem SubscribeAnswer-Objekt angefügt werden.

Öffnet der Benutzer ein Arbeitsblatt in *Algoria Worksheet*, wird im Hintergrund automatisch nach offenen Kollaborationssitzungen gesucht. Werden dabei Resultate erzielt, wird verglichen, ob die Sitzung dasselbe Arbeitsblatt bearbeitet. Ist auch dies der Fall, wird mithilfe der HasHistoryForClient-Methode abgeklärt, ob der Benutzer in dieser Sitzung tätig war. Trifft auch dies zu, wird eine Meldung angezeigt, ob der Benutzer dieser Kollaborationssitzung wieder teilnehmen möchte. Auf diese Weise soll verhindert werden, dass lokal Änderungen gemacht, diese aber nie an die Kollaborationspartner versendet werden.

5. Fazit

Nachdem die Arbeiten in diesem Projekt beschrieben wurden, werden hier die Fragen aus Kapitel 1.2 noch einmal aufgegriffen und Antworten dazu formuliert.

> Wie realisieren Sie eine kollaborative Lernumgebung mit Synchronisationprimitiven aus der Basis von *Algoria Worksheet*?

Durch die Vorarbeit aus den Vorgängerprojekten sind sowohl Grundlagen in der Implementierung als auch Analysen von geeigneten Synchronisationprimitiven vorhanden. Mithilfe letzterer ist die Entscheidung auf den Einsatz von Operation Serialization gefallen. Damit diese in *Algoria Worksheet* eingesetzt werden kann, wurden die Voraussetzungen dafür geschaffen. Dazu zählen sowohl das Implementieren einer Verlaufsliste direkt in *Algoria* als auch das Erweitern der Funktionalität des Kollaborationsservers. Mithilfe dieses Servers können mehrere Studierende kollaborativ an demselben Arbeitsblatt arbeiten. Das System beruht auf dem Austausch von gemachten Änderungen an der jeweils lokalen Version des Arbeitsblattes. Bearbeiten Studierende jedoch die gleiche Aufgabe innerhalb eines Arbeitsblattes, besteht die Möglichkeit eines Konfliktes. Solche Konflikte können je nach Reihenfolge der Änderungsausführung zu unterschiedlichen Resultaten und somit zum Verlust der Konsistenz führen. Eine Konfliktmatrix ermöglicht den Vergleich zweier Operationen und liefert, sofern einer besteht, den Typ des aus der Kombination entstehenden Konfliktes. Unter Verwendung dieser Konfliktmatrix konnte eine Integrations-Methode implementiert werden. Diese Methode integriert eine empfangene Operation so in die Verlaufsliste, dass keine Einbussen in der Konsistenz entstehen.

> Wie erreichen Sie eine standortunabhängige Kollaboration, sodass von überall her an der Lösung einer Aufgabe mitgearbeitet werden kann?

Durch die Integration des Kollaborationsservers direkt in *Algoria Worksheet* befindet sich dieser automatisch in demselben Netz wie die Maschine auf der die Kollaborationssitzung gestartet wurde. Somit müssen alle Kollaborationspartner in demselben Netzwerk sein und eine Kollaborationssitzung ist lokal an einen Ort gebunden. Abhilfe bezüglich diesem Problem schafft eine Kollaborationsserverinstanz, welche auf einer Maschine läuft, auf die via Internet zugegriffen werden kann. Damit eine standortunabhängige Kollaborationssitzung möglich ist, ist eine Service Applikation entwickelt worden, welche auf einem IIS läuft. Bei diesem Service können Kollaborationssitzung und die dazugehörigen Server-Instanzen gestartet werden.

> Wie kann erreicht werden, dass kurze Verbindungsunterbrüche zu keinen grösseren Einbussen in der Handhabung führen?

Verbindungsunterbrüche zwischen dem Server einer Kollaborationssitzung und einem der Teilnehmenden ist dann ein Problem, wenn dadurch Änderungen an einer Aufgabe verpasst werden. Aus diesem Grund ist die Möglichkeit implementiert worden, dass ein Server die verpassten Änderungen an einen Client liefert, sobald dieser sich wieder in die Kollaborationssitzung einklinkt. Das Prinzip hinter dieser Lösung ist, dass der Server eine Liste aller Operationen führt und zusätzlich die Information speichert, welcher Client welche Operation erhalten hat. Hat ein Client zu viele Operationen verpasst, wird das ganze Arbeitsblatt neu vom Initiator der Kollaborationssitzung verlangt und weitergeleitet.

> Wie überprüfen Sie die Praxistauglichkeit Ihres Systems auf sinnvolle Art und Weise?

Die Praxistauglichkeit der kollaborativen Lernumgebung kann direkt im Klassenverband getestet werden. So wäre es denkbar, dass ein Teil der Studierenden eine Aufgabenserie in Einzelarbeit löst. Ein zweiter Teil bearbeitet dieselbe Reihe von Aufgaben kollaborativ in einem Gruppenraum. Eine dritte Gruppe geht die Aufgabenserie jeweils von zuhause aus in einer dezentralen Kollaborationssitzung an. Leider konnte dieser Test während der Projektarbeit nicht durchgeführt werden.

6. Reflexion

6.1. Projektmanagement

Wie bereits in den vergangenen Projektarbeiten hat die Kommunikation zwischen Studierendem und Advisor reibungslos funktioniert. Durch die beibehaltenen wöchentlichen Sitzungen ist ein fortlaufender Dialog entstanden, welcher die Projektarbeit positiv beeinflusst hat. Für die Erarbeitung der Grundlagen für Operation Serialization, namentlich die Einbindung der Verlaufsliste in *Algoria*, wurde von Beginn an zu wenig Zeit eingeplant. Aus diesem Grund ist das eigentliche Ziel, die Implementierung von Operation Serialization, zunehmend nach hinten verschoben worden. Ansonsten wurde vor der wöchentlichen Besprechung der Zeitplan immer wieder betrachtet und analysiert, ob sich das Projekt noch auf einem guten Weg befindet.

6.2. Anforderungen

Die ersten drei der folgenden vier Absätze haben das Ziel, die Ergebnisse der Implementation der einzelnen Anforderungen zu analysieren und zu reflektieren. Im letzten Absatz werden Arbeiten besprochen, welche nicht direkt im Zusammenhang mit den Anforderungen aus der Tabelle 1.1 auf Seite 1.1 erledigt wurden.

Kollaborative Bearbeitung

Die Implementation der kollaborativen Bearbeitung erlaubt es, mehreren Benutzern gleichzeitig eine Zeichenfläche in einer Aufgabe für *Algoria Worksheet* zu editieren. Durch das Fehlen von UndoRedo-Operationen können in der vorliegenden Implementierung jedoch noch nicht alle lokalen Operationen übertragen werden. Einzige der Operationskatalog für Arrays ist schon sehr weit entwickelt. Wie bereits beschrieben, stellte die Verlaufsliste in Algoria eine grössere Herausforderung dar als gedacht. Durch eine teilweise vorhandene Verlaufsliste, welche aber nur pro Datenstruktur funktioniert, konnte ich einen Eindruck gewinnen, wie die globale Liste implementiert werden kann. Mithilfe einer statischen Konfliktmatrix wird die Ausführung einer Integration-Methode nach dem Vorbild von Operation Serialization ermöglicht. Dank dieser Methode wird das Auftreten von Inkonsistenzen zwischen den jeweils lokalen Versionen verhindert. Schwieriger als zu Beginn angenommen gestaltete sich das Detektieren von überlappenden Operationen. Hier besteht die Möglichkeit einer effizienteren Implementierung. Auch wurde die Zeitsynchronisierung während diesem Semester nur theoretisch betrachtet und fand in der Implementationsphase keinen Platz mehr.

Auch wenn ich in den vergangenen drei Projektarbeiten immer mit *Algoria* zu tun hatte, gewann ich nie einen so tiefen Einblick in das Projekt wie während dieser Arbeit. Durch das Implementieren einer neuen Funktionalität wurde jedoch mein Verständnis für *Algoria* gestärkt, was sich auch in der Arbeit an *Algoria Worksheet* positiv bemerkbar gemacht hat. Besonders erwähnenswert ist hier noch die Möglichkeit der Erweiterung dieser Operation Serialization Funktionalität. Die Erstellung und Erweiterung der Konfliktmatrix ist eine zeitintensive Arbeit. Während dieser Projektarbeit hat sich ein weiterer Ansatz zur Konfliktbestimmung herauskristallisiert. Es handelt sich um eine Bestimmung mithilfe von Annotationen. Diese Annotation beschreibt beispielsweise ob eine Operation auf einen oder mehrere Indizes zugreift oder zu einer Manipulation der Indizes hinter dem eigentlichen Ziel der Operation. Letzteres tritt zum Beispiel auf, wenn ein Feld in einem Array eingefügt oder gelöscht wird. Wenn mithilfe dieser Annotationen die Konflikttypen bestimmt werden können, ist das Erweitern der Konfliktmatrix überflüssig. Leider konnte diese Idee in dieser Arbeit nicht weiter aufgegriffen werden.

Internet Service

Mit der Implementation, welche zum jetzigen Zeitpunkt vorliegt, ist es möglich, auf einem Server, welcher vom Internet her erreichbar ist, eine Kollaborationssitzung zu starten. Leider kann die gestartete Sitzung nur aus dem FHNW Netzwerk aufgerufen werden. Grund dafür ist, dass ein zusätzlicher Port gegen aussen auf der Firewall hätte geöffnet werden müssen und dies möglicherweise eine zu lange Zeit gedauert hätte. Die Kommunikation konnte dennoch netzwerkübergreifend getestet werden. So können *Algoria Worksheet* Instanzen aus dem EDU und dem ADM Netz der FHNW kollaborativ zusammenarbeiten. Ohne die Verwendung des Internet Servers wäre dies nicht möglich. Durch die Realisierung des lokalen Kollaborationsservers als selfhosted Service wurde ich zum ersten Mal mit der Entwicklung einer Applikation für einen IIS konfrontiert. Für den Internet Server wären noch einige Erweiterungen denkbar. Eine Managementkonsole, möglicherweise als Webseite, würde einen Überblick über die laufenden Kollaborationssitzungen ermöglichen. Zudem ist die Verantwortlichkeit für die Kollaborationsserver-Instanzen nicht genau geklärt. Denkbar wäre, dass der Initiator der Kollaborationssitzung zusätzlich die Möglichkeit hätte, Einstellungen auf der verwendeten Server Instanz zu manipulieren.

Verbindungsunterbrüche

Die jetzige Implementation erlaubt es einem Client, nach dem Verlust der Verbindung, sich wieder in die Kollaborationssitzung einzuklinken. Um die verpassten Operationen zu erhalten, führt der Server eine Liste, in welcher nachgeführt wird, welcher Client welche Operationen bereits erhalten hat. Übersteigt die Anzahl der verpassten Operationen einen Schwellwert, wird das ganze Arbeitsblatt in der neusten Version geladen. Es ist vorstellbar, dass die implementierte Strategie, wann die gemachten Änderungen nachgesendet werden und wann das gesamte Arbeitsblatt benötigt wird, durch eine intelligentere zu ersetzen. Diese hat möglicherweise keinen fixen Schwellwert. Die Entscheidung, nur die verpassten Operationen oder das ganze Arbeitsblatt zu versenden, könnte unter Berücksichtigung der jeweils benötigten Zeit getroffen werden. Zusätzlich muss eine Lösung gefunden werden, wie mit Änderungen umgegangen wird, welche lokal während dem Verbindungsunterbruch vollzogen werden. Hier ist es denkbar die Verlaufsliste zu speichern und beim Wiedereintritt an den Server zu senden.

Sonstige Ergebnisse

Im Gegensatz zur vorhergehenden Arbeit sind die sonstigen Ergebnisse dieser Projektarbeit nicht implementationstechnischer Natur. Im Verlauf dieser Arbeit ist ein Paperdraft einer wissenschaftlichen Publikation mit dem Titel „Operation Serialization in Algoria“ entstanden. Diese Arbeit beschreibt, wie Operation Serialization in Algoria Worksheet angewendet wird, um eine kollaborative Bearbeitung zu ermöglichen. Dadurch, dass ich bis anhin nur auf der konsumierenden Seite war, habe ich sehr viel lernen können.

7. Ehrlichkeitserklärung

Hiermit bestätigt der Autor, diese Arbeit ohne fremde Hilfe und unter Einhaltung der gebotenen Regeln erstellt zu haben.

Reto Frey

Ort, Datum Unterschrift

Literaturverzeichnis

[Elli89] Ellis, C.A. und Gibbs, S.J. *Concurrency Control in Groupware Systems*. In Proceedings of *ACM SIGMOD Conference on Management of Data* (SIGMOD '89), S. 399-407, 1989.

[Frey11] Frey, R. und Zogg K. *IP6 Algoria Worksheet*, Projektbericht zur Bachelor Thesis, 2011.

[Frey12a] Frey R. *P7: Algoria - Persönliche Lernumgebung*, Projektbericht zum Informatik Projekt 7, 2012.

[Frey12b] Frey R. *P8: Algoria - Kollaboration im Schulzimmer*, Projektbericht zum Informatik Projekt 8, 2012.

[Igna06] Ignat, C.-L. and Norrie, M.C. *Drwa-Together: Graphical Editor for Collaborative Drawing*. In Proceedings of *International Conference on Computer Supported Cooperative Work* (CSCW'06), S. 269-278, 2006.

[Jens12] Jensen, P. [online] *http://www.me.xtexas.edu~jensen/models/network/net9.html* Zugegriffen am 10.09.2012.

[Jae09] Jae, H. C. Hoon, K. Sihai, W. Jaehoon, L. Hanlim, L. Seongtaek, H. Sungdae, C. YunJe, O. Tae-Jin, L. textitA Novel Method for Providing Precise Time Synchronization in a Distributed Control System Using Boundary Clock. *Instrumentation and Measurement, IEEE Transactions* , vol.58, no.8, S. 2824-2829, 2009

[Kope87] Kopetz, H. Ochsenreiter W. *Clock Synchronization in Distributed Real-Time Systems. Computers, IEEE Transactions*, vol.C-36, no.8, S. 933-940, Aug. 1987

[Micr12] Microsoft. MSDN Artikel zur Windows Communication Foundation [online] *http://msdn.microsoft.com/en-us/library/ms731082.aspx* Zugegriffen am 04.02.2012.

A. Aufgabenstellung P9

Fachhochschule Nordwestschweiz
Hochschule für Technik

MSE | MASTER OF SCIENCE IN ENGINEERING

Vertiefungsausbildung

Modulbeschreibung

P9 im Herbstsemester 2012

Institut: **IMVS**

Projekttitel: **Algoria: Verteiltes Klassenzimmer**

Thema: (Kurzbeschreibung)

Interaktive Lernumgebungen erlauben die gezielte und selbständige Auseinandersetzung mit einem vorgegebenen Lernstoff. Sie leiten den Lernenden durch den Lernprozess und helfen ihm, sein Wissen zu überprüfen und zu festigen.

Auf Basis der Magic-Ink-Software Algoria soll eine interaktive Lernumgebung für den Algorithmen und Datenstrukturen Unterricht in der Informatik entwickelt und in der Praxis erprobt werden. Die Interaktion soll sich an den gleichen Paradigmen wie bei Algoria orientieren (möglichst natürliche Interaktion ausgerichtet auf [Multi-]Touchscreens).

Projektteam:

Auftraggeber: (Adresse)	IMVS, Christoph Stamm
Student/-in	Reto Frey
Betreuer (extern):	
Advisor (intern):	Christoph Stamm

Fragestellungen: (erwartete Arbeiten)

Realisieren Sie eine kollaborative Lernumgebung mit Synchronisationsprimitiven auf der Basis von Algoria Worksheet in Absprache mit Ihrem Betreuer. Diese Lernumgebung soll es ermöglichen, dass Studierende von überall her an der Lösung von Algoria Arbeitsblättern teilhaben können. Wie kann erreicht werden, dass kurze Verbindungsunterbrüche zu keinen grösseren Einbussen in der Handhabung führen? Überprüfen Sie die Praxistauglichkeit Ihres Systems auf sinnvolle Art und Weise. Erstellung Sie ein zum Thema passendes wissenschaftliches Paper.

Ergänzenden Veranstaltungen: Informatik-Seminar

Starttermin: 28.09.2012

Abgabetermin: 08.02.2013

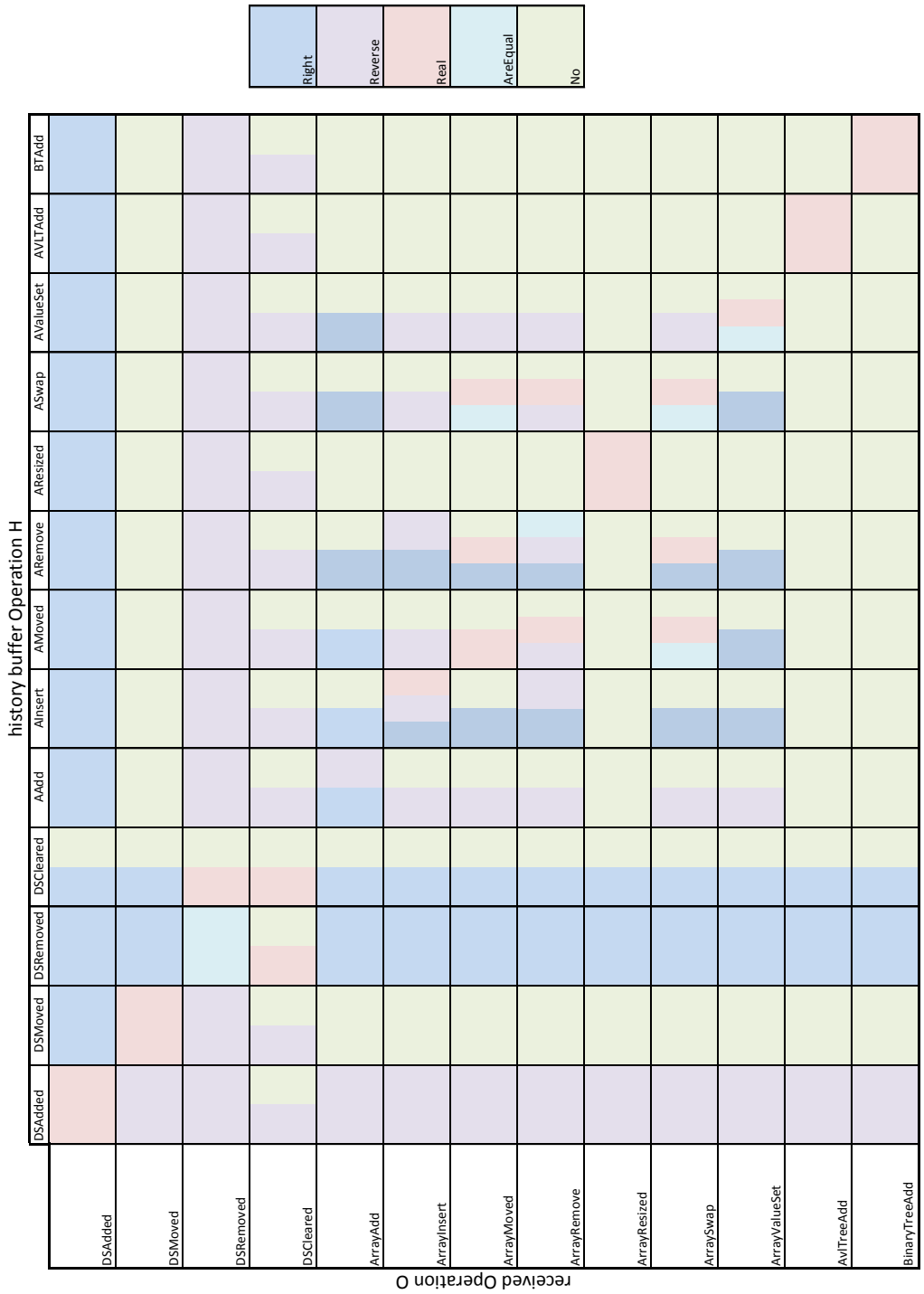
Ort, Datum: _____ _____

Unterschriften: _____ _____

Student/Studentin Advisor

B. Konfliktmatrix

Konfliktmatrix



1 / 3

Datenstruktur Typ Unabhängig

Tabelle 0.1: Datenstrukturtyp unabhängige Operationen

O	H	**Resultat**
DSAdded	DSCleared	$\exists x \in H.Guids \vert x = O.Guid\ ?\ Right : No$
DSMoved	DSCleared	$\exists x \in H.Guids \vert x = O.Guid\ ?\ Right : No$
DSRemoved	DSCleared	$\exists x \in H.Guids \vert x = O.Guid\ ?\ Real : No$
DSCleared	DSAdded	$\exists x \in O.Guids \vert x = H.Guid\ ?\ Reverse : No$
DSCleared	DSMoved	$\exists x \in O.Guids \vert x = H.Guid\ ?\ Reverse : No$
DSCleared	DSRemoved	$\exists x \in O.Guids \vert x = H.Guid\ ?\ Real : No$
DSCleared	DSCleared	$\exists x \in O.Guids, y \in H.Guids \vert x = y\ ?\ Real : No$
DSCleared	Any	$\exists x \in O.Guids \vert x = H.Guid\ ?\ Reverse : No$
Any	DSCleared	$\exists x \in H.Guids \vert x = O.Guid\ ?\ Right : No$

Array

Tabelle 0.2: Array Operationen

O	H	**Resultat**
ArrayAdd	ArrayAdd	$O.PreviousSize < H.PreviousSize\ ?\ Right : Reverse$
ArrayAdd	ArrayInsert	$O.PreviousSize <= H.Index\ ?\ Right : No$
ArrayAdd	ArrayMoved	$O.PreviousSize <= H.From \parallel O.PreviousSize <= H.To\ ?$ $Right : No$
ArrayAdd	ArrayRemove	$O.PreviousSize <= H.Index\ ?\ Right : No$
ArrayAdd	ArraySwap	$O.PreviousSize <= H.IndexA \parallel$ $O.PreviousSize <= H.IndexB\ ?$ $Right : No$
ArrayAdd	ArrayValueSet	$O.PreviousSize <= H.Index\ ?\ Right : No$
ArrayInsert	ArrayAdd	$O.Index >= H.PreviousSize\ ?\ Reverse : No$
ArrayInsert	ArrayInsert	$O.Index == H.Index\ ?$ $Real : O.Index > H.Index\ ?$ $Right : Reverse$
ArrayInsert	ArrayMoved	$O.Index <= H.From \parallel O.Index <=\ ?\ Reverse : No$
ArrayInsert	ArrayRemove	$O.Index <= H.Index\ ?\ Reverse : Right$
ArrayInsert	ArraySwap	$O.Index <= H.IndexA \parallel O.Index <= H.IndexB\ ?$ $Reverse : No$
ArrayInsert	ArrayValueSet	$O.Index <= H.Index\ ?\ Reverse : No$
ArrayMoved	ArrayAdd	$O.From >= H.PreviousSize \parallel O.To >= H.PreviousSize\ ?$ $Reverse : No$
ArrayMoved	ArrayInsert	$O.From <= H.Index \parallel O.To <= H.Index\ ?\ Right : No$
ArrayMoved	ArrayMoved	$\vert\{O.From...O.To\} \cap \{H.From...H.To\}\vert > 0\ ?\ Real : No$
ArrayMoved	ArrayRemove	$O.From > H.Index\ \&\ O.To > H.Index\ ?$ $Right : \vert\{O.From...O.To\} \cap \{H.Index\}\vert > 0\ ?$ $Real : No$
ArrayMoved	ArraySwap	$\vert\{O.IndexA\} \cap \{H.From...H.To\}\vert > 0 \parallel$ $\vert\{O.IndexB\} \cap \{H.From...H.To\}\vert > 0\ ?$ $Real : No$
ArrayMoved	ArrayValueSet	$\vert\{O.From...O.To\} \cap \{H.Index\}\vert > 0\ ?\ Reverse : No$
ArrayRemove	ArrayAdd	$O.Index >= H.PreviousSize\ ?\ Reverse : No$

Fortsetzung auf der folgenden Seite

Tabelle 0.2 – *Fortsetzung der letzten Seite*

O	H	**Resultat**
ArrayRemove	ArrayInsert	$O.Index >= H.Index\ ?\ Right : Reverse$
ArrayRemove	ArrayMoved	$O.Index < H.From\ \&\ O.Index > H.To\ ?$ $Reverse : \lvert\{O.Index\} \cap \{H.From...H.To\}\rvert > 0\ ?$ $Real : No$
ArrayRemove	ArrayRemove	$O.Index == H.Index\ ?$ $Equal : O.Index < I.Index\ ?$ $Reverse : Right$
ArrayRemove	ArraySwap	$O.Index == H.IndexA \parallel O.Index == H.IndexB\ ?$ $Real : O.Index <= H.IndexA \parallel O.Index <= H.IndexB\ ?$ $Reverse : No$
ArrayRemove	ArrayValueSet	$O.Index <= H.Index\ ?\ Reverse : No$
ArraySwap	ArrayAdd	$O.IndexA >= H.PreviousSize \parallel$ $O.IndexB >= H.PreviousSize\ ?$ $Reverse : No$
ArraySwap	ArrayInsert	$O.IndexA >= H.Index \parallel O.IndexB >= H.Index\ ?$ $Right : No$
ArraySwap	ArrayMoved	$\lvert\{O.IndexA\} \cap \{H.From...H.To\}\rvert > 0 \parallel$ $\lvert\{O.IndexB\} \cap \{H.From...H.To\}\rvert > 0\ ?$ $Real : No$
ArraySwap	ArrayRemove	$O.IndexA == H.Index \parallel O.IndexB == H.Index\ ?$ $Real : O.IndexA > H.Index \parallel O.IndexB > H.Index\ ?$ $Right : No$
ArraySwap	ArraySwap	$\lvert\{O.IndexA, O.IndexB\} \cap \{H.IndexA, H.IndexB\}\rvert = 2\ ?$ $Egual :$ $\lvert\{O.IndexA, O.IndexB\} \cup \{H.IndexA, H.IndexB\}\rvert < 4\ ?$ $Real : No$
ArraySwap	ArrayValueSet	$O.IndexA == H.Index \parallel O.IndexB == H.Index$ $?\ Reverse : No$
ArrayValueSet	ArrayAdd	$O.Index >= H.PreviousSize\ ?\ Reverse : No$
ArrayValueSet	ArrayInsert	$O.Index >= H.Index\ ?\ Right : No$
ArrayValueSet	ArrayMoved	$\lvert\{O.IndexA\} \cap \{H.From...H.To\}\rvert > 0\ ?\ Right : No$
ArrayValueSet	ArrayRemove	$O.Index >= H.Index\ ?\ Right : No$
ArrayValueSet	ArraySwap	$O.Index == H.IndexA \parallel O.Index == H.IndexB\ ?$ $Right : No$
ArrayValueSet	ArrayValueSet	$O.Index == H.Index\ ?\ O.Value == H.Value\ ?$ $Equal : Real : No$

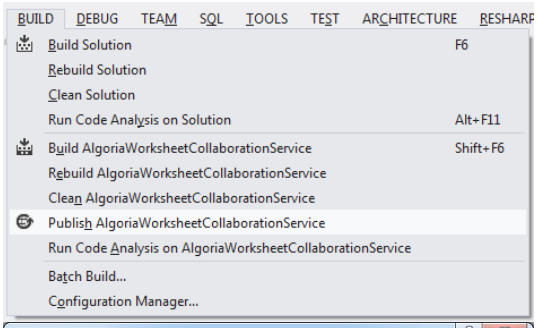
C. How To: Collaboration Service Update

Update des Algoria Worksheet Collaboration Service

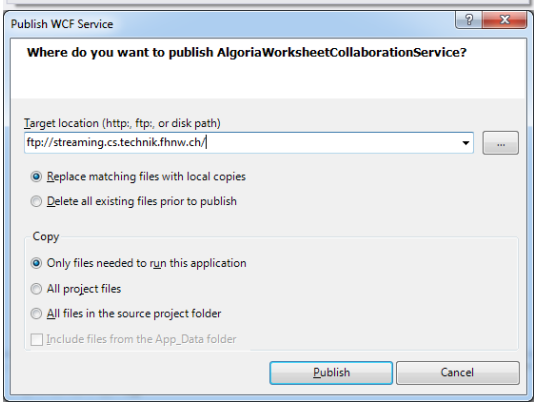
Die folgenden Schritte beschreiben das Vorgehen, wenn der Algoria Worksheet Internet Service mit einer neuen Funktionalität erweitert werden soll. Für das Durchführen eines solchen Updates muss sich der Computer im Netzwerk der FHNW befinden. ADM oder dem EDU-Netz spielt hierbei keine Rolle.

- Die Änderungen am Service durchführen (Contract-Interface und Implementation).
- Einen Build des Projektes durchführen.

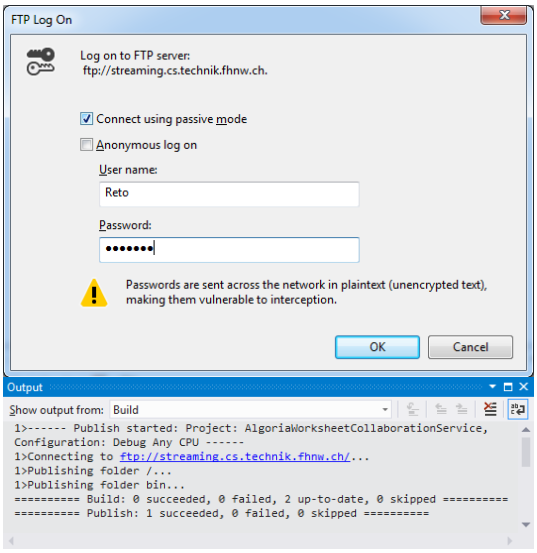
- Im Solution Explorer das Projekt AlgoriaWorksheetCollaborationService auswählen
- Im Menü Build den Punkt Publish AlgoriaWorksheetCollaborationService auswählen.



- Als Ziel des Publish-Vorganges die Adresse „ftp://streaming.cs.technik.fhnw.ch“ auswählen.
- „Publish“ klicken.



- Benutzername: Reto
  Passworkt: Alg4Str
- „Ok“ klicken.



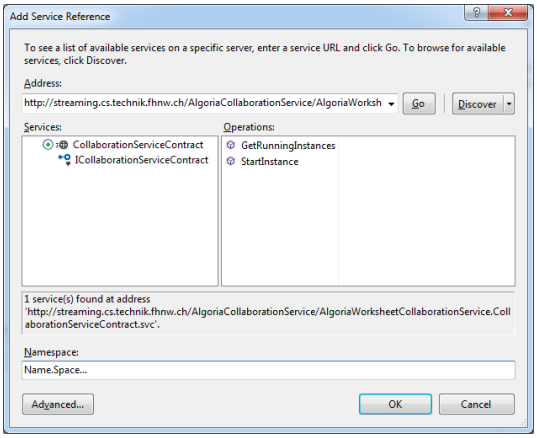
- Die Output-Anzeige von Visual Studio nach erfolgtem Publish sollte folgendermassen aussehen.

Falls sich am Contract-Interface etwas geändert hat, müssen die Service-Referenzen der Client-Programme, welche diesen Service verwenden, aktualisiert werden.

- In Visual Studio das gewünschte Projekt auswählen
- Den Ordner „Service Referenz“ auswählen
- Durch einen Rechtsklick das Kontextmenü öffnen und „Update service reference“ auswählen

Soll der Service in einem neuen Projekt hinzugefügt werden sind folgende Schritte notwendig:

- In Visual Studio das gewünschte Projekt auswählen
- Durch einen Rechtsklick das Kontextmenü öffnen und „Add service reference“ auswählen
- Im Feld für die Adresse die Serviceadresse[1] eingeben
- „Go“ klicken
- Den gewünschten Service Contract auswählen
- Im Feld für den Namespace den gewünschten Namespace-Namen eingeben
- Ok klicken



[1] http://streaming.cs.technik.fhnw.ch/AlgoriaCollaborationService/AlgoriaWorksheetCollaborationService.CollaborationServiceContract.svc