CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Masterstudiengänge Informatik/Wirtschaftsinformatik

Projekt-Abschlussbericht

Multidimensionales Process Mining

vorgelegt von

Andrej Albrecht
Bernd Nottbeck
Bernhard Bruns
Christopher Licht
Jannik Arndt
Krystian Zielonka
Markus Holznagel
Moritz Eversmann
Naby Moussa Sow
Roman Bauer
Thomas Meents

Gutachter:

Thomas Vogelgesang, Timo Michelsen
Prof. Dr. Dr. h.c. H.-Jürgen Appelrath

Oldenburg, 31. März 2014

Abstract

Immer mehr Ereignisse werden automatisiert oder manuell festgehalten. Moderne Datenverarbeitungstechniken erlauben es, diese in Event Logs gespeicherten Daten durch den Einsatz von Algorithmen besser auszuwerten. So kann beispielsweise das implizierte Wissen über Prozesse in Unternehmen oder öffentlichen Einrichtungen in explizites Wissen umgewandelt werden. Dies ermöglicht ein besseres Verständnis von Abläufen und unterstützt die Identifizierung von Optimierungspotenzialen, Engpässen und deren Ursachen. Im Gesundheitswesen werden beispielsweise eine Vielzahl von Informationen erfasst. Jede Abteilung in einem Krankenhaus kennt die internen Abläufe zur Behandlung von Patienten. Ein Gesamtüberblick über den Behandlungsprozess und seine verschiedenen Teilprozesse ist jedoch häufig nur schwer möglich. Zudem können Prozessabläufe aufgrund von verschiedenen Faktoren völlig unterschiedlich sein (beispielsweise durch den Behandlungsort, den Versicherungsstatus oder die Ethnie). Diesem Problem widmete sich die Projektgruppe *Multidimensionales Process Mining* und hat daher den *Process Cube Explorer* entwickelt.

Er ist ein Forschungsframework, das von elf Studierenden in der Abteilung *Informationssysteme* der Universität Oldenburg zwischen April 2013 und März 2014 entwickelt wurde. Er ermöglicht die Auswahl von Event Logs aus einer multidimensionalen Datenstruktur und das Ausführen von *Process Mining*-Algorithmen auf diesen Daten. Darüber hinaus unterstützt er den Benutzer in der Auswertung der resultierenden Prozessmodelle. So kann aus dem impliziten Wissen der protokollierten Daten explizites Wissen über den zugrundeliegenden Prozessverlauf gewonnen werden. Dies wiederum kann genutzt werden, um die Prozesse besser zu verstehen und zu optimieren. Auch können durch die Auswahl der Eigenschaften des Event Logs Rückschlüsse auf die Gründe der unterschiedlichen Ausführungen gezogen werden.

In diesem Abschlussbericht werden die Grundlagen des *Process Mining* mit seinen Teilbereichen *Process Discovery*, *Conformance Checking* und *Model Enhancement* sowie weiteren Ansätzen wie dem *Prozessmodellvergleich* und einer *Consolidation* vorgestellt. Außerdem wird die multidimensionale Struktur der Event Logs vorgestellt, auf denen die Software arbeitet. Es folgen die Anforderungen an das Programm und Details zur Architektur und Implementierung insbesondere der Mining-Algorithmen. Abschließend wird das Vorgehen im Projekt vorgestellt und ein Ausblick gegeben, wie der entstandene Prototyp weiter genutzt werden kann.

Inhalt

1 Einleitung

Diese Arbeit stellt den *Process Cube Explorer* vor, ein Forschungsframework das es ermöglicht, *Process Mining* auf multidimensional strukturierten Daten durchzuführen. Es hilft, Event Logs gezielt anhand ihrer Eigenschaften auszuwählen und auf diesen Aufzeichnungen echter Prozessausführungen verschiedene Mining-Algorithmen auszuführen, sodass der Benutzer übersichtliche Prozessmodelle erhält. So ist es sehr einfach möglich, das implizite Wissen aus realen Prozessen zu extrahieren und darzustellen.

Event Logs sind Protokolle von Prozessausführungen. Sie können sich in ihrer Güte, Ausführlichkeit oder Größe stark unterscheiden, gemeinsam haben sie jedoch eine Auflistung von Aktivitäten und ein Attribut mit dem diese chronologisch sortiert werden können. Die zugrundeliegenden Prozesse können sehr verschieden sein: Von einfachen, regelmäßigen Prozessen wie der Fließbandproduktion über größtenteils ähnlich verlaufende Handlungen wie den Bestellvorgang eines Online-Shops bis hin zu hoch individualisierten Abläufen wie der Behandlung eines Patienten in einem Krankenhaus. Besonders bei letzteren sind weitere Eigenschaften des Prozesses interessant, anhand derer die Unterschiede der Verläufe erklärt werden können. So hängt der Behandlungsverlauf möglicherweise nicht nur von der Krankheit des Patienten, sondern auch von seinem Alter, Geschlecht oder sogar vom Krankenhaus ab.

Die untersuchten Event Logs können in eine multidimensionale Struktur gebracht werden wie sie aus dem Data Warehousing bekannt sind. Sämtliche Eigenschaften eines Falls werden als Dimensionen modelliert, sodass auf diesen Daten leicht navigiert, aggregiert und selektiert werden kann. Der *Process Cube Explorer* ermöglicht genau dies. Darüber hinaus können auch die Events eigene Dimensionen besitzen, deren Ausprägung sich ebenfalls auf die Auswahl der Daten auswirken kann.

Drei bekannte Mining-Algorithmen ermöglichen es dann, aus den ausgewählten Daten Prozessmodelle zu erstellen. Die Liste der Algorithmen ist natürlich erweiterbar. Am Ende erhält der Benutzer die verschiedenen Modelle für die von ihm gewählten Eigenschaften übersichtlich angezeigt. Die Software unterstützt ihn nun darin, Unterschiede oder Modelle mit bestimmten Eigenschaften zu finden oder die Güte der Modelle durch ein *Conformance Checking* zu überprüfen.

Die Software wurde als Forschungsframework konzipiert und ist daher an vielen Stellen erweiterbar. Sie macht es sehr einfach, neue Algorithmen oder andere Konzepte des Process Mining auszuprobieren, ermöglicht gleichzeitig aber auch eine Anwendung auf echten Daten. Entstanden ist sie in einer Projektgruppe in der Abteilung *Informationssysteme* der Universität Oldenburg unter der Betreuung von Thomas Vogelgesang und Timo Michelsen. Ein Team aus elf Studierenden hat sich ein Jahr lang mit dem Thema beschäftigt, zunächst in einer Seminarphase mit den theoretischen Grundlagen, dann praktisch mit der Implementierung der Software. Das Ergebnis ist der *Process Cube Explorer*, die vorliegende Dokumentation sowie eine Veröffentlichung im Rahmen der Informatiktage 2014 der Gesellschaft für Informatik.

1.1 Motivation

Process Mining dient vor allem dazu, implizites Wissen in explizites umzuwandeln. Implizites Wissen ist dezentral verteiltes Wissen das aus Routinen, Gewohnheiten und Einzelerfahrungen besteht. Es ist zum Beispiel auf verschiedene Mitarbeiter eines Krankenhauses verteilt: Jeder einzelne weiß für

seinen Bereich, was er wann zu tun hat. Dieses Wissen ist allerdings oft nicht Dokumentiert und Prozessmodelle die einen Überblick über den gesamten Behandlungsverlauf geben fehlen.

Die Prämisse des *Process Discovery*, der Teilbereich des *Process Mining* der sich mit dem Aufdecken des impliziten Wissens beschäftigt, ist, dass die einzelnen Aktivitäten Spuren hinterlassen. Diese Spuren werden auch *Traces* genannt, mehrere *Traces* ergeben ein *Event Log*. Aus diesen *Event Logs* können mithilfe von Mining-Algorithmen nun Prozessmodelle, sprich explizites Wissen, gewonnen werden, die den gesamten Ablauf beschreiben.

Aus diesem Grund ist *Process Mining* besonders für Bereiche interessant, in denen viele verschiedene Akteure am selben Prozess beteiligt sind und in denen sich die Prozessausführungen unterscheiden können. Ein klassischer Anwendungsfall sind Versorgungsprozesse in Krankenhäusern. Hier gibt es meistens eine relativ gute Protokollierung der Abläufe und viele beteiligte Akteure. Gleichzeitig gibt es in der Regel keine Instanz die alle Prozesse überblickt und optimieren könnte.

Die Gesundheitsforschung bringt jedoch weitere Herausforderungen mit sich: Im Gegensatz zu Autos auf einem Fließband unterscheidet sich die Behandlung von Patienten in der Regel. Es ist also unmöglich (oder zumindest nicht sinnvoll), ein Prozessmodell für die Behandlung *aller* Patienten zu erstellen. Nicht einmal für alle Patienten mit derselben Krankheit kann von einer einheitlichen Behandlung ausgegangen werden. Daher ist es wünschenswert die Event Logs nach bestimmten Eigenschaften oder Kriterien zu filtern. So können zum Beispiel alle Daten für eine klar definierte Patientengruppe betrachtet werden.

Der *Process Cube Explorer* reagiert auf diese beiden Anforderungen: Er bietet eine Möglichkeit um aus strukturierten Event Logs Teilmengen zu extrahieren und auf diesen Mining-Algorithmen auszuführen. So können die Verläufe der realen Daten sehr einfach visualisiert und miteinander verglichen werden. Das hierdurch gewonnene Wissen kann zu einer Verbesserung der Abläufe – zum Beispiel in einem Krankenhaus – führen.

1.2 Ziele

Die Aufgabenstellung der Projektgruppe *Multidimensional Process Mining* bestand in der Entwicklung einer Software für die explorative Navigation auf Prozessdaten. Vom Benutzer ausgewählte Daten sollen von einem Algorithmus zu einem Prozessmodell umgewandelt und angezeigt werden. Dafür war zunächst der Aufbau eines Data-Warehouses für multidimensionale Ereignisdaten erforderlich. Als Daten-Grundlage für die Umsetzung des Data-Warehouses diente die MIMIC II Database des PhysioNet (siehe Abschnitt 5.2.3).

Daraus ergab sich für die erste Projekthälfte zunächst das Ziel, einen vertikalen Prototypen zu entwickeln, in dem lediglich die grundlegenden Funktionen verfügbar waren. Darin enthalten war der Aufbau des Data-Warehouses, eine Benutzungsoberfläche, eine Datenbankanbindung, ein Mining-Algorithmus sowie die Visualisierung von Prozessmodellen.

In der zweiten Projekthälfte sollte der Prototyp zu einem horizontalen Prototypen ausgebaut werden. Dafür war es erforderlich, die Datenmenge des Data-Warehouses zu vergrößern, einen weiteren Mining-Algorithmus sowie eine Konformitätsprüfung zwischen Prozessmodell und Datengrundlage zu implementieren. Während der zweiten Projekthälfte sind zwei weitere Ziele hinzugekommen. Der Mining-Algorithmus *Inductive Miner* wurde aufgrund seiner Aktualität in die Zielsetzung mit aufgenommen. Zudem wollte die Projektgruppe die ursprünglich als Schnittstelle gedachte Konsolidie-

rung von Prozessmodellen in den Funktionsumfang der Software mit einbeziehen. Dadurch soll dem Benutzer das Verständnis der Ergebnisse sowie die Identifizierung interessanter Eigenschaften in den Prozessmodellen leichter fallen.

1.3 Überblick

Der Abschlussbericht ist in die folgenden Kapitel und Abschnitte gegliedert.

Zu Beginn werden in Kapitel 2 die Grundlagen zum Process Mining, zu Petrinetzen, dem multidimensionalen Ansatz, den implementierten Algorithmen und Funktionen sowie die Qualitätskennzahlen erläutert. Außerdem wird in den Grundlagen ein Beispielprozess eingeführt, auf dessen in der gesamten Dokumentation referenziert wird.

Darauffolgend werden in Kapitel 3 die Anforderungen an die zu erstellende Software erläutert. Dafür werden zunächst die funktionalen 3.1 und nicht-funktionalen 3.2 Anforderungen definiert. Folgend darauf ist die Systemabgrenzung 3.3, welches verdeutlicht, welche Funktionalitäten Bestandteil der Software sind. Abschließend im Abschnitt 3.4 ein möglicher Anwendungsfall vorgestellt.

Abschnitt 4 widmet sich der Softwarearchitektur. Anhand einer übersichtlichen Grafik ist ersichtlich, wie sich Architektur der Software zusammensetzt.

Kapitel 5 widmet sich der Beschreibung der Implementierung. Dafür werden in Abschnitt 5.1 zunächst die in diesem Projekt verwendeten Technologien vorgestellt. Des Weiteren wird in Abschnitt 5.2 der Aufbau des Data-Warehouses sowie in den Abschnitten 5.3 und 5.4 die Datenbankanbindung und die Datenauswahl erläutert. Hinzu kommt im Abschnitt 5.5 die Beschreibung der implementierten Mining-Algorithmen, im Abschnitt 5.6 die Ergebnisdarstellung sowie im Abschnitt 5.7 das Conformance Checking. Im Abschnitt 5.8 wird beschrieben wie die Software auf Funktionalität überprüft wurde.

Kapitel 6 widmet sich der Organisation innerhalb des Projekts. Es beinhaltet eine Beschreibung des verwendeten Vorgehensmodells Scrum, die Rollenverteilung sowie der Organisationssysteme Confluence und Jira.

Kapitel 7 fasst alle wesentlichen Punkte des Abschlussberichts zusammen und gibt einen Rückblick auf die erbrachte Projektarbeit. Weiterhin findet in diesem Kapitel ein Ausblick auf die Erweiterungsmöglichkeiten des erstellten Prototypen statt.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen, die zum Verständnis der Software und der weiteren Dokumentation benötigt werden, erklärt. Hierbei steht natürlich das Process Mining – als Kern dieser Arbeit – im Vordergrund. Nach einer allgemeinen Erklärung wird besonders auf die Algorithmen des *Process Discovery* eingegangen, da das Nachvollziehen der Implementierung (siehe Kapitel 5) ein Verständnis der theoretischen Grundlagen voraussetzt. Außerdem werden die Grundlagen der implementierten *Conformance Checking*-Algorithmen kurz vorgestellt. Da die umgesetzten Mining-Algorithmen alle ein Petrinetz ausgeben, werden diese in Abschnitt 2.2 kurz eingeführt. Danach werden die theoretischen Überlegungen für den Vergleich von Prozessmodellen beschrieben. Der zweite große Themenblock ist der *multidimensionale Ansatz*, der in Abschnitt 2.7 beschrieben wird. Er stellt den größten Unterschied zu bereits bestehenden Tools wie ProM [DMV+05, VBDA10] dar und bedarf daher einiger Erklärung.

Die gesamte Dokumentation wird sich auf ein Beispiel beziehen, dessen Daten auch in einem MySQL-Script mitgeliefert werden. Es behandelt einen (fiktiven) Prozessablauf, bei dem ein Patient mit einem Knochenbruch in ein Krankenhaus aufgenommen wird. Zunächst wird er am Empfang oder in der Notaufnahme von einem Mitarbeiter aufgenommen (*Aufnahme*). Es folgt die *Anamnese* durch einen Arzt. Anschließend wird dem Patienten Blut entnommen (*Blutentnahme*). Daraufhin wird er entweder geröntgt (*Röntgen*) oder es wird ein *MRT*-Bild erstellt. Da dies vor oder nach der Blutentnahme erfolgen kann werden diese Prozesse aus Sicht des Process Mining *parallel* ausgeführt. Auf jeden Fall müssen beide Prozesse abgeschlossen sein, bevor die *Befundung* geschehen kann. Danach führt ein Chirurg eine *Operation* durch, wonach ein Helfer dem Patienten einen Verband anlegt. Dieser Verband wird unter Umständen mehrfach erneuert, das bedeutet *entfernt* und wieder *angelegt*. Der letzte Prozessschritt ist die *Entlassung*.

Das Prozessmodell zu diesem Ablauf ist in Abbildung 2.1 als Petrinetz dargestellt, die genaue Semantik dieser Darstellung wird im Abschnitt 2.2 erklärt.

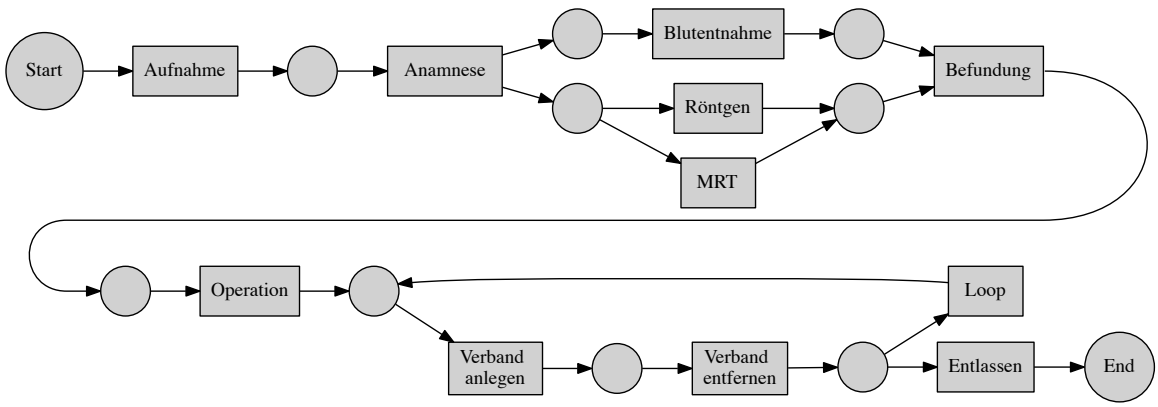


Abbildung 2.1: Beispielprozess: Ein Patient mit Knochenbruch im Krankenhaus.

Der Fall in Abbildung 2.1 hat wiederum bestimmte Eigenschaften wie die ICD-10 T14.2, das Geschlecht, Alter und Versicherung des Patienten sowie der Ort des Krankenhauses. Weiter besitzt jede einzelne Aktion (*Event*) Eigenschaften, wie zum Beispiel einen ausführenden Krankenhausmitarbeiter und eine Krankenhausabteilung. Diese Eigenschaften werden im Abschnitt 2.7 relevant.

2.1 Process Mining

Process Mining ist eine vergleichsweise junge Wissenschaftsdisziplin, die zwischen Computational Intelligence und Data Mining auf der einen Seite und Prozessmodellierung und Analyse auf der anderen einzuordnen ist [PM12]. Es kann als ein spezielles Data Mining verstanden werden, bei dem eine ganz bestimmte Art von Wissen das Ziel ist, nämlich das Prozesswissen [Sch04]. Dieses jedoch liegt oftmals nur als verteiltes Hintergrundwissen in den Köpfen der beteiligten Mitarbeiter vor. Wie die Aufgaben in ihrem jeweiligen Bereich abzulaufen haben, wissen sie aus ihrem Arbeitsalltag heraus. Das Wissen über den gesamten Prozess fehlt jedoch in den meisten Fällen.

Bei den Daten, die das Process Mining verwendet, handelt es sich um Aufzeichnungen der Prozessausführung. Diesen Ausführungen liegt implizit Prozesswissen zugrunde. Die Ausführenden der Prozesse wissen was, wie, wann und wo zu tun ist [Sch04]. Mit Techniken des Process Mining soll dieses Wissen in Form von Modellen explizit verfügbar gemacht werden, damit es gesichert, übertragen und wiederverwendet werden kann. Die Träger des Wissens stehen jedoch hierfür oftmals nicht oder nur eingeschränkt zur Verfügung. Eine Möglichkeit, diesem Problem zu begegnen, besteht darin, von den Prozessausführungen auf das Prozesswissen zu schließen. Die Grundidee von Process Mining ist es, reale Prozesse – im Gegensatz zu vermuteten oder angenommenen Prozessen – durch Extrahieren von Wissen aus Event Logs heutiger Informationssysteme zu erkennen, zu überwachen und zu verbessern [Aal11]. Diese Gewinnung von formalisiertem Prozesswissen aus den Aufzeichnungen von Prozessausführungen ist das Ziel und der Nutzen von Process Mining [Sch04].

In vielen Programmen und vielen Geschäftsabläufen werden die Aktivitäten in Log-Dateien aufgezeichnet. Jedoch speichern die meisten Systeme solche Informationen in unterschiedlichen Formaten und Strukturen. Beispielsweise können die Ereignisdaten über mehrere Tabellen verteilt sein oder müssen aus den ausgetauschten Informationen von Subsystemen erfasst werden. Die Extraktion von Daten ist ein Bestandteil jeder Process Mining Bemühung [Aal11]. So werden diese Daten in Event Logs zusammengetragen und aufbereitet, um sie für das Process Mining nutzen zu können.

Ein Event Log protokolliert die Ausführung eines oder mehrerer Prozesse, womit jede Art von Ablauf gemeint ist, z. B. die Bestellung von Waren in einem Onlineshop oder die Versorgung eines Patienten in einem Krankenhaus. Es besteht aus einer Menge von Events, welche im Event Log als Zeilen dargestellt sind. Ein Event ist die Ausführung einer Aktivität (Activity) und gehört zu einem konkreten Case . Es kommt in einem Event Log genau einmal vor, wobei dieselbe Aktivität in verschiedenen Events mehrfach vorkommen kann. Es kann weitere Eigenschaften wie z. B. Zeitstempel, Rollen oder Geschlecht beinhalten. Der Begriff Case bezeichnet eine Sammlung zusammengehöriger Events und weiterer Daten, die über alle Events konstant sind.

In Tabelle 2.1 ist ein Beispiel für ein Event Log mit drei Cases abgebildet. Die erste Spalte identifiziert die Case-Instanz. Zu jedem Case gehört genau ein Patient. Die einzelnen Zeilen stellen Events dar, die unterschiedliche Aktivitäten, Behandelnde und Zeitstempel besitzen. Im Event Log sind drei Ausführungen (also Cases) des Prozesses aus Abbildung 2.1 protokolliert.

Die Daten aus einem Event Log dienen als Ausgangspunkt für das Process Mining. Es gibt drei Arten des Process Mining, die diese Event Logs nutzen können. Diese sind Process Discovery (siehe Abschnitt 2.3), Conformance Checking (siehe Abschnitt 2.4) und Model Enhancement (siehe Abschnitt 2.5).

Case	Aktivität	Behandelnder	Patient	Zeitstempel
1	Aufnahme	Herr Buchholz	Herr Meyer	2013-2-24, 08:05:01
2	Aufnahme	Herr Buchholz	Frau Hanse	2013-2-24, 08:15:23
1	Anamnese	Dr. Köhler	Herr Meyer	2013-2-24, 08:25:31
3	Aufnahme	Herr Buchholz	Herr Müller	2013-2-24, 08:29:23
2	Anamnese	Dr. Köhler	Frau Hanse	2013-2-24, 08:45:38
1	Blutentnahme	Frau Koch	Herr Meyer	2013-2-24, 08:47:13
3	Anamnese	Dr. Köhler	Herr Müller	2013-2-24, 08:52:38
2	Blutentnahme	Frau Koch	Frau Hanse	2013-2-24, 08:56:32
1	Röntgen	Frau Weishaupt	Herr Meyer	2013-2-24, 09:13:37
3	Entlassung	Herr Buchholz	Herr Müller	2013-2-24, 09:17:42
2	MRT	Frau Wessling	Frau Hanse	2013-2-24, 09:24:41
1	Befundung	Dr. Köhler	Herr Meyer	2013-2-24, 09:32:56
2	Befundung	Dr. Köhler	Frau Hanse	2013-2-24, 09:51:36
2	Verband anlegen	Frau Müller	Frau Hanse	2013-2-24, 10:09:11
2	Entlassung	Herr Buchholz	Frau Hanse	2013-2-24, 10:17:42
1	Operation	Dr. Köhler	Herr Meyer	2013-2-25, 08:35:23
1	Verband anlegen	Frau Müller	Herr Meyer	2013-2-25, 09:05:11
1	Verband entfernen	Dr. Köhler	Herr Meyer	2013-2-29, 10:47:13
1	Entlassung	Herr Buchholz	Herr Meyer	2013-2-29, 11:13:47

Tabelle 2.1: Beispiel eines Event Logs

2.2 Petrinetze

Ein Petrinetz ist ein gerichteter Graph bestehend aus Knoten und Kanten, wobei zwischen zwei Typen von Knoten unterschieden wird: Stellen und Transitionen (siehe Abbildung 2.2). Stellen sind die passiven Bestandteile eines Petrinetzes und werden als Ellipse oder Kreis dargestellt. Eine Stelle kann eine, keine oder mehrere Markierungen (*Token*) enthalten. Durch eine Markierung wird der Zustand eines modellierten Systems abgebildet. Eine Transition steht für eine Aktivität und wird als ein Rechteck dargestellt. Die Kanten verbinden immer nur unterschiedliche Arten von Knoten und repräsentieren einen Kausalzusammenhang des modellierten Systems [W.10].

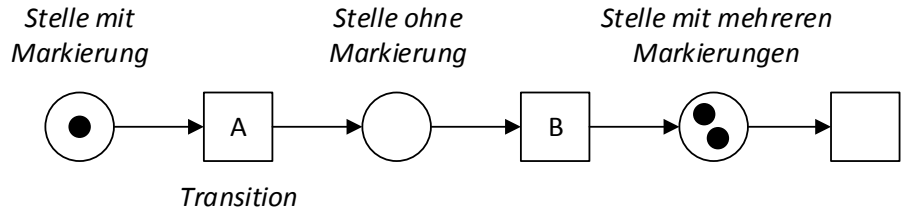


Abbildung 2.2: Petrinetz [W.10]

Enthält eine Stelle mindestens eine Markierung, so sind die direkt nachfolgenden Transitionen feuerbereit. Als Feuern wird der Übergang des Systems von einem Zustand in einen anderen bezeichnet. Eine Transition t und alle Stellen, die mit ihr verbunden sind, werden als eine Umgebung bezeichnet. Innerhalb der Umgebung wird zwischen Vorbedingungen und Nachbedingungen unterschieden. Vor-

bedingungen sind die Stellen, die in dem gerichteten Graph direkt vor der Transition stehen und als Nachbedingungen werden die Stellen bezeichnet, die direkt nach der Transition folgen. Beim Feuern wandert eine Markierung aus jeder Vorbedingung zu jeweils jeder der Nachbedingungen.

Die Konstellation der verschiedenen Knoten, Stellen und Transitionen bestimmt die Kausalität des darzustellenden Petrinetzes. Folgen auf eine Stelle mehr als eine Transition so handelt es sich um eine XOR-Split-Konstellation . Dabei ist zu beachten, dass nur eine der Transitionen ($t1$ oder $t2$) gefeuert werden kann (siehe Abbildung 2.3). Bei einer XOR-Join-Konstellation genügt es bereits, wenn eine der beiden Transitionen ($t3$ oder $t4$) gefeuert wird, um die nachfolgende Stelle $s6$ zu markieren.

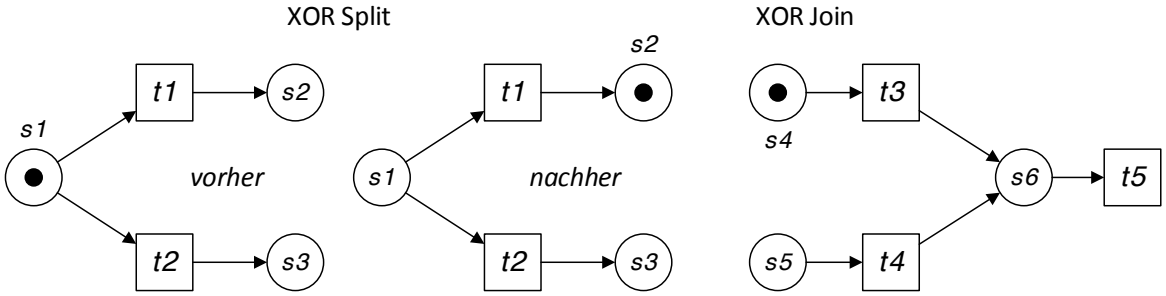


Abbildung 2.3: XOR Konstellation [W.10]

Folgen auf eine Transition mehr als eine Stelle so handelt es sich um eine AND-Split-Konstellation . Hierbei ist zu beachten, dass, nachdem die Transition $t1$ gefeuert wurde, alle nachfolgenden Stellen markiert werden. Bei einer AND-Join-Konstellation müssen alle Vorbedingung markiert sein, damit die Transition $t4$ gefeuert werden kann (siehe Abbildung 2.4).

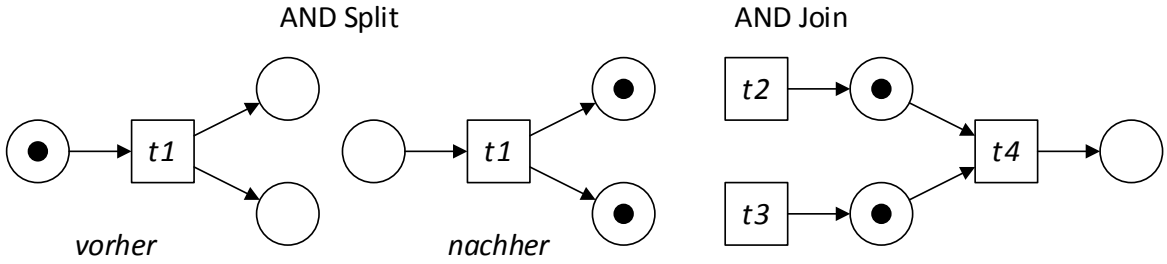


Abbildung 2.4: AND Konstellation [W.10]

Eine besondere Form der Petrinetze sind Workflow-Netze. Sie besitzen genau eine Start- und eine End-Stelle. Am Anfang besitzt nur die Start-Stelle eine Markierung, nachdem das Netz durchgelaufen wurde besitzt nur noch die End-Stelle eine Markierung.

2.3 Process Discovery

Process Discovery ist das wichtigste der drei Teilgebiete des Process Mining. Seine Aufgabe ist es, aus einem Event Log ein Prozessmodell zu generieren. Dies kann, abhängig vom zugrundeliegenden Prozess, verschiedene Schwierigkeiten bieten. Zum Beispiel kann es bei einer Teilmenge der Aktivitäten egal sein, in welcher Reihenfolge sie ausgeführt werden. Im Prozessmodell wird dies

durch eine AND-Beziehung dargestellt. Das Event Log repräsentiert jedoch eine konkrete Ausführung, bei der die Reihenfolge definiert ist. Es müssen also mehrere Ausführungen desselben Prozesses betrachtet werden, damit entdeckt werden kann, wenn sich die Reihenfolge bestimmter Events häufiger unterscheidet.

Auch ist es möglich, dass an einem bestimmten Punkt im Prozessverlauf jeweils nur eine von mehreren möglichen Aktivitäten ausgeführt werden kann. Hier sind wieder mehrere Event Logs nötig, um zu erkennen, dass diese Aktivitäten in einer XOR-Beziehung stehen.

Eine weitere große Herausforderung ist Rauschen (engl. *Noise*). Aktivitäten die nur sehr selten ausgeführt werden oder fehlerhaft im Event Log eingetragen wurden, also eigentlich nicht in das Prozessmodell gehören, müssen als solche erkannt und ignoriert werden. Insbesondere in Kombination mit den AND- und XOR-Parallelitäten sowie optionalen Aktivitäten ist es hier sehr schwierig die richtige Balance zwischen Precision und Generalization zu finden: ein präzises Prozessmodell passt genau auf die im Event Log vorhandenen Ausführungen, lässt aber keine weiteren zu. Ein generalisierendes Prozessmodell lässt unter Umständen zu viele falsche Ausführungen zu (siehe auch Abschnitt 2.3.5). Modelle mit hoher Precision neigen außerdem dazu, viele verschiedene Ausführungen darzustellen, was zu so genannten Spaghetti-Modellen führt [CA07].

Event Logs geben außerdem keine Auskunft darüber, ob eine oder mehrere Aktivitäten wiederholt werden können (*Loops*) oder ob zwischen weiter entfernten Aktivitäten eine Abhängigkeit besteht (*Long Distance Dependencies*).

Die Herausforderung beim Process Discovery besteht also vor allem darin, qualitativ hochwertige Prozessmodelle aus schwach strukturierten Prozessen zu extrahieren. Je hochwertiger ein Prozessmodell ist, desto besser kann es genutzt werden um einen Prozess zu analysieren und zu verbessern. Aus diesem Grund gibt es verschiedene Mining-Algorithmen, die unterschiedlich mit den Event Log-Daten umgehen. In der Software werden außerdem diverse Qualitätskennzahlen angezeigt (siehe Abschnitt 5.6.3), die weitere Informationen zum Prozessmodell und dem Verlauf des Mining-Prozesses wiedergeben.

2.3.1 Alpha Miner

Einer der ersten Algorithmen, um Prozesse aus Event Logs zu gewinnen war der Alpha Miner. Das Ergebnis dieses Algorithmus ist ein Workflow-Netz. Der Alpha Miner basiert auf den vier Anordnungsbeziehungen $>_L$, $\rightarrow_L$, $\neq_L$ und $||_L$, welche aus den Event Logs abgeleitet werden können.

- $A >_L B$, wenn ein *Trace* $\sigma =< t_1, t_2, t_3, ...t_n >$ und $i \in \{1, ..., n-1\}$ mit $\sigma \in L$ existiert und $t_i = A$ und $t_{i+1} = B$,
- $A \Rightarrow_L B$, wenn $A >_L B$ und $B \not> _L A$
- $A \neq_L B$, wenn $A \not> _L B$ und $B \not> _L A$
- $A||_L B$, wenn $A >_L B$ und $B >_L A$

Sei das Event Log $L_1 = [< a, b, c, d >^5, < a, c, b, d >^8, < a, e, d >^9]$, wobei ein Element mit spitzen Klammern ein *Trace* (der Event Log eines Case) ist. Der Exponent gibt an wie oft der *Trace* im Event Log vorkommt. In L_1 gilt $c >_{L_1} d$, weil d direkt auf c in *Trace* $< a, b, c, d >$ folgt und es gilt $d \not> _{L_1} c$, weil c in keinem *Trace* direkt auf d folgt. Diese beiden Beziehungen führen dazu das $c \rightarrow_{L_1} d$

gilt. Weiterhin gilt $b \neq_{L_1} e$, weil $b \not\gtrdot_{L_1} e$ und $e \not\gtrdot_{L_1} b$. Ebenfalls gültig ist beispielsweise $b||_{L_1} c$, weil $c >_{L_1} b$ und $b >_{L_1} c$.

Die Anordnungsbeziehungen werden für jedes Paar von Aktivitäten abgeleitet. Zur Visualisierung eignet sich eine Footprint-Matrix. Im Fall von L_1 würde sie, wie in Tabelle 2.2 dargestellt, aussehen.

	a	**b**	**c**	**d**	**e**
a	$\#_L$	$\rightarrow_L$	$\rightarrow_L$	$\#_L$	$\rightarrow_L$
b	$\leftarrow_L$	$\#_L$	$\|\|_L$	$\rightarrow_L$	$\#_L$
c	$\leftarrow_L$	$\|\|_L$	$\#_L$	$\rightarrow_L$	$\#_L$
d	$\#_L$	$\leftarrow_L$	$\leftarrow_L$	$\#_L$	$\leftarrow_L$
e	$\leftarrow_L$	$\#_L$	$\#_L$	$\rightarrow_L$	$\#_L$

Tabelle 2.2: Footprint-Matrix für L_1

Nach der Extraktion der Anordnungsbeziehungen werden diese auf typische Prozessmuster abgebildet (siehe Abbildung 2.5). Wenn auf Aktivität a Aktivität b folgt, wird $a >_L b$ abgeleitet. Im Prozessmodell taucht b als Nachfolger von a auf (siehe Abbildung 2.5a). Wenn nach a eine Wahlmöglichkeit zwischen b und c besteht und b niemals auf c und umgekehrt folgen kann, wird $a >_L b$, $a >_L c$ und $b \neq_L c$ abgeleitet. Dargestellt wird das im Prozessmodell mit einem XOR-Split (siehe Abbildung 2.5b). Wenn $a >_L c$, $b >_L c$ und $a \neq_L b$ gilt, bedeutet dies, dass nach a oder b immer c ausgeführt werden muss und a und b niemals aufeinander folgen. Dies wird im Prozessmodell auf einen XOR-Join abgebildet (siehe Abbildung 2.5c). Wenn $a >_L b$, $a >_L c$ und $b||_L c$, kann nach a sowohl b als auch c ausgeführt werden und b und c können direkt aufeinander folgen. Zur Darstellung wird ein AND-Split (siehe Abbildung 2.5d) verwendet. Wenn $a >_L c$, $b >_L c$ und $a||_L b$, müssen a und b in beliebiger Reihenfolge aufeinanderfolgen und anschließend mit einem AND-Join in ein c synchronisiert werden (siehe Abbildung 2.5e).

Da der Alpha Miner in dieser Form nicht in der Lage ist, kurze Loops (der Länge 1 und 2) zu erkennen, wurde später der ursprüngliche Alpha Miner als Alpha Miner+ und Alpha Miner++ erweitert. Um kurze Loops erkennen zu können, muss die Vollständigkeit der Event Logs gewährleistet werden. Dann müssen die Anordnungsbeziehungen umdefiniert und neue eingeführt werden. Mit diesen Änderungen können nun Loops der Länge 2 erkannt werden. Damit auch Loops der Länge 1 erkannt werden können, sind vor und nach der Anwendung des Algorithmus weitere Schritte notwendig (pre-/post-processing). So müssen die Loops der Länge 1 zuerst aus den Event Logs entfernt werden, bevor der Alpha Miner die Grundstruktur des Graphen erkennen kann. Nach der Anwendung des Algorithmus müssen diese Loops wieder in den Prozess eingebunden werden.

Zum besseren Verständnis, wie der Alpha Miner+ und Alpha Miner++ funktioniert, sei an dieser Stelle auf [MDAW05] und [Wes12] verwiesen.

2.3.2 Heuristic Miner

Der *Heuristic Miner* nach [WAM06] berechnet die Abhängigkeit zweier Events und betrachtet nur die Beziehungen, deren Abhängigkeit über einem Schwellenwert liegen. So wird Rauschen automatisch unterdrückt und es entstehen, wenn die Datenbasis groß genug ist, sehr kompakte Modelle. Der Algorithmus kann in zwei Schritte unterteilt werden. Im ersten wird ein Abhängigkeitsgraph berechnet,

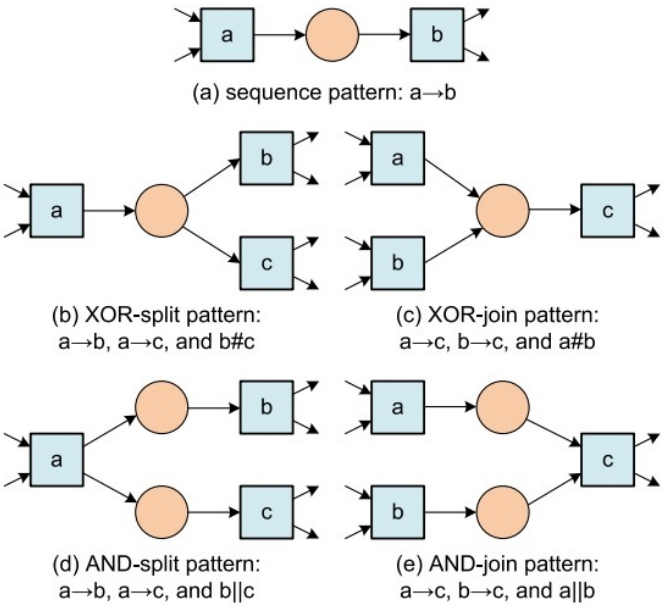


Abbildung 2.5: Typische Prozessmuster und Spuren, die sie in einem Event Log hinterlassen [Aal10]

im zweiten wird für parallele Events die Art der Parallelität (*and* oder *xor*) berechnet. Dabei wird direkt ein Petrinetz erstellt.

Um den Abhängigkeitsgraphen aufzustellen, wird zunächst für alle Kombinationen von Events a und b die Abhängigkeit anhand der Formel

$$a \Rightarrow b = \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1}$$

mit $|a > b|$ als die Häufigkeit, mit der b auf a in $\mathcal{W}$ folgt, berechnet. Diese Funktion bildet auf das Intervall $(-1, 1)$ ab, wobei Werte nahe der Null bedeuten, dass die beiden Events unabhängig voneinander sind. Werte nahe der 1 deuten auf eine Abhängigkeit hin und negative Werte sprechen für eine Abhängigkeit in umgekehrter Reihenfolge. Es gilt $a \Rightarrow b = -1 \cdot (b \Rightarrow a)$.

Da für *jede* Kombination aus zwei Events das gesamte Event Log durchsucht werden muss, ist dies ein sehr rechenintensiver Teil des Algorithmus. In der resultierenden Abhängigkeitsmatrix sind die Einträge des unteren Dreiecks jeweils die Inversen der Einträge im oberen Dreieck. Somit reicht es, nur die obere Dreiecksmatrix zu berechnen. Dadurch wird der Rechenaufwand um etwa die Hälfte reduziert.

Aus der Abhängigkeitsmatrix wird ein Abhängigkeitsgraph erstellt, indem die Spalte mit den meisten negativen Werten als Startknoten verwendet wird und Verbindungen zu allen Knoten gezogen werden, deren Abhängigkeit über einem heuristisch festgelegten Schwellenwert liegt. Ein Beispiel für einen Abhängigkeitsgraphen verdeutlicht Abbildung 2.6.

Der Abhängigkeitsgraph ist noch kein Petrinetz, da er keine Stellen besitzt und somit auch noch kein Prozessmodell darstellen kann. Insbesondere hat er keine AND- und XOR-Splits und -Joins, die im Event Log auch nicht explizit zu finden sind. Den richtigen Split zu finden ist jedoch nicht

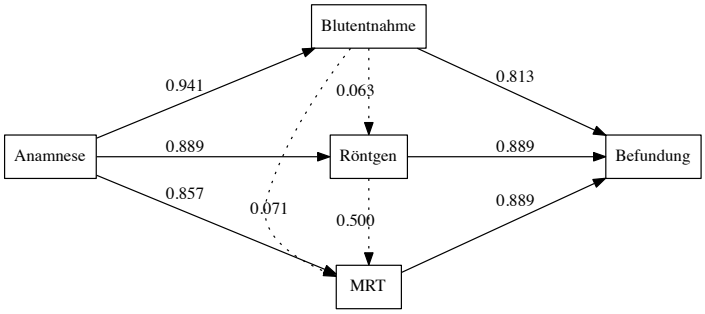


Abbildung 2.6: Ein typischer Abhängigkeitsgraph aus dem Heuristic Miner.

schwierig: für je zwei Folgetransitionen einer Transition gilt entweder, dass sie im weiteren Verlauf *beide* auftauchen ($\Rightarrow$ AND-Split) oder nur *einer* der beiden auf ($\Rightarrow$ XOR-Split) vorhanden ist. Formal bedeutet das:

$$a \rightsquigarrow (b \wedge c) = \frac{|b > c| + |c > b|}{|a > b| + |a > c| + 1}$$

Es werden also alle Vorkommnisse, in denen c auf b oder b auf c folgt, ins Verhältnis zur Anzahl der Vorkommnisse von b und c gesetzt. Dabei werden statt b und c die Kombinationen ab und ac betrachtet, damit Loops das Ergebnis nicht verfälschen. Die Funktion bildet ab auf $[0, 1)$, wobei Werte < 0.1 eine XOR-Beziehung von b und c bedeuten, während Werte > 0.1 auf eine AND-Beziehung schließen lassen. Der empirische Schwellenwert von 0.1 stammt aus [WAM06].

Es ist zu beachten, dass die Formel *immer* eine Entscheidung für XOR oder AND ausgibt, daher ist es nicht sinnvoll, sie auf sequentielle Events anzuwenden. Aus diesem Grund muss dieser Schritt *nach* dem Erstellen des Abhängigkeitsgraphen durchgeführt werden.

Wenn die Parallelitäten erkannt sind, kann daraus direkt ein Petrinetz erzeugt werden. Dieses bildet unter Umständen gewisse Long Distance Dependencies noch nicht ab, allerdings unterstützt der *Heuristic Miner* nach [WAM06] dies auch noch nicht.

2.3.3 Inductive Miner infrequent

Der *Inductive Miner infrequent* (IMi) ist im Process Mining einer der aktuellsten Mining-Algorithmen und basiert auf dem *Inductive Miner* [LFA13a]. Die Datenstruktur des Prozessmodells ist ein Block-strukturierter binärer Prozess-Baum. In dieser Datenstruktur werden die Knoten als Blätter *leaf* und die Prozessmodell-Operatoren (*Sequence*, *Loop*, *Exclusive Choice (XOR)*, *Parallel* (AND)) als Zweige eingefügt. Ein Leaf bzw. ein Prozessmodell-Operator wird erst in den Prozess-Baum eingefügt, sobald dieser durch ein *Divide-and-Conquer*-Verfahren in einen kleinstmöglichen Prozessblock aufgeteilt ist [LFA13a].

Das Divide-and-Conquer-Verfahren verläuft beim *Inductive Miner* wie folgt: Es werden ein Direct-Follow-Graph und ein Eventual-Follow-Graph gebildet. Der Direct-Follow-Graph stellt die Häufigkeit mit der ein Event auf den direkten Nachbarn im Event Log folgt dar. Der Eventual-Follow-Graph hingegen besagt, in welcher Häufigkeit ein Event aus der gesamten Menge der Events im Event Log auf ein anderes folgt. Besteht eine Verbindung zwischen zwei oder mehreren Events und sind diese Events

disjunkt zu den anderen Events, so können diese Events getrennt betrachtet werden. [LFA13a] Der Inductive Miner trennt diese disjunkten Mengen von den anderen Events aus dem Event Log. Dieses wird auch als *Cut* bezeichnet. Ein Cut repräsentiert somit einen Prozessmodell-Operator. Es werden so viele Cuts durchgeführt bis alle Verbindungen der Events aus der disjunkten Menge verarbeitet und keine Cuts mehr möglich sind. In der kleinsten Menge entspricht dies genau einem Event und ist somit ein Leaf. Das Ergebnis (ein Teilbaum) wird im Prozess-Baum gespeichert. Abschließend werden die Teilbäume zu einem gesamten Prozess-Baum zusammengefügt. Dieser Prozess-Baum kann rekursiv in ein Petrinetz umgewandelt werden (siehe Abschnitt 2.2) [LFA13a, LFA13b].

Der Mining-Algorithmus *Inductive Miner infrequent* unterscheidet sich vom *Inductive Miner* dadurch, dass seltene Events sowie seltene Verbindungen zwischen den Events bei jedem Cut herausgefiltert werden.

Tabelle 2.3 ordnet den Prozessmodell-Operatoren aus dem Petrinetz Symbole zu.

Prozessmodell-Operator	**Symbol**
Sequence	$\rightarrow$
Loop	$\circlearrowleft$
Exclusive Choice (XOR)	X
Parallel	$\wedge$

Tabelle 2.3: Symbole der Prozessmodell-Operatoren [LFA13a]

Abbildung 2.7 zeigt den Beispielprozess aus Abbildung 2.1 als Prozess-Baum. Jeder Knote besitzt einen linken und einen rechten Zweig. Das oberste Element stellt die Wurzel (root) dar. Vom Root-Element ausgehend wird zunächst der linke Teilbaum und anschließend der rechte Teilbaum erstellt.

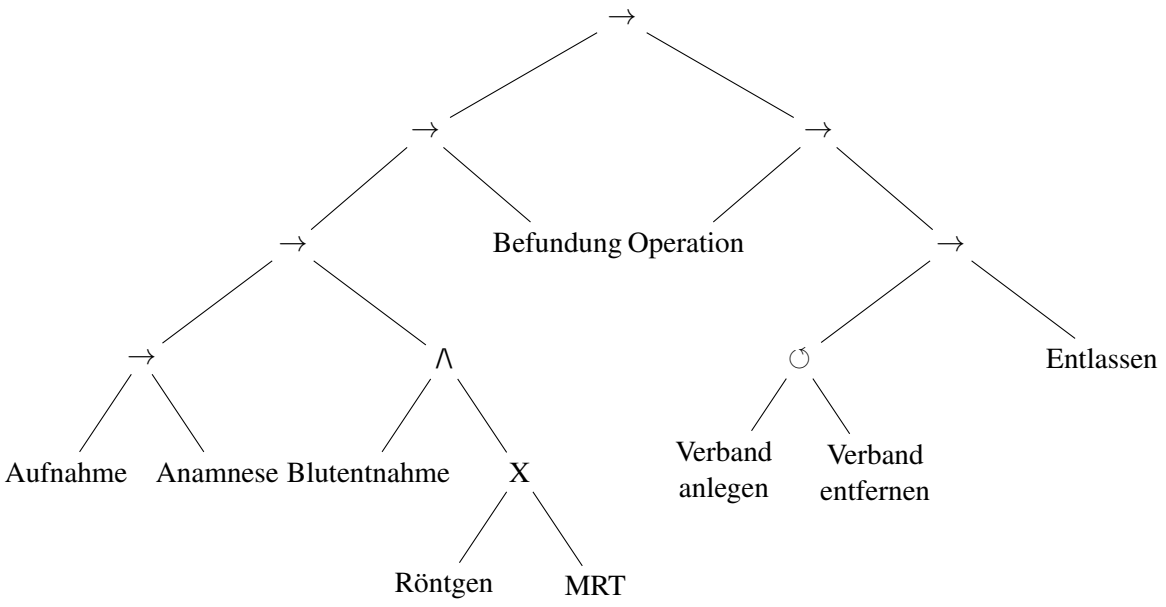


Abbildung 2.7: Prozess-Baum des Beispielprozess

2.3.4 Weitere Miner

In der Seminarphase am Anfang des Projektes hat sich die Projektgruppe mit vier Mining-Algorithmen näher beschäftigt. Davon wurde der *Heuristic Miner* umgesetzt, weil er relativ gut nachvollziehbar ist und vergleichsweise gute Ergebnisse liefert. Des weiteren wurde der *Alpha Miner* implementiert, weil er weit verbreitet ist und trotz einiger Schwächen als Standard in vielen Programmen vorhanden ist und so als ein guter Vergleich dient. Gegen Ende der Projektlaufzeit ist der *Inductive Miner - infrequent* veröffentlicht worden und ist also einer der aktuellsten Algorithmen. Daher wurde dieser ebenfalls umgesetzt.

Neben den drei ausgewählten Algorithmen gibt es jedoch noch viele weitere mit teilweise komplett unterschiedlichen Ansätzen. Aus dieser Menge sollen zwei, mit denen sich die Projektgruppe am Anfang kurz beschäftigt hat, die jedoch nicht implementiert wurden, vorgestellt werden.

Der *Genetic Miner* basiert auf genetischen bzw. evolutionären Algorithmen, die aus dem Bereich der Computational Intelligence stammen. Die Grundidee des Ansatzes ist, eine Menge von möglichen Lösungskandidaten (die *Population*) zu mutieren, d. h. zufällig zu verändern, und zu rekombinieren, d. h. in Teilen neu zusammenzusetzen, danach für alle Kandidaten eine *Fitness* zu berechnen und nur jene weiter zu betrachten, deren Fitness ausreichend groß ist. Diese "Überlebenden" bilden die Basis für die nächste Iteration von Mutationen.

Im Kontext des Process Mining bedeutet das konkret, dass eine Population von Prozessmodellen zufällig erzeugt wird. Die vorkommenden Events sollten etwa dem Event Log entsprechen, müssen diesen aber nicht vollständig enthalten, da dieser auch Rauschen enthalten kann. Danach wird für jedes Modell dessen Fitness berechnet. Hierfür werden Methoden des Conformance Checking (siehe Abschnitt 2.4) benötigt. Die besten Modelle können nun weiter verändert werden, um die Fitness weiter zu erhöhen. Mutationen können z. B. das zufällige Auswechseln, Hinzufügen oder Entfernen von Events sein, Änderungen der Reihenfolge oder der Parallelitäten. Für eine Rekombination würden erfolgreiche Teilstücke der Modelle miteinander kombiniert werden.

In der Theorie kann der *Genetic Miner* perfekte Prozessmodelle erzeugen, da er kein Wissen über längere Verläufe, Long Distance Dependencies oder komplizierte Parallelitäten benötigt, sondern diese durch den "zielstrebigen Zufall" irgendwann automatisch erzeugt werden. Allerdings ist dieser Prozess nicht deterministisch, d. h. es ist weder garantiert, dass dieses Ziel erreicht wird, noch dass in einer bestimmten Zeit oder Anzahl von Iterationen ein zufriedenstellendes Ergebnis erzielt wird. Besonders aus diesem Grund haben wir uns dagegen entschieden, den Algorithmus zu implementieren.

Der *Fuzzy Miner* nutzt Fuzzylogik als Grundlage. Diese geht über die Boolesche Logik hinaus und erlaubt auch Zugehörigkeitswerte zwischen 0 und 1. So können unscharfe Begriffe in ein maschinenverständliches System übersetzt werden, z. B. wenn Begriffe wie "warm" und "kalt" in Temperaturen umgesetzt werden sollen oder "schnell" und "langsam" in Geschwindigkeiten.

Der Mining-Algorithmus nutzt vor allem zwei Konzepte. Die *Signifikanz* sagt aus, wie relevant eine Aktivität für das gesamte Modell ist, die *Korrelation* sagt, wie stark zwei aufeinanderfolgende Aktivitäten zusammenhängen. Ein initiales Modell das sämtliche Aktivitäten und Beziehungen enthält (wie z. B. der Alpha Miner es erstellt) wird in drei Phasen reduziert. Zunächst werden durch Löschen von Kanten Konflikte gelöst und im zweiten Schritt Kanten mit geringer Korrelation entfernt. Im letzten Schritt werden die Aktivitäten abhängig von ihrer Signifikanz zusammengefasst oder entfernt.

Der *Fuzzy Miner* besitzt eine sehr hohe Flexibilität, was ihn in manchen Situationen sehr nützlich macht, die Bedienung meistens jedoch erschwert. Aus diesem Grund haben wir uns dafür entschieden, ihn zunächst nicht zu implementieren.

2.3.5 Qualitätskennzahlen

Da die Mining-Algorithmen unterschiedliche Ansätze verfolgen, erstellen sie aus demselben Event Log nicht unbedingt dasselbe Prozessmodell. Vor allem der Umgang mit Rauschen macht hierbei große Unterschiede. Unvollständige Event Logs können ebenfalls zu der Generierung von falschen Prozessmodellen führen. Damit eine Aussage gemacht werden kann, wie gut ein Prozessmodell aus einem Event Log generiert wurde oder die Qualität aus einer Menge von generierten Prozessmodellen bewertet werden kann werden Qualitätskennzahlen benötigt. Für gewöhnlich unterscheidet man zwischen Fitness, Precision, Generalization und Simplicity [Aal11]. Da die Qualitätskennzahlen teils in einer Wechselbeziehung zueinander stehen und sich somit gegenseitig beeinflussen, ist es je nach Anwendungsfall notwendig sie gegeneinander abzuwägen. In vielen Fällen soll das Prozessmodell ein möglichst detailgetreues Abbild des Verhaltens aus dem Event Log abbilden und somit die Mehrzahl der Cases wiedergeben. In anderen Fällen soll ein möglichst einfaches und Komplexität-reduziertes Abbild aus dem Event Log erstellt werden soll, welches hierdurch bewusst auf Cases verzichtet. Im Folgenden werden die oben genannten Qualitätskennzahlen kurz vorgestellt.

Fitness

Eine perfekte Fitness bedeutet, dass das Prozessmodell das Verhalten aus dem Event Log vollständig wiedergibt. Dabei kann die Fitness auf unterschiedliche Weise definiert werden. Auf der Case-Ebene gibt die Fitness den Grad des Cases aus einem Event Log an, welche auf das generierte Prozessmodell abspielbar sind. Eine schlechte Fitness bedeutet, dass eine Vielzahl von Cases auf das Prozessmodell nicht abspielbar sind, während bei einer Fitness von 100% alle abspielbar sind. Das Token Replay Verfahren nutzt diese Definition von Fitness. Auf der Event-Ebene gibt die Fitness den Anteil der Events an, die sich an das Verhalten aus dem Event Log halten. Eine schlechte Fitness würde bei dieser Definition bedeuten, dass einige Events nicht das Verhalten aus dem Event Log darstellen. An diese Definition, der Event-Ebene lehnt sich das Comparing Footprint Verfahren an [Aal11].

Precision

Ein Prozessmodell sollte möglichst nur das Verhalten zulassen, welches sich aus dem Event Log ergibt. Dieses Verhalten wird durch die Precision bewertet. Eine niedrige Precision deutet darauf hin, dass das Prozessmodell weitere Verhaltensweisen zulässt als im Event Log eigentlich möglich sind. Ein Flower Model[1] würde beispielsweise alle möglichen Reihenfolgen der Events zulassen, auch wenn einige Reihenfolgen laut Event Log nicht möglich sein sollten. Wird nur das Verhalten aus dem Event Log im Prozessmodell zugelassen, deutet dies auf eine hohe Percision hin [Aal11].

[1] Ein Flower Model stellt alle Abläufe eines Event Logs, aber auch jedes anderen Event Logs, das sich auf die selbe Menge an Events bezieht, dar. Eine Stelle verweist auf viele Transitionen, die wiederum auf die selbe Stelle zurück verweisen. Das Model ähnelt einer Blume.

Generalization

Die Generalization gibt an, in wie weit das Verhalten aus dem Event Log im Prozessmodell verallgemeinert wurde. Ein Prozessmodel soll generalisieren, dabei aber nicht das Verhalten welches in dem Event Log vorhanden ist einschränken. Events mit dem gleichen Namen sollten, bei einem Petrinetz als Prozessmodell, in einer Transition zusammengefasst werden [Aal11].

Simplicity

Die Komplexität des Prozessmodells sollte auf das Nötigste reduziert werden. Diese Kennzahl steht in einem engen Konflikt mit den anderen Qualitätskennzahlen. Wo bei den anderen Kennzahlen ein möglichst genaues Abbild des Verhaltens angestrebt wird, soll mit dieser Kennzahl erreicht werden, dass ein möglichst einfaches Prozessmodell erstellt wird [Aal11].

2.4 Conformance Checking

Beim Process Discovery können durch die Mining-Algorithmen nicht immer Prozessmodelle generiert werden, die die Event Logs und damit die aufgezeichnete Realität genau widerspiegeln. Die Gründe dafür können zum einen falsch oder gar nicht aufgezeichnete Vorgänge in den Event Logs sein. Zum anderen kann es auch an dem Grad der Genauigkeit (Precision) liegen der mit dem Prozessmodell erreicht werden soll. Ein Prozessmodell, in dem alle Cases dargestellt werden, kann bei einer großen Anzahl von Transitionen und Kanten unübersichtlich aussehen. Daher ist es in einigen Fällen ausreichend wenn nur ein großer Teil der Cases aus dem Event Log auf das Prozessmodell abspielbar sind, wenn dafür die Komplexität der Prozessmodelle reduziert werden kann. Einige Mining-Algorithmen können Cases die nur selten auftreten während des Minings entfernen. Um nach dem Mining eine Aussage über die Konformität zu machen, muss diese überprüft werden können. Dies erfolgt mit Hilfe des Conformance Checking. Das Conformance Checking wird folgendermaßen beschrieben:

„*Conformance checking relates events in the event log to activities in the process model and compares both. The goal is to find commonalities and discrepancies between the modeled behavior and the observed behavior.* “ [Aal11]

Das Ziel des Conformance Checking ist es, Gemeinsamkeiten und Diskrepanzen zwischen dem modellierten und dem beobachteten Verhalten zu finden. Dies geschieht indem das Event Log mit dem Prozessmodell verglichen wird. In der Literatur werden unterschiedliche Methoden für das Conformance Checking beschrieben: Fitness, Precision (Behavioral Appropriateness) und Structure (Structural Appropriateness). In dieser Ausarbeitung wird nur auf die Fitness eingegangen, da mit dieser Qualitätskennzahl die genaueste Aussage zur Konformität gemacht werden kann [Aal11]. Für die Berechnung der Fitness gibt es die Verfahren *Comparing Footprint* und *Token Replay*, welche in den Kapiteln 2.4.1 und 2.4.2 beschrieben werden.

Die Informationen, die durch das Conformance Checking ermittelt werden, können zusätzlich dazu benutzt werden, um ein bestehendes Prozessmodell anzupassen, damit dieses das ermittelte Verhalten der Realität besser als vorher darstellt. Dieses Vorgehen wird als Model Enhancement bezeichnet und wird im Kapitel 2.5 beschrieben.

2.4.1 Comparing Footprint

Beim Comparing Footprint Verfahren wird von einem Event Log und dem zu vergleichenden Prozessmodell ein so genannter Footprint erstellt. Dazu werden alle vorhandenen Events von einem Event Log in einer Tabelle dargestellt. Tabelle 2.4 zeigt den Footprint des Petrinetz aus der Abbildung 2.1, auf das sich die nachfolgende Beschreibung für die Erstellung eines Footprint bezieht. Zunächst werden sämtliche Events in die Spalten und Zeilenüberschriften eingefügt, anschließend werden sämtliche Beziehungen zwischen den Events in die einzelnen Zellen eingetragen. Hierzu werden die vier Symbole #, $\leftarrow$, $\rightarrow$ und $\|$ verwendet [Aal11]. Das #-Symbol bedeutet, dass keine Beziehung zwischen den Events aus der Zeile und der Spalte besteht. Weist das Event aus der Zeile eine Verbindung zum Event aus der Spalte auf, so wird dies durch das Symbol $\rightarrow$ verdeutlicht. Der $\leftarrow$ stellt den umgekehrten Verlauf dar. Das $\|$-Symbol gibt an, dass die beiden Events aus der Zeile und der Spalte eine parallele Verbindung zu einander besitzen. Nachdem für das Event Log und das Petrinetz jeweils ein Footprint erstellt wurde, müssen die beiden Footprints miteinander verglichen werden. Die Unterschiede der beiden Footprints werden ebenfalls in einer Footprint-Tabelle dargestellt. Dabei werden die beiden unterschiedlichen Verbindungen mit einem Doppelpunkt voneinander in die Zelle eingetragen (Bsp.: #:$\rightarrow$). Um nun die Fitness zu berechnen wird die Formel 2.1 verwendet. In dieser wird die Anzahl der Unterschiede und die Anzahl der möglichen Beziehungen zwischen den einzelnen Events eingetragen. Das Ergebnis ist eine Zahl zwischen 0 und 1 die aussagt, wie viele Beziehungen der Events zwischen den beiden Footprints konform sind.

$$fitness = 1 - \frac{\text{Anzahl der Unterschiede}}{\text{Anzahl der Möglichkeiten}} \tag{2.1}$$

Equation 2.1: Formel zur Fitness-Berechnung bei Comparing Footprint [Aal11]

	Aufnahme	**Anamnese**	**Blutentnahme**	**Röntgen**	**MRT**	**Befundung**	**Operation**	**Verband anlegen**	**Verband entfernen**	**Entlassen**
Aufnahme	#	$\rightarrow$	#	#	#	#	#	#	#	#
Anamnese	$\leftarrow$	#	$\rightarrow$	$\rightarrow$	$\rightarrow$	#	#	#	#	#
Blutentnahme	#	$\leftarrow$	#	$\|$	$\|$	#	$\rightarrow$	#	#	#
Röntgen	#	$\leftarrow$	$\|$	#	$\|$	$\rightarrow$	#	#	#	#
MRT	#	$\leftarrow$	$\|$	$\|$	#	$\rightarrow$	#	#	#	#
Befundung	#	#	$\leftarrow$	$\leftarrow$	$\leftarrow$	#	$\rightarrow$	#	#	#
Operation	#	#	#	#	#	$\leftarrow$	#	$\rightarrow$	#	#
Verband anlegen	#	#	#	#	#	#	$\leftarrow$	#	$\|$	#
Verband entfernen	#	#	#	#	#	#	#	$\|$	#	$\rightarrow$
Entlassen	#	#	#	#	#	#	#	#	$\leftarrow$	#

Tabelle 2.4: Footprint von dem Petrinetz aus Abbildung 2.1

2.4.2 Token Replay

Das Token Replay Verfahren überprüft für alle Cases eines Event Logs, ob sie vom Prozessmodell dargestellt werden. Das Prozessmodell muss hierfür ein Petrinetz bzw. Workflow-Netz sein. Zu Beginn liegt nur auf der ersten Stelle, dem *Start-Place*, eine Markierung (*Token*). Nun wird für alle Events im betrachteten Case der Reihe nach die zugehörige Transition gesucht und versucht, diese zu feuern (zur Funktion von Petrinetzen siehe Abschnitt 2.2). Nachdem alle Events probiert wurden sollte nun genau

eine Markierung auf der letzten Stelle (*End-Place*) liegen. Andernfalls ist der Case fehlgeschlagen. Die Markierung wird nun wieder auf den *Start-Place* verschoben und der nächste Case abgespielt.

Es gibt zwei Möglichkeiten, weshalb das Abspielen eines Case scheitert: Entweder ist eine Transition im Netz gar nicht vorhanden, oder sie ist nicht aktiviert, sprich auf der Stelle vor ihr befindet sich keine Markierung.

Für die Berechnung der Fitness werden sämtliche generierte (p), konsumierte (c), vermisste (m) und wiedergefundene (r) Markierungen in die Formel 2.2 eingetragen. Das Ergebnis gibt den Anteil der abspielbaren Cases an, eine Fitness von 1 bedeutet also, dass 100% der Cases abspielbar waren, eine Fitness von 0 bedeutet genau das Gegenteil. Die Informationen der vermissten und wiedergefundenen Token helfen bei der Suche nach fehlerhaften Events bzw. falsch modellierten Transitionen im Prozessmodell [Aal11].

$$fitness = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right) \tag{2.2}$$

Equation 2.2: Formel zur Fitness-Berechnung bei Token Replay [Aal11]

2.5 Model Enhancement

Ein weiterer Bereich des Process Mining behandelt die Erweiterung und Verbesserung von bereits existierenden Prozessmodellen und wird Model Enhancement genannt. Während das Conformance Checking die Übereinstimmung zwischen einem Prozessmodell und einem Event Log misst, erzeugt das Model Enhancement aus diesen beiden ein neues Modell. Dies kann zum Beispiel zur Reperatur eines Modells oder als Anpassung nach einem Concept Drift, also der Veränderung der äußeren Gegebenheiten der Prozessausführung, geschehen.

Beim Model Enhancement kann es sich aber auch um die Erweiterung von Prozessmodellen handeln, zum Beispiel wenn dem Modell eine neue Perspektive hinzugefügt wird. So kann durch das Übertragen der Zeitstempel aus dem Event Log auf das Prozessmodell gezeigt werden, an welchen Stellen Flaschenhälse auftreten, sprich auf die Ausführung einzelner Aktivitäten gewartet werden muss [Aal11]. Eine weitere mögliche Perspektive sind die Häufigkeiten mit der die einzelnen Events auftreten [Aal11].

2.6 Prozessmodellvergleich

Bei dem Vergleich von Prozessmodellen soll die Differenzberechnung dem Benutzer aufzeigen an welchen Stellen sich zwei Prozessmodelle voneinander unterscheiden. Das kann vor allem bei großen Prozessmodellen mit wenigen Abweichungen ein Vorteil sein.

Grundsätzlich unterscheiden sich die Algorithmen zur Differenzberechnung in zustands- und änderungsbasierte Algorithmen. Zustandsbasierte Algorithmen vergleichen die vom Mining-Algorithmus resultierenden Prozessmodelle direkt miteinander, wohingegen änderungsbasierte Algorithmen sich für automatisch generierte Prozessmodelle eignen, bei denen sich das Prozessmodell versionsabhängig verändert und ein Change-Log zum Prozessmodell verfügbar ist.

Der änderungsbasierte Algorithmus verwendet zur Berechnung den Change-Log der jeweiligen Version des Prozessmodells [KHL+10]. Zu den zustandsbasierten Algorithmen gibt es drei Ansätze, die folgend erklärt werden:

Snapshot-Algorithmus

Der Snapshot-Algorithmus benötigt zur Berechnung jeweils eine Liste aller Knoten der zu vergleichenden Prozessmodelle als Schlüssel-Wert-Paar. Eines der Prozessmodelle ist das Referenzmodell anhand dessen die Schnittmenge der Listen verglichen wird. Bestehen Knoten im Referenzmodell, die nicht in den anderen Prozessmodellen enthalten sind, markiert der Algorithmus diese als *deleted*. Knoten die nicht im Referenzmodell, jedoch in den anderen Prozessmodellen enthalten sind, werden als *added* markiert. Knoten, bei denen Vorgänger und Nachfolger den gleichen Schlüssel aber unterschiedliche Werte haben, werden als *changed* markiert. Die restlichen Knoten erhalten keine Markierung [Cor13].

Provenance-Differencing-Algorithmus

Dieser Algorithmus geht davon aus, dass die Prozessverläufe aus allen Prozessmodellen von einer übergeordneten Spezifikation abgeleitet sind, die einer baumartigen Struktur ähnelt. Ein Prozessmodell stellt demnach einen individuellen Durchlauf durch den Baum dar. So lassen sich mit diesem Algorithmus Differenzen in polynomieller Zeit berechnen. Zudem führt der Algorithmus Berechnungen von Änderungen durch, indem sowohl Einfüge- als auch Löschoperationen von Teilbäumen durchgeführt werden. Weiterhin wird die Änderungsdistanz zwischen den zu vergleichenden Prozessmodellen ermittelt. Durch diese Operationen werden fehlerhafte Bäume verhindert [Cor13].

High-Level-Diff-Ansatz

Dieser Ansatz untersucht die einzelnen Prozessverläufe eines Prozessmodells auf Modellebene anhand eines semantischen Vergleichs. Es wird jedoch vorausgesetzt, dass die Modelle dann äquivalent sind, wenn die Prozessverläufe ebenfalls äquivalent sind. Insgesamt wird der Prozessmodellvergleich in drei Schritten durchgeführt:

- Es wird eine Löschoperation für die Aktivitäten durchgeführt, die im ersten aber nicht im zweiten Prozessmodell vorhanden sind [Cor13].
- Nachdem für alle in beiden Prozessmodellen vorkommenden Aktivitäten eine Menge der minimalen Bewegungsoperationen berechnet worden ist wird anschließend eine Bewegungsoperation für Elemente vollzogen bei der die Aktivitäten an ihre Position im zweiten Prozessmodell bewegt werden [Cor13].
- Es wird für alle Aktivitäten, die im zweiten Prozessmodell aber nicht im ersten Prozessmodell vorkommen, eine Einfügeoperation durchgeführt [Cor13].

Gegenüber dem Snapshot-Algorithmus erlaubt der High-Level-Diff-Ansatz nur syntaktisch und gegebenenfalls semantische, korrekte Differenzen der Aktivitäten im Prozessmodell [Cor13].

2.7 Multidimensionaler Ansatz

Process Mining Techniken offenbaren die tatsächliche Ausführung von Prozessen. Wenn diese Techniken angewandt werden ohne die zugrunde liegenden Daten zu differenzieren ist es schwierig, ein Verständnis für die Prozessabläufe zu entwickeln, weil die generierten Prozessmodelle heterogene Informationen vermischen. Gerade Gesundheitsversorgungsprozesse unterscheiden sich oft, weil sie von verschiedenen Faktoren beeinflusst werden, wie z. B. dem Alter und der Krankheit des Patienten oder der Region in der die Behandlung stattfindet. Diese Faktoren sind Eigenschaften der jeweiligen Cases und können einem Event Log entnommen werden. Wenn beispielsweise regionale Unterschiede in der Qualität von Behandlungen einer bestimmten Krankheit bestehen, dann werden diese Unterschiede in einem einzigen Prozessmodell nicht deutlich. Aus diesem Grund müsste das Event Log zunächst in die verschiedenen Patientengruppen entsprechend den Regionen aufgeteilt und das Prozessmodell für jede Region aufgezeigt werden. Anschließend könnten die konkreten Unterschiede identifiziert und analysiert werden. Die Unterschiede könnten in der unterschiedlichen Ausstattung, der abweichenden Erfahrung des Krankenhauspersonals mit der Krankheit oder der unterschiedlichen Anwendung von innovativen Therapiemethoden liegen. Ein weiterer Grund könnte die abweichende Altersverteilung in den verschiedenen Regionen sein. Dies legt den Verdacht nahe, dass der Gesundheitsversorgungsprozess hauptsächlich von dem Alter des Patienten abhängt. Daher wäre die Analyse des Einflusses des Patientenalters der nächste mögliche Schritt. Dafür muss das Event Log zusätzlich anhand des Alters der Patienten aufgeteilt werden [Vog13].

Um Gesundheitsprozesse mit Process Mining zu analysieren bedarf es somit einer differenzierten Betrachtung der Datengrundlage. Hierfür eignet sich das Konzept des multidimensionalen Datenwürfels. Das Konzept wird in den nächsten Abschnitten (2.7.1 und 2.7.2) genauer erläutert.

2.7.1 Logische Datenmodellierung

Das multidimensionale Datenmodellierungskonzept gilt als eine effektive und ausdrucksstarke Technik zur Modellierung auswertungsorientierter Datenanalysen im Entscheidungsfindungsprozess. Darüber hinaus genießt diese multidimensionale Modellierungstechnik einen anwachsenden Einsatz in Data-Warehouse-Systemen und ermöglicht eine einfachere Navigation durch die Datenbestände und deren Strukturen [Far11].

Ein Data-Warehouse (DWH) ist ein System, das Informationen aus vielen unterschiedlichen Quellen in einer für die Entscheidungsfindung optimierten Datenbank integriert. Die hier integrierten und gespeicherten Daten werden auf Basis eines multidimensionalen Datenmodells mit OLAP-Operationen (On-Line Analytical Processing) explorativ und interaktiv analysiert [VK12].

In diesem Abschnitt werden Begriffe, die essenziell für die multidimensionalen Modellierung sind näher, beschrieben und definiert. Anschließend werden die verschiedenen Operatoren zur Analyse der multidimensionalen Daten erklärt.

Dimension

Die Dimensionen repräsentieren eine spezifische Perspektive der Daten, aus deren Sicht die Fakten (variabel) analysiert werden, und dienen der orthogonalen Strukturierung des Datenraums. Die Dimensionen stellen somit betriebliche Sichten bzw. Analysewege der Fakten dar [SSG13]. Es kann

beispielsweise der Verkauf aus der Sicht von Produkten, Orten, Zeiträumen, etc. (Dimensionen) analysiert werden. Die Attribute von Dimensionen können aufgrund funktionaler Abhängigkeiten hierarchisch strukturiert werden und stellen somit die Grundlage für das Navigieren entlang der Dimensionen dar [LU07]. Hierbei wird die Hierarchie durch Stufen definiert, zwischen denen eine Eltern-Kind Beziehung in einer Dimension besteht. So können beispielsweise in der Zeitdimension die Monate zu Quartalen oder in der Altersdimension Alter zu Altersgruppen zusammengefasst werden.

Fakten

Fakten stellen den quantifizierenden Anteil des multidimensionalen Modells dar und sind Gegenstand von Analyse und Auswertung. Fakten können zum Beispiel betrieblichen Sachverhalte wie Umsatz, Verkaufszahlen oder Kosten sein. Die Fakten bilden die Inhalte der Zellen eines Würfels und haben eine informative Eigenschaft [GRC09]. Es wird zwischen direkt erfassten und Fakten, die durch Anwendung arithmetischer Operationen aus anderen Kennzahlen abgeleitet werden, unterschieden.

Datenwürfel

Ein Datenwürfel stellt eine mehrdimensionale Darstellung von Kennzahlen dar [VK12]. Ein Würfel besteht aus Datenzellen, die eine oder mehrere Kennzahlen beinhalten [AB09]. Durch die Kombination der Dimensionen, die den Würfel aufspannen (Achsen des Würfels), wird der Würfel gebildet, und er besteht in diesem Sinne aus (Daten) Zellen. Diese Zellen repräsentieren die Schnittpunkte der Dimensionen (z. B. Altersgruppe, Regionen, Zeit) und beinhalten eine oder mehrere Kenngrößen auf Detailebene (Rohdaten), beispielsweise einzelne Krankheiten einer Altersgruppe pro Region an einem bestimmten Monat. Ein Würfel ist die Ausprägung eines Würfelschemas. Des Weiteren ist es möglich Würfel zu aggregieren [Far11] und das dabei resultierende Ergebnis ist wiederum ein Würfel. Die Granularität der Daten ist die Menge der Klassifikationsstufen, die ein Aggregationsniveau eindeutig identifiziert.

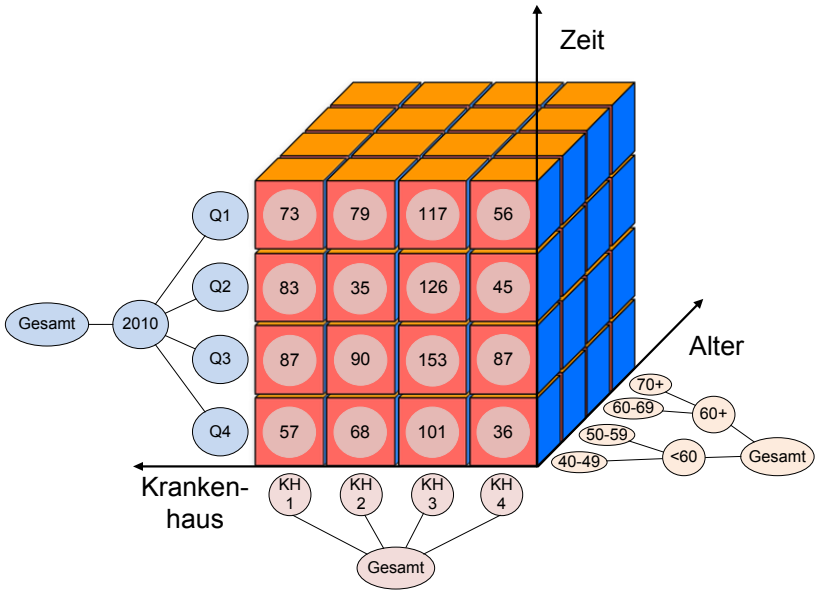


Abbildung 2.8: Ein multidimensionaler Datenwürfel, entnommen aus [Kra12].

Datenanalyse

Die Eigenschaft von OLAP ist nicht nur auf das multidimensionale Datenmodell begrenzt, die OLAP Methode gibt auch die Möglichkeit eine Datenanalyse durch bestimmte Operatoren durchzuführen [AB09]. Hierbei stellt die Datenanalyse anhand solcher OLAP-Instrumente einen dynamischen Prozess dar, bei dem der Benutzer mithilfe von multidimensionalen Operatoren durch die multidimensionale Datenstruktur navigiert. Im Folgenden werden einige der OLAP-Operatoren näher beschrieben.

Drill-Down: Mit dieser Operation werden die Daten verdichtet und aggregiert. Hierbei kann der Benutzer die Dimensionshierarchie durchlaufen, indem er von einem bestimmten Aggregationsniveau auf die nächsttiefere und damit detailliertere Hierarchiestufe steigt. Es wird in den Datenwürfel hinein gezoomt und die Granularität erhöht [GRC09].

Roll-Up: Bei dieser Operation durchläuft der Benutzer den umgekehrten Weg, indem von einer feinen Detailstufe zu immer stärkeren Aggregationen aufsteigt. Hier wird, entlang der Hierarchie, aus dem Würfel herausgezoomt und die Granularität wird gesenkt [GRC09].

Drill-Across: Durch die Anwendung des Drill-Across-Operators werden Kennzahlen eines Datenwürfels anhand anderer Dimensionen oder Klassifikationshierarchien eines anderen Datenwürfels betrachtet. Es wird also von einem Datenwürfel zu einem anderen gewechselt und Anfragen über mehr als eine Faktentabelle durchgeführt [GRC09].

Slicing und Dicing: Slicing und Dicing ermöglichen die Darstellung der Daten in bestimmten Teilsichten, den Scheiben (Slicing) oder Würfeln (Dicing). Der Slice-Operator ermöglicht die Betrachtung eines Ausschnitts des multidimensionalen Datenwürfels anhand der Klassifikationsstufen einer Dimension. Durch Bildung einer Aggregation über einen Klassifikationsknoten können einzelne Scheiben aus dem Datenwürfel herausgeschnitten und isoliert betrachtet werden. Beim Dice-Operator kann eine individuelle Sicht der Daten durch das Herausschneiden eines Teilwürfels erzeugt werden [GRC09].

Pivot (Rotate): Durch diese Operation ist es möglich den Datenwürfel um seine Achsen zu drehen. Die Drehung erfolgt durch das Vertauschen der Dimensionen Dieser OLAP-Operator ermöglicht es, die multidimensionalen Daten aus verschiedenen Perspektiven auszuwerten, zu analysieren und verschafft dem Benutzer gleichzeitig mehr Flexibilität bei der Analyse [AB09, GRC09].

2.7.2 Umsetzung der Multidimensionalen Datenmodellierung

Zur Umsetzung eines multidimensionalen Datenmodells müssen die multidimensionalen Daten in einer bestimmten Struktur in das Datenbankmanagementsystem gespeichert werden. Hierbei besteht die Möglichkeit diese Daten entweder in relationale Strukturen (ROLAP) oder direkt in multidimensionale Strukturen und Arrays (MOLAP) umzusetzen.

ROLAP

Der ROLAP-Ansatz stellt eine OLAP-Funktionalität mit relationalen Datenbanken zur Speicherung und Analyse von multidimensionalen Daten dar. Das Konzept baut auf bestehenden relationalen Technologien auf und basiert auf der direkten Umsetzung von Star- bzw. Snowflake-Schema auf Relationen eines RDBMS. Hierbei werden die Fakten und Dimensionen in einer Faktentabelle abgebildet. Aufgrund der vielen Vorteile wie z. B. Massendatenhaltung, Skalierbarkeit, Sicherheit, etc. von RDBMS ist der ROLAP der meist eingesetzte Implementierungsansatz in den Data Warehouse Systemen [Far11].

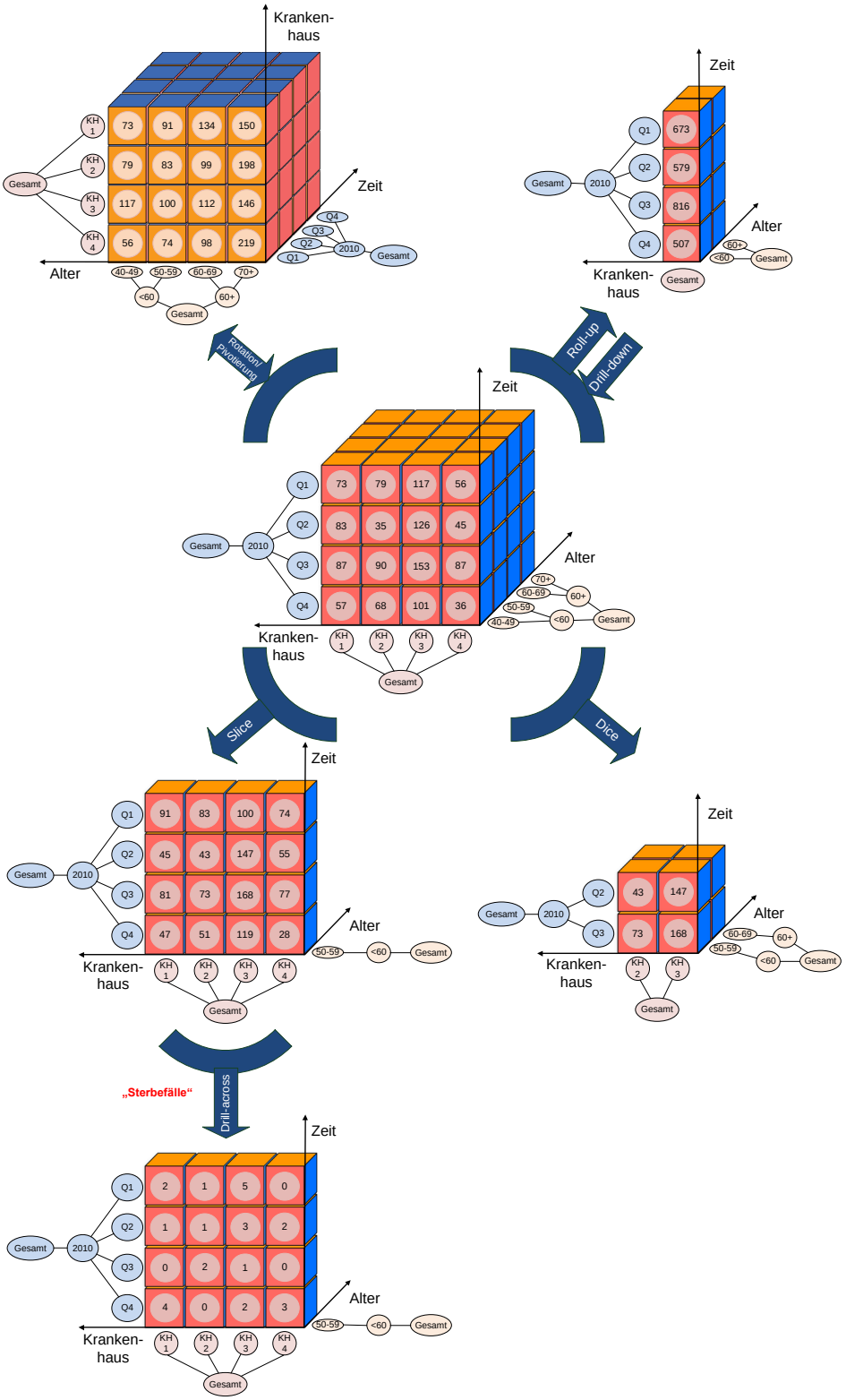


Abbildung 2.9: OLAP-Operationen am Beispiel eines dreidimensionalen Würfels, entnommen aus [Kra12]

Star-Schema: Das Star-Schema ist ein Datenmodellierungsverfahren zur Abbildung multidimensionaler Daten in einem RDBMS [KH06]. Im Star-Schema wird jede Dimension durch eine Tabelle repräsentiert, deren Spalten aus den Primärschlüsseln und weiteren Attributen der zugrundeliegenden Entitäten gebildet werden. Die Faktentabelle enthält die Kennzahlen und steht in der Mitte des Star-Schemas, umgeben von den nicht normalisierten Dimensionstabellen. Der Primärschlüssel der Faktentabelle setzt sich aus den Primärschlüssel der Dimensionstabellen zusammen. Mit Hilfe der Fremdschlüsselbeziehungen wird sichergestellt, dass der Schlüssel der Faktentabelle nur aus existierenden Primärschlüsseln der Dimensionen bestehen kann. Einer der Hauptvorteile dieser Modellierung ist die einfache Struktur der Tabellen und die flexible Darstellung von Klassifikationshierarchien als Spalten in Dimensionstabellen.

Snowflake-Schema: Dies ist eine Variation des Star-Schemas mit einer Faktentabelle in der Mitte, umgeben von normalisierten Dimensionstabellen. Bei dieser Modellierung bleibt die Faktentabelle unverändert, während die Dimensionstabellen durch Aufgliederung ihrer Hierarchieebenen in Normalform gebracht werden [GRC09]. Hierbei wird für jede Ebene in einer hierarchischen Dimension eine Tabelle angelegt, die mit referenzierenden Schlüsseln verknüpft wird. Als Vorteil resultiert eine besser strukturierte Datenmenge mit weniger Redundanz. Dem entgegen steht jedoch der Nachteil, dass durch erhöhte Komplexität zeitintensive und komplizierte Join-Abfragen notwendig sind, die wiederum zu längeren Antwortzeiten führen. In der vorliegenden Projektarbeit wurde aufgrund der Komplexität der zu untersuchenden Daten eine Mischung aus den beiden Ansätzen gewählt. Im Abschnitt 5.2 wird dieser Implementierungsansatz wie er in dieser Arbeit umgesetzt wurde ausführlich beschrieben.

MOLAP

Beim MOLAP-Konzept wird für die Implementierung des Datenmodells in einem MDDBMS eine multidimensionale Datenstruktur (Arrays) eingesetzt, die sich besonders durch eine hohe Verarbeitungsgeschwindigkeit und Funktionalität zur Analyse und Auswertung von Daten charakterisiert. Der Datenwürfel wird auf physischer Ebene als multidimensionale Matrix im Hauptspeicher gespeichert (Hypercube-Ansatz), was dazu führt, dass MOLAP-Systeme relativ gute Antwortzeiten und im Ergebnis eine große Flexibilität und Schnelligkeit aufweisen [GRC09].

Eine weitere Möglichkeit der Speicherung von multidimensionalen Daten ist das hybride OLAP (HOLAP). Der HOLAP-Ansatz versucht bei der Umsetzung die Vorteile von MOLAP (hohe Performance) und ROLAP (hohe Skalierbarkeit) zu kombinieren [Mad00]. Hierbei werden häufig angefragte Daten mit kleinem bis mittlerem Volumen, in der Regel, in MDDBMS gespeichert, die umfangreichen und selten benötigten Daten hingegen im RDBMS.

2.8 Consolidation

Das Ziel der Consolidation ist, die Komplexität der Mining-Ergebnisse zu reduzieren [Vog13]. Bei dem in dieser Arbeit verwendeten mehrdimensionalen Ansatz kann es unter Umständen vorkommen, dass es sehr viele Prozessmodelle zu analysieren gibt. Daher ist es eine Erleichterung wenn die Ergebnisse vor der Betrachtung auf interessante bzw. wichtige Prozessmodelle reduziert werden. Alternativ können ähnliche Modelle durch ein Clustering zusammengefasst werden oder nur für eine Gruppe

repräsentative Modelle angezeigt werden [Vog13]. Dieses Vorgehen erleichtert es dem Betrachter, schneller Wissen aus den generierten Modellen zu erlangen.

3 Anforderungen

In diesem Kapitel werden die Anforderungen an das Projekt und die zu entwickelnde Software erhoben. Die Anforderungen werden in funktionale und nicht-funktionale Anforderungen unterteilt. Des weiteren wird eine Systemabgrenzung vorgenommen und der Einsatzzweck der Software durch einen Anwendungsfall verdeutlicht.

3.1 Funktionale Anforderungen

Die funktionalen Anforderungen wurden zu Beginn des Projektes von der Projektgruppe unter Rücksprache mit den Betreuern erhoben. Zudem wurden diese Anforderungen in Muss- und Soll-Kriterien unterteilt. Ziel innerhalb der ersten Projekthälfte war es, einen vertikalen Prototypen mit den unten genannten Anforderungen zu verwirklichen. Im folgenden werden die verschiedenen funktionalen Anforderungen vorgestellt:

F1	Es muss ein DWH für multidimensionale Eventdaten aufgebaut werden.
F2	Die Event Logs müssen in eine multidimensionale Datenstruktur integriert werden.
F3	Gängige Datenbankanbindungen müssen unterstützt werden.
F4	Die Software muss eine explorative Navigation auf den Eventdaten durch OLAP-Operationen ermöglichen.
F5	Die Software muss die Abbildung von Attributen in Dimensionshierarchien unterstützen.
F6	Die Zellen des Datenwürfels müssen die Menge von Traces enthalten.
F7	Das Programm muss die Aggregation von Zellen ermöglichen.
F8	Die Software muss den Einsatz von Process Mining-Algorithmen auf den vorher selektierten Daten ermöglichen, hierzu muss ein ausgewählter Algorithmus beispielhaft implementiert werden.
F9	Die generierten Prozessmodelle müssen in einer geeigneten Form visualisiert werden, die Implementierung soll sich auf eine einfache Visualisierung beschränken.
F10	Die Software soll über eine Schnittstelle verfügen um Consolidation-Algorithmen einzubinden, eine Implementierung von konkreten Algorithmen ist nicht vorgesehen.

Zu beginn der zweiten Projekthälfte wurden die bereits erhobenen Anforderungen um zusätzliche Anforderungen erweitert. Der vertikale Prototyp sollte weitere Funktionalitäten aufweisen und bereits vorhandene Funktionen optimiert werden. Diese Anforderungen wurden in die folgenden Muss-, Soll- und Kann-Kriterien unterteilt:

F11	Die Software muss eine Microsoft SQL Datenbankanbindung unterstützen.
F12	Die Erweiterbarkeit des Systems um weitere Datenbankanbindungen muss durch eine Schnittstelle gewährleistet werden.
F13	Das Programm muss die Möglichkeit bieten die Metadaten des DWH extern in eine XML-Datei zu speichern und diese zu serialisieren.

F14	Die Anwendung muss Informationen bezüglich der Anzahl der vorhandenen Events und Cases innerhalb der ausgewählten Daten bereitstellen.
F15	Der während der ersten Projekthälfte implementierte Mining-Algorithmus Heuristic Miner muss optimiert werden.
F16	Der Mining-Algorithmus Alpha Miner muss implementiert werden.
F17	Der Mining-Algorithmus Inductive Miner muss implementiert werden.
F18	Die Software muss Informationen bezüglich der genutzten Events oder auch der Fitness des Prozessmodells bereitstellen
F19	Grundlegende Funktionen eines Conformance Checkings müssen implementiert werden.
F20	Die Software soll die Möglichkeit bieten die Dimensionen innerhalb der Events weiter zu aggregieren.
F21	Die Software soll eine Schnittstelle besitzen, um weitere Mining-Algorithmen einbinden zu können.
F22	Das Programm soll über rudimentäre Funktionen einer Consolidation verfügen.
F23	Die Anwendung soll über eine Funktion für einen Prozessmodellvergleich verfügen.
F24	Der implementierte Visualisierungs-Algorithmus soll optimiert werden.
F25	Um die Qualität des Programmcodes zu erhöhen sollen Unit-Tests erstellt werden.
F26	Die Software kann, bei der Visualisierung der Prozessmodelle, zusätzliche Informationen wie zum Beispiel die durchschnittliche Behandlungsdauer bereitstellen.
F27	Die Anwendung kann verschiedene Eventtypen innerhalb der Visualisierung durch eine farbliche Hervorhebung kenntlich machen.
F28	Das Programm kann über grundlegende Funktionen eines Decison Mining verfügen, welche Aussagen bezüglich der getroffenen Entscheidungen innerhalb des Prozessmodells ermöglicht.
F29	Eine Implementierung eines weiteren Visualisierungs-Algorithmus, in Form eines Kausal- oder BPMN-Netzes, kann umgesetzt werden.
F30	Die Software kann über eine Schnittstelle für EventLog-Operations verfügen.
F31	Die Anwendung kann eine Schnittstelle für ProcessModell-Operations beinhalten.

3.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen stellen die Qualitätsanforderungen an eine Software dar. Sie zeigen auf, welche Eigenschaften das zu entwickelnde Softwaresystem zusätzlich zur Funktionalität aufweisen soll. Im Allgemeinen beziehen sich die nicht-funktionalen Anforderungen auf ein gesamtes System und wirken einschränkend auf das System und deren Funktionen [Amr12]. Als Grundlage für das Softwareprojekt dienen die Qualitätsanforderungen nach DIN ISO 9126. Diese Anforderungen bilden eine Grundlage für das Formulieren nicht-funktionaler Anforderungen [Tie09].

N1 Die gesamte Funktionalität des Softwareprojekts ist gemäß den Anforderungen angemessen zu implementieren. Hierdurch soll unter anderem die Richtigkeit der Funktionalität sichergestellt werden. Bei dem Aufruf bestimmter Funktionen in der Software, wie z. B. das Anwenden von OLAP-Operatoren, soll gewährleistet werden, dass daraus die korrekten SQL-Statements erzeugt werden. Die Sicherstellung der Funktionalität soll durch Komponententests erfolgen. Hierbei soll eine Testabdeckung von 60% angestrebt werden.

N2 Eine weitere Anforderung stellt der Datenschutz dar. Da die Software mit sensiblen Daten arbeitet, soll sichergestellt werden, dass der Zugriff nur von autorisierten Personen erfolgt. So sollen beispielsweise die Zugangsdaten der verwendeten Datenbanken ausschließlich verschlüsselt abgelegt werden.

N3 Die Bedienung der Software soll intuitiv und leicht erlernbar sein, sodass sich die Benutzer in kürzester Zeit zurechtfinden. Jegliche Begrifflichkeiten, Beschreibungen und Meldungen im Programm sollen in englischer Sprache verfasst werden. Der Ablauf der einzelnen Programmfunktionen soll klar gegliedert sein und dem Ablauf des Process Mining entsprechen. Des Weiteren ist zu beachten, dass ein bestimmter Schritt des Process Mining Workflow nur dann vollzogen werden kann, sobald alle dazugehörigen Schritte abgeschlossen sind. Ist dieses nicht der Fall, muss der Benutzer darüber informiert werden. Da gewisse Erfahrungen im Bereich des Process Mining bei künftigen Benutzern vorausgesetzt werden können, ist die Verwendung gängiger Begrifflichkeiten des Process Mining im Programm sinnvoll und sollte dementsprechend verwendet werden. Außerdem sollen jegliche Fehlzustände des Programms den Benutzern so detailliert wie möglich mitgeteilt werden, sodass diese entsprechend reagieren können.

N4 Die Software soll mit Visual Studio 2012 / 2013 (.NET Framework 4.5) in der Programmiersprache C# entwickelt werden. Des Weiteren soll die Programmierung mit dem Einsatz des Grafik-Frameworks WPF erfolgen. Hierdurch soll eine strikte Trennung der Präsentations- und Geschäftslogik erreicht werden.

N5 Der Quellcode soll den gängigen Codekonventionen für C# (.NET Naming Guidelines) entsprechen. Des Weiteren sollen jegliche Variablen, Funktionsnamen und Kommentare des Quellcodes in englischer Sprache verfasst werden. Darüber hinaus soll jede Funktion eine Beschreibung ihres Ablaufs enthalten sowie einen Autor, um eine direkte Zuordnung der Funktion sicherzustellen. Wenn Teile des Quellcodes nicht selbsterklärend sind, sollen diese entsprechend kommentiert werden. Das Programm soll so entwickelt werden, dass Änderungen oder Erweiterungen ohne großen Aufwand vorgenommen werden können. Dies beinhaltet unter anderem die Verwendung von Schnittstellen, wodurch eine einfache Erweiterbarkeit gewährleistet werden soll.

N6 Das Programm soll unter dem Betriebssystem Windows lauffähig sein. Des Weiteren soll der Aufwand der Installation der Software in einem angemessenen Rahmen bewegen.

3.3 Systemabgrenzung

Nachdem in den vorherigen Abschnitten die funktionalen und nicht-funktionalen Anforderungen erläutert wurden, erfolgt in diesem Abschnitt eine Systemabgrenzung.

Es sollen Aussagen über die zu entwickelten Komponenten getroffen werden.

Als Datenbasis des Systems dient ein DWH für multidimensionale Eventdaten. Dieses ist aufzubauen. Die Entwicklung eines eigenen Datenbankmanagementsystem (DMBS) ist nicht Teil des Projekts. Vielmehr soll dort auf bestehende Standards, wie zum Beispiel MySQL oder Oracle, zurückgegriffen werden. Somit ist die eigentliche Datenbank nicht Teil des Systems. Hingegen müssen die Mining-Algorithmen selbstständig implementiert werden. Die theoretische Grundlagen bilden hierzu die jeweiligen wissenschaftlichen Veröffentlichungen. Diese müssen analysiert werden und ggf. für den Anwendungsfall angepasst werden. Darüber hinaus sind weitere Kernkomponenten, wie die Consolidation, der Prozessmodellvergleich oder auch das Conformance Checking Teil des Systems. Auch diese müssen auf Basis von wissenschaftlichen Veröffentlichen selbstständig implementiert werden. Bei der Entwicklung des Systems ist zudem stets auf eine mögliche Erweiterbarkeit der Komponenten durch Schnittstellen zu achten.

3.4 Anwendungsfälle

In diesem Abschnitt wird der erwartete Weg erklärt, den der Benutzer durch die Software navigiert. Abbildung 3.1 zeigt diesen Prozessablauf in vereinfachter Form.

Nach dem Start der Software kann der Benutzer eine Datenbankverbindung herstellen oder ein zuvor gespeichertes Ergebnis laden.

Ist die Datenbankverbindung noch nicht konfiguriert, muss der Benutzer die Datenbankverbindungskonfiguration aufrufen, die Verbindungsdaten eingeben, hinzufügen und sich mit dieser verbinden. Es können mehrere Datenbankverbindungen hinterlegt sein, von denen jeweils eine ausgewählt werden kann. Kann die Verbindung nicht hergestellt werden, muss der Benutzer die Verbindungsdaten korrigieren und sich erneut verbinden.

Anschließend hat er die Möglichkeit die Metadaten des DWH‘s anzupassen, damit die im weiteren Verlauf benutzten Tabellennamen lesbarer sind. Hier kann er den Classifier der Eventtabelle auswählen sowie die Bezeichnungen der verschiedenen Dimensionen und Level an seine Bedürfnisse anpassen.

Im nächsten Schritt erfolgt die Auswahl der Daten. Hierzu wählt der Benutzer die verschiedenen Dimensionen und anschließend ihre Granularität aus. Dieses ist sowohl für die Cases als auch für die Events möglich.

Nach der Auswahl der Daten kann der Benutzer einen Mining-Algorithmus auswählen und ggf. konfigurieren. Es folgt der Start des Mining-Algorithmus. Eine Fortschrittsanzeige gibt Aufschluss darüber, wie der Stand des Mining-Prozess ist.

Nach dem Mining werden die Ergebnisse in einer Ergebnis-Matrix angezeigt. Hier besteht die Möglichkeit, die einzelnen Prozessmodelle zu betrachten und zu interpretieren. Eine Konsolidierung erleichtert das Auffinden von relevanten Modellen. Ebenso können zwei Prozessmodelle miteinander verglichen werden um Unterschiede darzustellen.

Die Durchführung des Conformance Checking ist ebenfalls möglich. Darüber hinaus lassen sich hier die Prozessmodelle drucken, exportieren oder das Ergebnis für eine spätere Verwendung speichern.

Der Benutzer hat an jeder Stelle die Möglichkeit einen Schritt zurückzuspringen, um bspw. die Auswahl oder Konfiguration der Mining-Algorithmen zu ändern, andere Daten oder eine andere Datenbank auszuwählen.

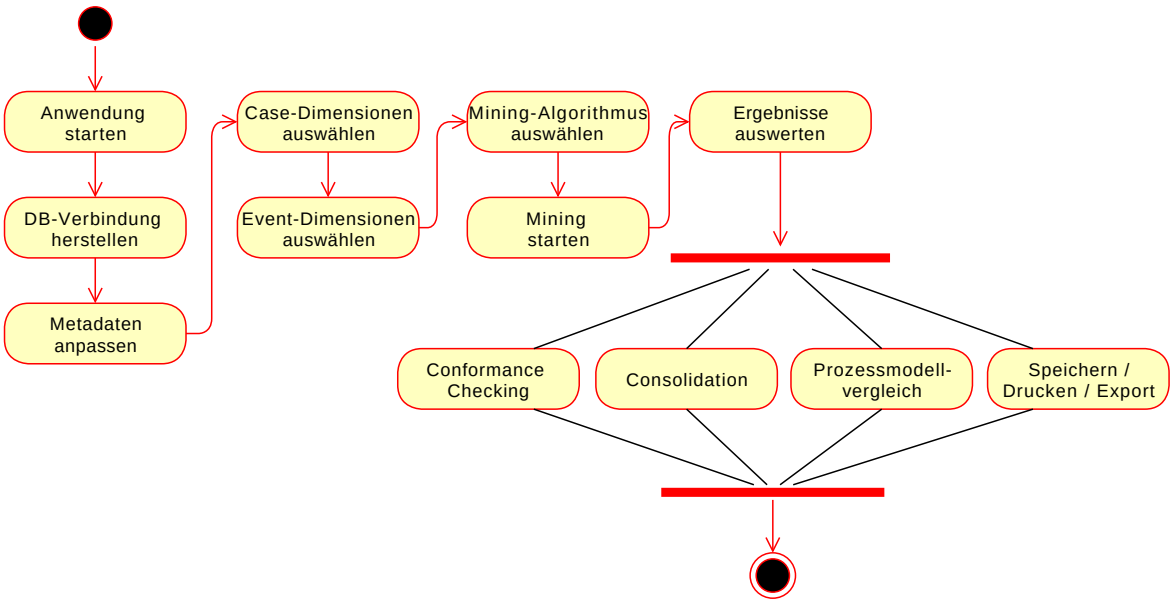


Abbildung 3.1: Prozessablauf

4 Softwarearchitektur

Die Architektur des Programms verfolgt vor allem zwei Ziele: Eine einfache Erweiterbarkeit und eine Ausrichtung am Model-View-Controller Architekturmuster. Aus diesen Gründen ist die Softwarelösung (in Visual Studio “Solution” genannt) in mehrere Teilprojekte unterteilt, die jeweils ein Konzept des Process Mining umsetzen (siehe Abschnitt 2), oder ein in sich abgeschlossener Teilbereich des Programms sind.

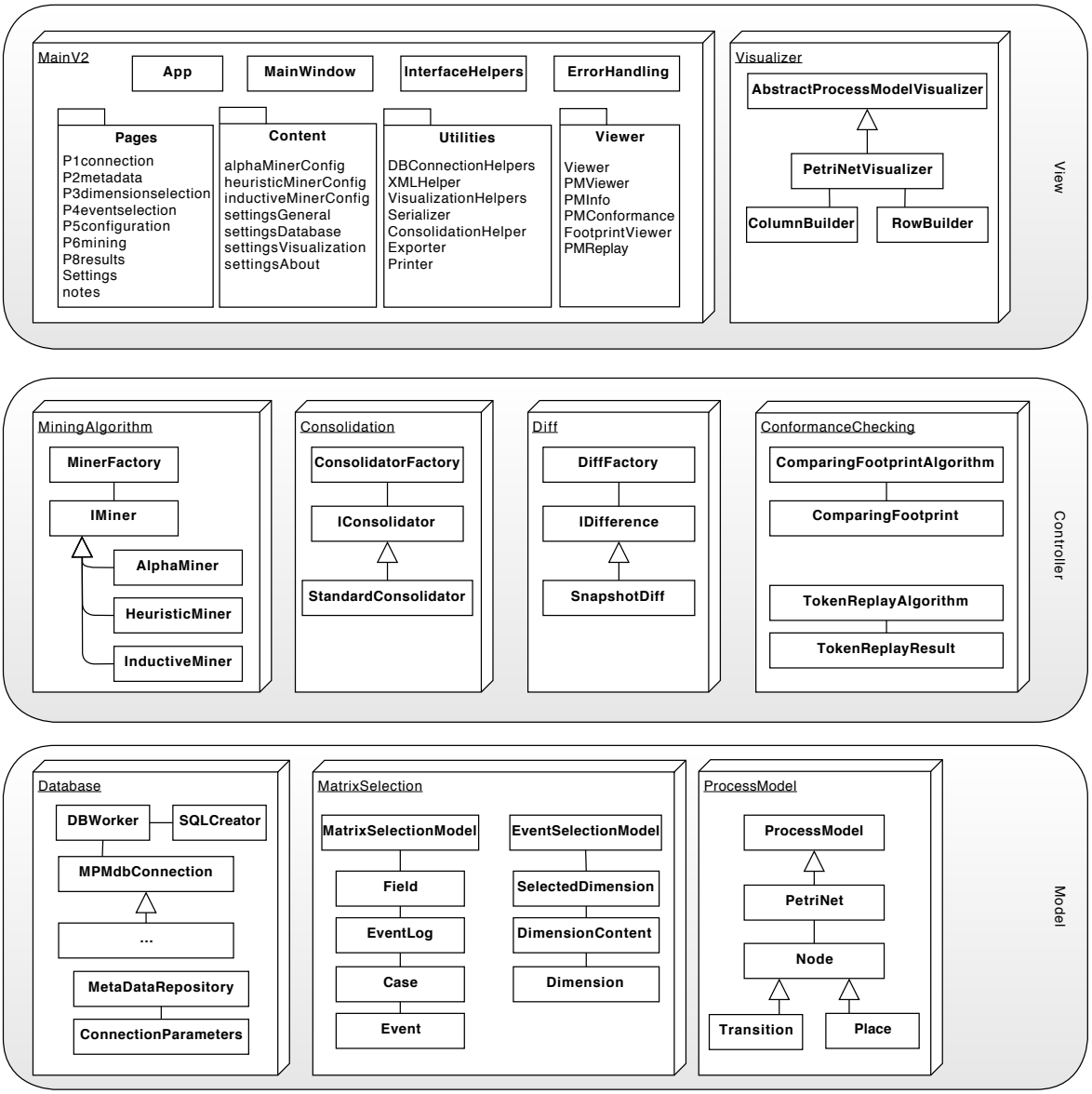


Abbildung 4.1: Die Softwarearchitektur des Process Cube Explorers

Die konzeptuelle Trennung ist in Abbildung 4.1 zu erkennen: Die Pakete *Model* (für *Process Model*) und *MatrixSelection* stellen die Datenstrukturen bereit, die im gesamten Programm verwendet werden. Das Paket *Database* vereint alle Model- und Controller-Klassen die zum Laden der Daten

und Erstellen der internen Datenstrukturen benötigt werden. Die wichtigsten Controller-Pakete sind *MiningAlgorithm*, *Consolidation*, *Diff* und *ConformanceChecking*. Sie arbeiten alle ausschließlich auf den Datenstrukturen der Model-Pakete. Die Benutzeroberfläche befindet sich im Hauptpaket *MainV2*, über das alle Controllerklassen aufgerufen werden. Die View-Schicht beinhaltet daneben das *Visualization*-Paket zur Darstellung von Prozessmodellen.

Im Folgenden werden alle Pakete mit ihren wichtigsten Eigenschaften genauer betrachtet.

4.1 Die Model-Pakete

Die wichtigste Aufgabe des Models ist es, die aus der Datenbank geladenen Daten übersichtlich und strukturiert zu speichern. Gleichzeitig müssen aus der Auswahl, die der Benutzer trifft, später SQL-Statements generiert werden. Deshalb werden sämtliche Metadaten gespeichert und mit den eigentlichen Daten verknüpft. Dies geschieht im *MatrixSelection*-Paket (siehe Abbildung 4.2).

Die *MatrixSelectionModel*-Klasse speichert eine Liste der *Fields* der Matrix (*MatrixFields*) und eine Liste der Dimensionen mit allem, was der Benutzer ausgewählt hat (*SelectedDimensions*). Ein *SelectedDimension*-Objekt beinhaltet eine Dimension, welche wiederum Unter-Dimensionen verwaltet (*DimensionLevelList*) und sämtliche Metadaten wie Tabellennamen, Fremdschlüsselbeziehungen und Spaltennamen (in *DimensionColumnNames*-Objekten) speichert. Außerdem werden hier die *DimensionContent*-Daten, sprich die konkreten Ausprägungen der Dimensionen, gespeichert. Diese tauchen ebenfalls im *SelectedDimension*-Objekt auf, dort jedoch nur jene, die der Benutzer ausgewählt hat.

Wenn der Benutzer die Dimensionen ausgewählt hat, werden die *Field*-Objekte erstellt. Sie entsprechen der Matrix die auch in der GUI sichtbar ist. Sie enthalten je ein *DimensionContent*-Objekt das ihre Ausprägung der ersten beiden Dimensionen angibt. Darüber hinaus können Filter ausgewählt werden, die in der *AdditionalFiltersList* als weitere *SelectedDimension*-Objekte gespeichert werden.

Field-Objekte beinhalten aber vor allem den generierten SQL-Code, später das fertige *Processmodel* (siehe Abbildung 4.3) und das *Event Log*. Ein *Event Log*-Objekt besteht primär aus einer Liste von *Case*-Objekten, verfügt aber noch über einige weitere, hilfreiche Methoden und Properties. Ein *Case*-Objekt wiederum speichert neben einer Liste von *Event*-Objekten beliebige weitere Daten (z. B. den Patientennamen). *Event*-Objekte besitzen eine *Name*-Property, in der der Name der Aktivität gespeichert wird. Das ist die Tabellenspalte, die vom Benutzer als *Classifier* ausgewählt wird. Alle weiteren Informationen werden in einem zusätzlichen Dictionary gespeichert.

Das Paket *Model* (siehe Abbildung 4.3) enthält Klassen, um Prozessmodelltypen darzustellen. Sämtliche Modelltypen leiten von der abstrakten Klasse *Procesmodel* ab, damit im restlichen Code nur auf diese verwiesen werden braucht. Aktuell gibt es eine Implementierung für Petrinetze, weil die drei umgesetzten Mining-Algorithmen Petrinetze erzeugen. Als weitere Typen sind z. B. Causalnets oder BPMN denkbar.

Die *PetriNet*-Klasse besteht hauptsächlich aus einer Liste von Transitionen und einer Liste von Stellen (“Places”). Beide Typen leiten von der abstrakten Klasse *Node* ab, um gemeinsame Aktionen (z. B. das Löschen einer Node) zu ermöglichen. Eigene Methoden erlauben es, dem Netz Transitionen und Places so hinzuzufügen, dass sie automatisch richtig miteinander verknüpft sind (*AddTranistion()* und *AddPlace()*). Dafür stellen die *Transition*- und die *Place*-Klasse ebenfalls eigene Methoden

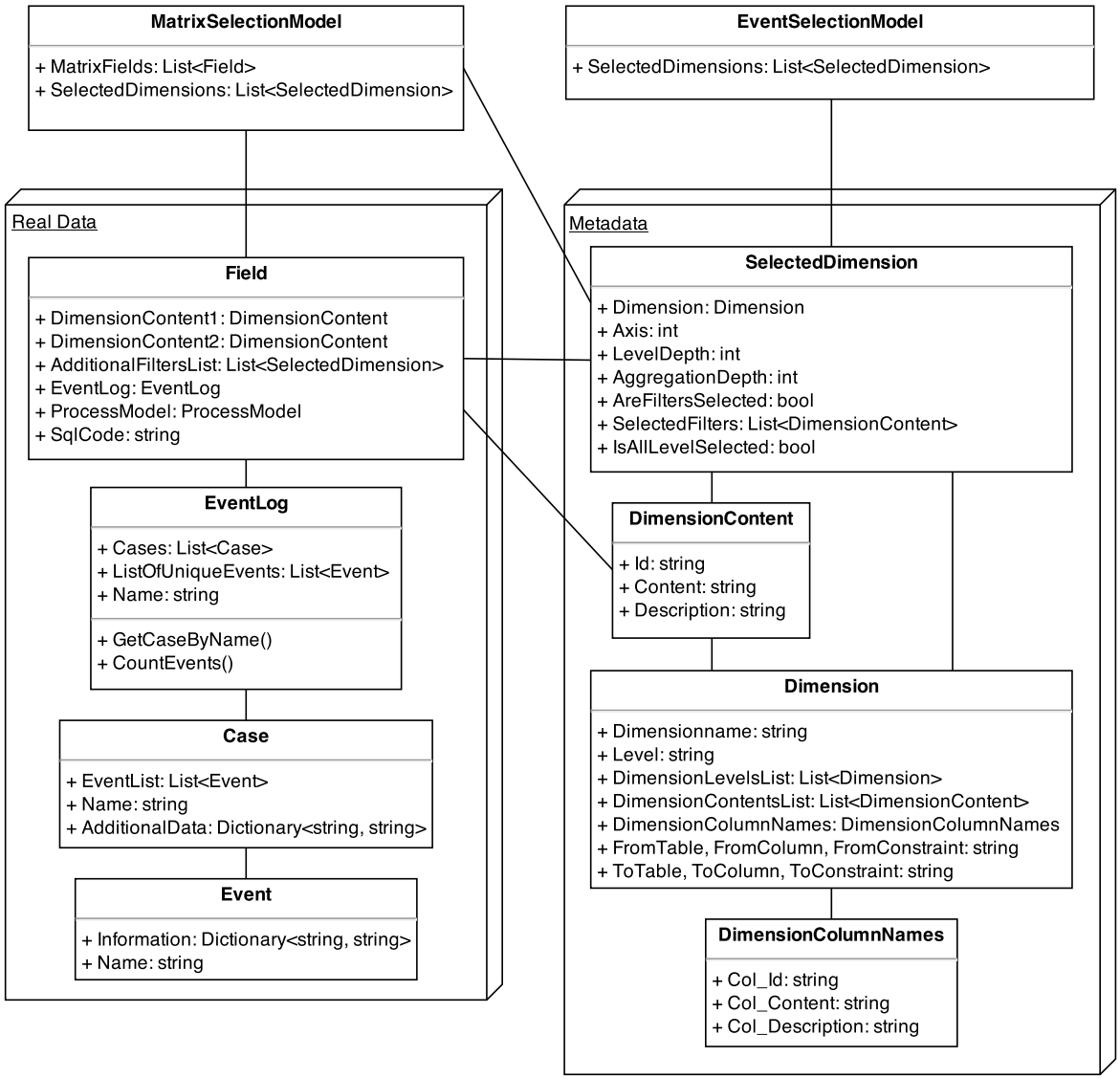


Abbildung 4.2: Die wichtigsten Klassen des *MatrixSelection*-Pakets

bereit (*AppendIncomingTransition()* etc.). Außerdem ist das Feuern von Transitionen sehr einfach im Netz (*FireTransition()*) oder direkt an der Transition (*Fire()*) möglich, was besonders für den Token-Replay-Algorithmus notwendig ist. Die Transitionen besitzen zudem Fields um Metadaten zu speichern, insbesondere wenn es sich bei der Transition nicht um eine Aktivität handelt, sondern um ein Hilfskonstrukt um die Prozesslogik darzustellen. Hierzu zählen Self-Loops, AND-Splits und -joins.

Die Visualisierung der Petrinetze wird in Abschnitt 5.6.1 beschrieben.

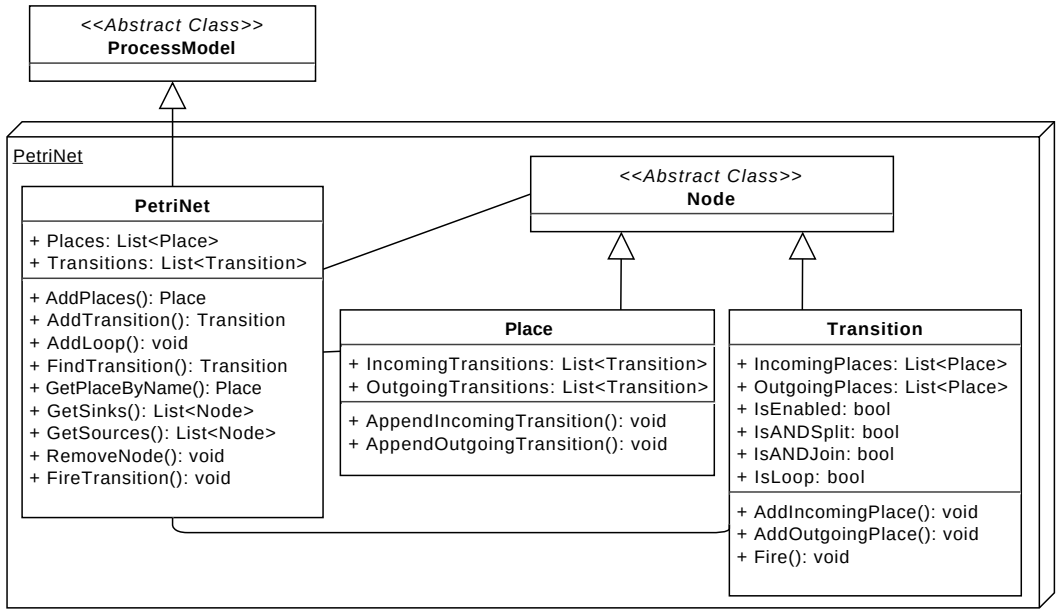


Abbildung 4.3: Die wichtigsten Klassen des *Model*-Pakets

Ein für das Model sehr wichtiges Paket ist das *Database*-Paket (siehe Abbildung 4.4). Es vereint Controller- und Model-Klassen, die der Einfachheit halber in einem Paket zusammengefasst sind. Seine wichtigsten Aufgaben sind, die Verbindung zu einer Datenbank herzustellen, Daten daraus zu laden, den SQL-Code dafür zu generieren und die Metadaten vorzuhalten. Abgesehen von der Eingabe der Zugangsdaten ist es für Benutzer des Pakets egal, welcher Datenbanktyp zugrunde liegt.

Die zentrale Klasse ist der statische *DBWorker*. Sie nimmt alle datenbankbezogenen Befehle wie das Herstellen einer Verbindung oder das Laden der Daten entgegen und setzt sie entsprechend dem aktuellen Datenbanktyp und Systemzustand um. Um eine Verbindung aufzubauen, wird außerhalb des Pakets ein *ConnectionParameters*-Objekt erstellt und der Methode *ConfigureDBConnetion()* übergeben. Via *OpenConnection()* wird die Verbindung hergestellt und mit *BuildMetadataRepository()* die Tabellen- und Spaltennamen, Inhalte der Dimensionstabellen und Fremdschlüsselbeziehungen geladen und in einem *MetaDataRepository*-Objekt gespeichert. Für die Datenbankanbindung gibt es die abstrakte Klasse *MPMdbConnection*, von der die datenbankspezifischen Klassen *MPMOracleConnection*, *MPMMySQLConnection*, *MPMMicrosoftSQL*, *MPMPostgreSQLConnection* und *MPMSQLiteConnection* abgeleitet sind. Diese beinhalten die spezifischen Implementierungen um eine Verbindung herzustellen, Metadaten auszulesen und Faktendaten zu laden. Hiervon ausgenommen ist jedoch die Erzeugung der SQL-Statements um für die ausgewählten Fields die Daten zu laden. Diese werden im *SQLCreator* generiert. Je nach Datenbanktyp unterscheiden sich die SQL-Statements syntaktisch. Aus diesem

Grund kann der *SQLCreator* für spezifische Datenbanktypen abgeleitet werden und die jeweiligen Besonderheiten berücksichtigen. Um dem Programm einen weiteren Datenbanktypen hinzuzufügen muss somit eine Ableitung von *MPMdbConnection* und gegebenenfalls vom *SQLCreator* implementiert werden.

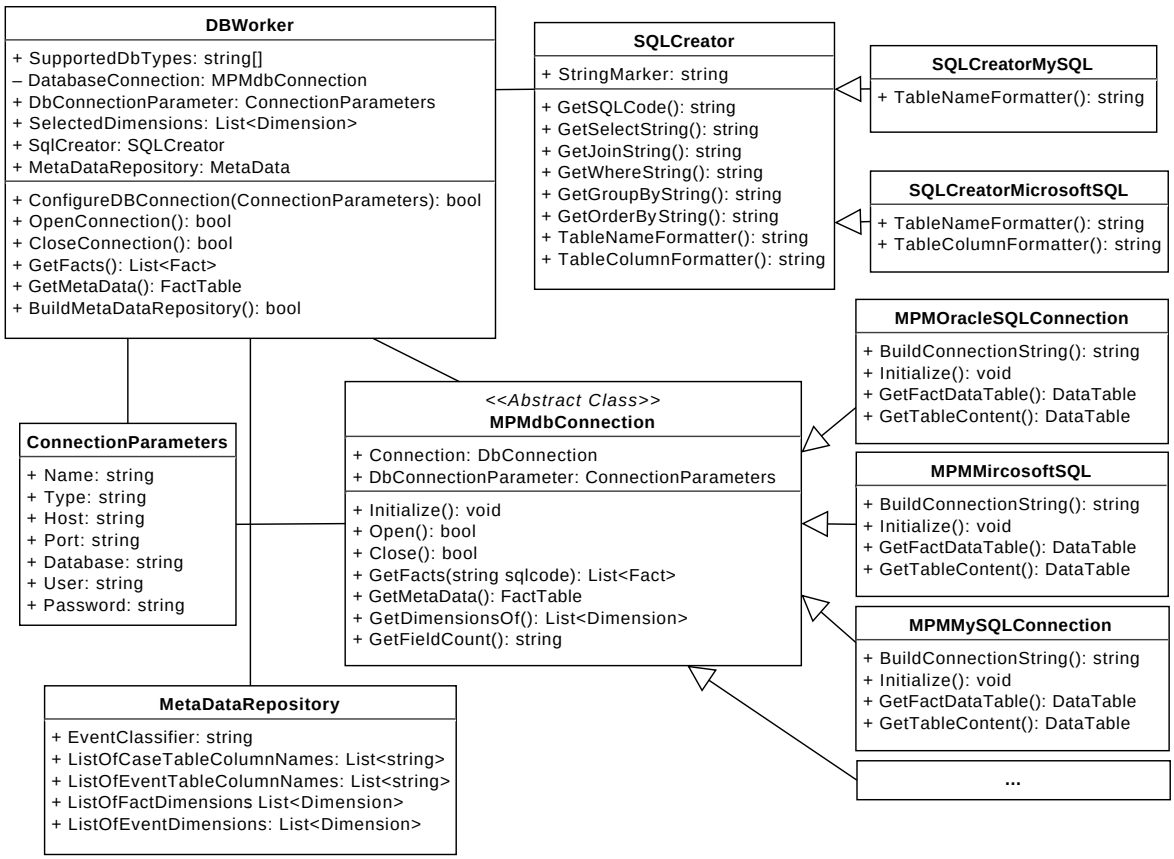


Abbildung 4.4: Die wichtigsten Klassen des *Database*-Pakets

4.2 Die Controller-Pakete

Da der *Process Cube Explorer* eine Software für das Process Mining ist, sind die Mining-Algorithmen der wichtigste Bestandteil. Sie befinden sich im Controllerpaket (siehe Abbildung 4.5). Obwohl sich die Algorithmen grundlegend unterscheiden, muss die Schnittstelle trotzdem leicht erweiterbar sein. Daher liegt allen Algorithmen das Interface *IMiner* zugrunde, welches die Funktion *Mine()* definiert, die ein *Processmodel* (siehe Abbildung 4.3) zurückgibt. Der Aufruf der Miner, geschieht über die statische *MinerFactory*, der mit *CreateMiner* der Name des Algorithmus sowie das *Field* (siehe Abbildung 4.2) übergeben wird. Sie erzeugt ein Objekt vom Typ *IMiner*, übergibt diesem das *Field* und gibt das Objekt zurück, an dem die *Mine()*-Methode aufgerufen werden kann.

Zum Hinzufügen eines weiteren Algorithmus muss also das Interface implementiert werden, die neue Klasse in der *MinerFactory* eingetragen werden und eine entsprechende Konfigurationsseite für die Benutzeroberfläche erstellt und in die *ListOfMiners* eingetragen werden.

Die Implementierung der drei umgesetzten Mining-Algorithmen wird in Abschnitt 5.5 genauer beschrieben.

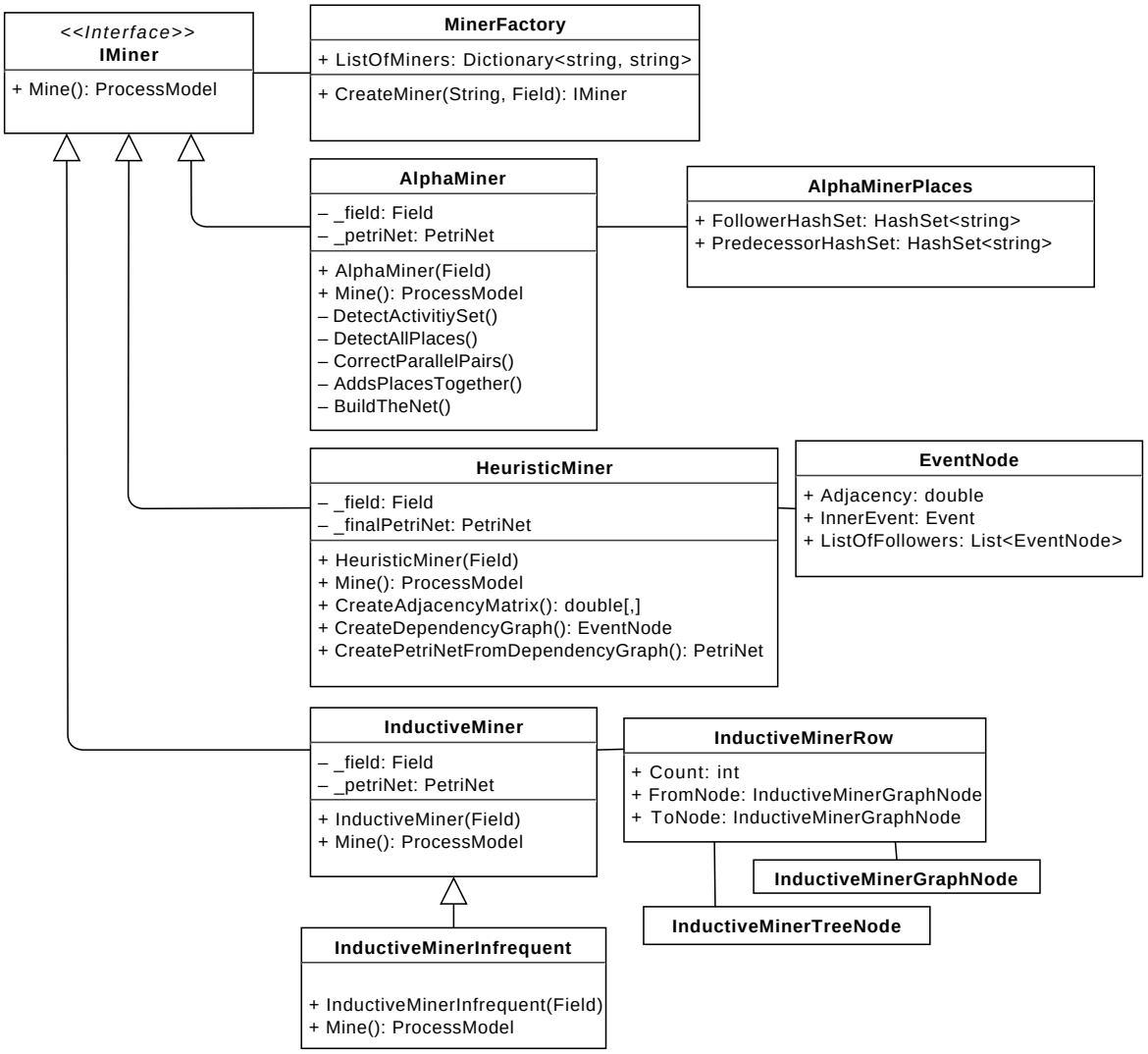


Abbildung 4.5: Die wichtigsten Klassen des *MiningAlgorithm*-Pakets

Eine wichtige Anforderung (siehe Abschnitt 3.1) war eine Schnittstelle für sogenannte Consolidation-Prozesse. Diese reduzieren die aus Prozessmodellen bestehende Ergebnismenge anhand bestimmter Kriterien. Die Kriterien werden vom Benutzer festgelegt. Der *IConsolidator*-Schnittstelle (siehe Abbildung 4.6) wird eine Liste von *Field*s übergeben und eine, in der Regel, kleinere Liste von *Field*s zurück erhalten. Um die Funktion der Schnittstelle zu zeigen ist ein *StandardConsolidator* implementiert, anhand dessen einfache Filter auf die Ergebnismatrix angewandt werden können (siehe Abschnitt 5.6.4).

Ein automatischer Prozessmodellvergleich ist im *Diff*-Paket (siehe Abbildung 4.8) umgesetzt. In der Benutzeroberfläche lassen sich Modelle auswählen die dann als Liste an eine Implementierung des *IDifference*-Interface übergeben werden. Diese gibt ein einzelnes *Processmodel* zurück, das dann

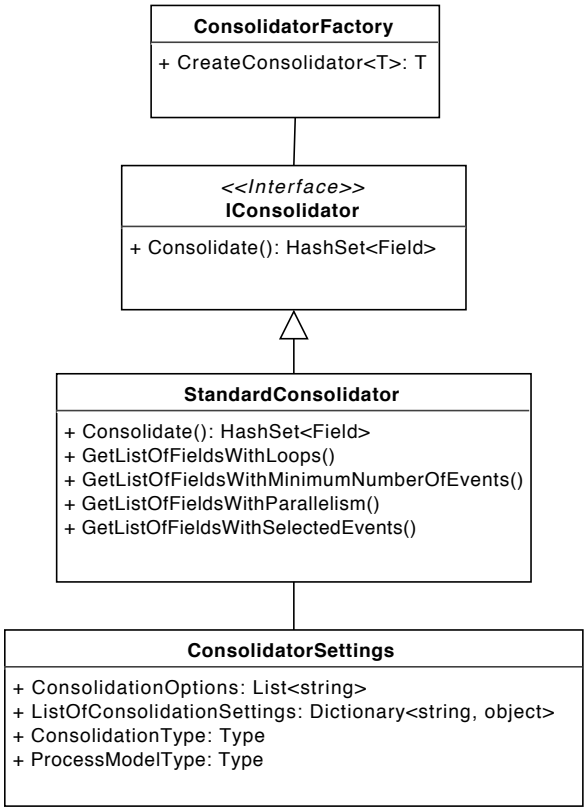


Abbildung 4.6: Die wichtigsten Klassen des *Consolidation*-Pakets

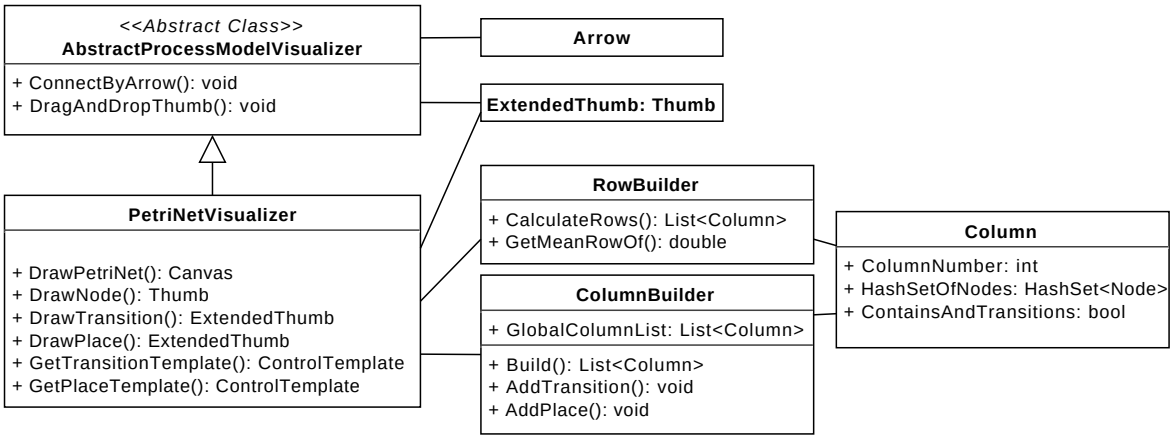


Abbildung 4.7: Die wichtigsten Klassen des *Visualization*-Pakets

von der GUI angezeigt wird. Aktuell ist eine Implementierung des Snapshot-Algorithmus vorhanden (siehe Abschnitte 2.6 und 5.6.5).

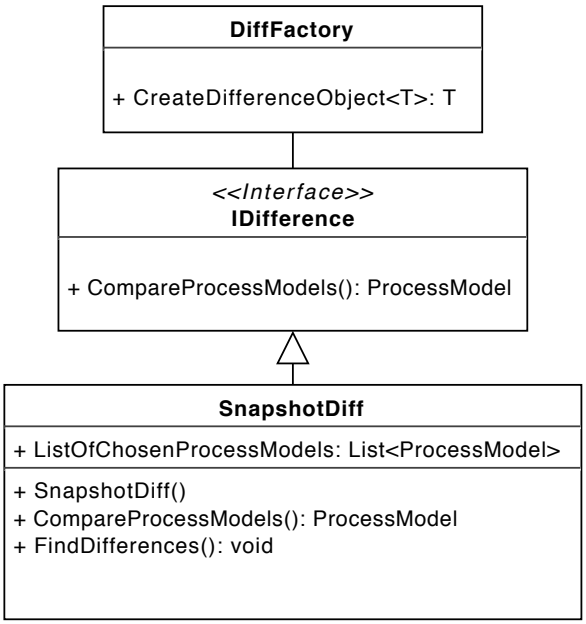


Abbildung 4.8: Die wichtigsten Klassen des *Diff*-Pakets

Die Algorithmen des *Conformance Checking* unterscheiden sich stark und benötigen sehr stark angepasste Benutzeroberflächen. Aus diesem Grund ist eine allgemeine Schnittstelle implementiert und die beiden Algorithmen, Comparing Footprint (siehe Abschnitt 2.4.1) und Token Replay (siehe Abschnitt 2.4.2), direkt in die Software integriert. Dies erleichtert die Bedienung dem Benutzer stark. Einzelheiten zur Implementierung stehen in Abschnitt 5.7, eine Übersicht über die Architektur des Paketes befindet sich in Abbildung 4.9.

4.3 Die View-Pakete

Die gesamte Benutzeroberfläche befindet sich im Paket *MainV2*. Es nutzt das ModernUI-Framework (siehe nächster Abschnitt 4.4) und damit eine Struktur von Fenstern (*ModernWindow*) und Seiten (*UserControl*s). Der Kasten *Pages* in Abbildung 4.10 zeigt die Seiten, die den Benutzer durch den Mining-Prozess führen: vom Laden der Daten (*P1connetion*), Bearbeiten der Metadaten (*P2metadata*) über die Auswahl der Dimensionen (*P3dimensionselection* und *P4eventselection*) und das Mining (*P5configuration* und *P6mining*) bis zur Anzeige der Ergebnisse (*P8results*). Die Programmeinstellungen (*Settings*) und Notizen (*Notes*) sind von überall erreichbar. *P4configuration* und *Settings* laden zudem weitere Unterseiten aus dem *Content*-Ordner nach. Besonders bei der Auswahl und Konfiguration der Mining-Algorithmen macht dies das Programm leicht erweiterbar. Aus vielen der Seiten wird auf den *Utilities*-Ordner zugegriffen, der GUI-spezifische Controller-Funktionalitäten gruppiert.

Diese Seiten werden alle im *MainWindow* angezeigt. Dieses wird durch die Klasse *InterfaceHelpers* unterstützt und stellt eine eigene *ErrorHandling*-Klasse für alle anderen Klassen bereit. Eine zweite *ModernWindow*-Instanz ist die Klasse *Viewer*, die durch Doppelklick auf die fertigen Modelle in

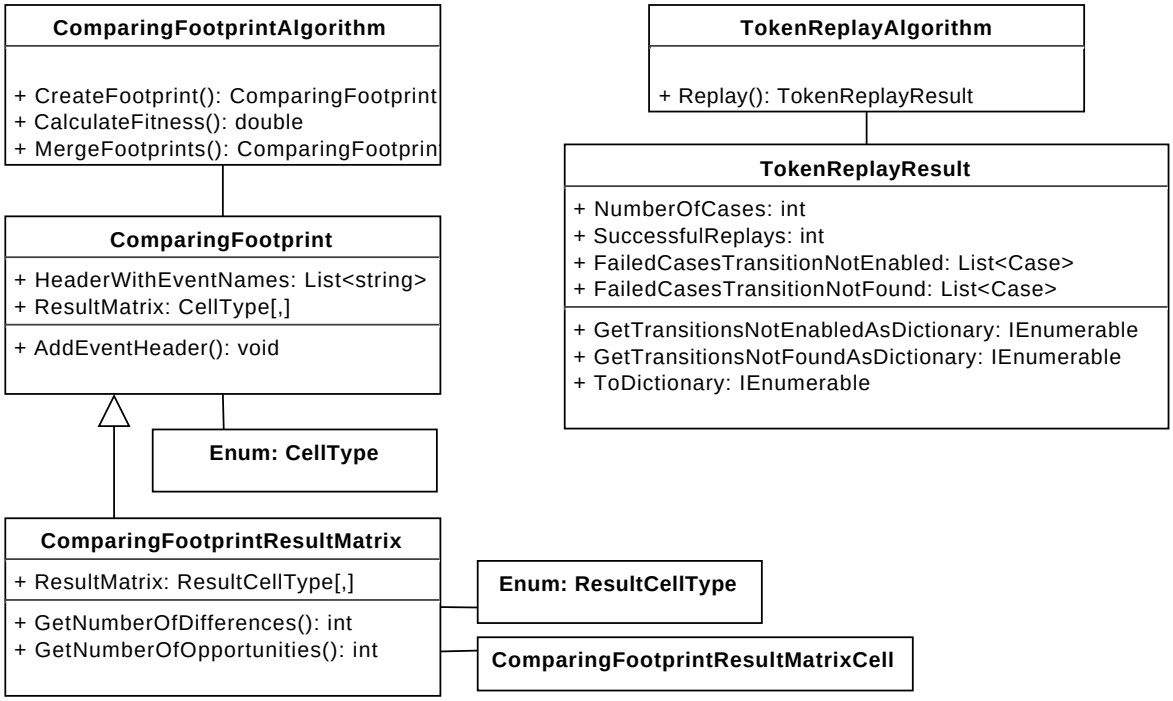


Abbildung 4.9: Die wichtigsten Klassen des *Conformance Checking*-Pakets

P8results geöffnet werden kann. Sie zeigt das Modell, Informationen dazu und die Möglichkeiten für Conformance Checking an.

Das *MainV2*-Paket enthält sämtliche Benutzeroberfläche mit Ausnahme der Darstellung der Prozessmodelle. Die Darstellung ist erweiterbar und befindet sich in *Visualization*. Die Klasse AbstractProcessModelVisualizer (siehe Abbildung 4.7) stellt einige Grundfunktionalitäten bereit, muss für die Darstellung eines Modells jedoch dafür erweitert werden.

Das *Visualization*-Paket ist stark mit der Oberfläche verknüpft, da in C# und WPF nur direkt auf einen Canvas gezeichnet werden kann. Die Implementierung der Petrinetze ist in Abschnitt 5.6.1 weiter erklärt.

4.4 Weitere Pakete

Die Softwarelösung enthält für jedes vorgestellte Paket ein Paket mit Unit-Tests. Diese sind an der Endung des Paketnamens zu erkennen. Als Hilfe für die Tests gibt es außerdem das Paket *ExampleData*, das Beispiel-Event Logs und -Petrinetze enthält.

Als weiteres Paket befindet sich *FirstFloor.ModernUI* in der Softwarelösung. Der Code stammt aus dem *Modern UI for WPF*-Projekt, einem Open-Source-Projekt das unter MS-PL-Lizenz auf https://mui.codeplex.com bereitgestellt wird. In der Software wird die aktuelle Version 1.0.5 benutzt, wobei einige Änderungen am Design vorgenommen wurden, weshalb der Quelltext in den Process Cube Explorer integriert wurde.

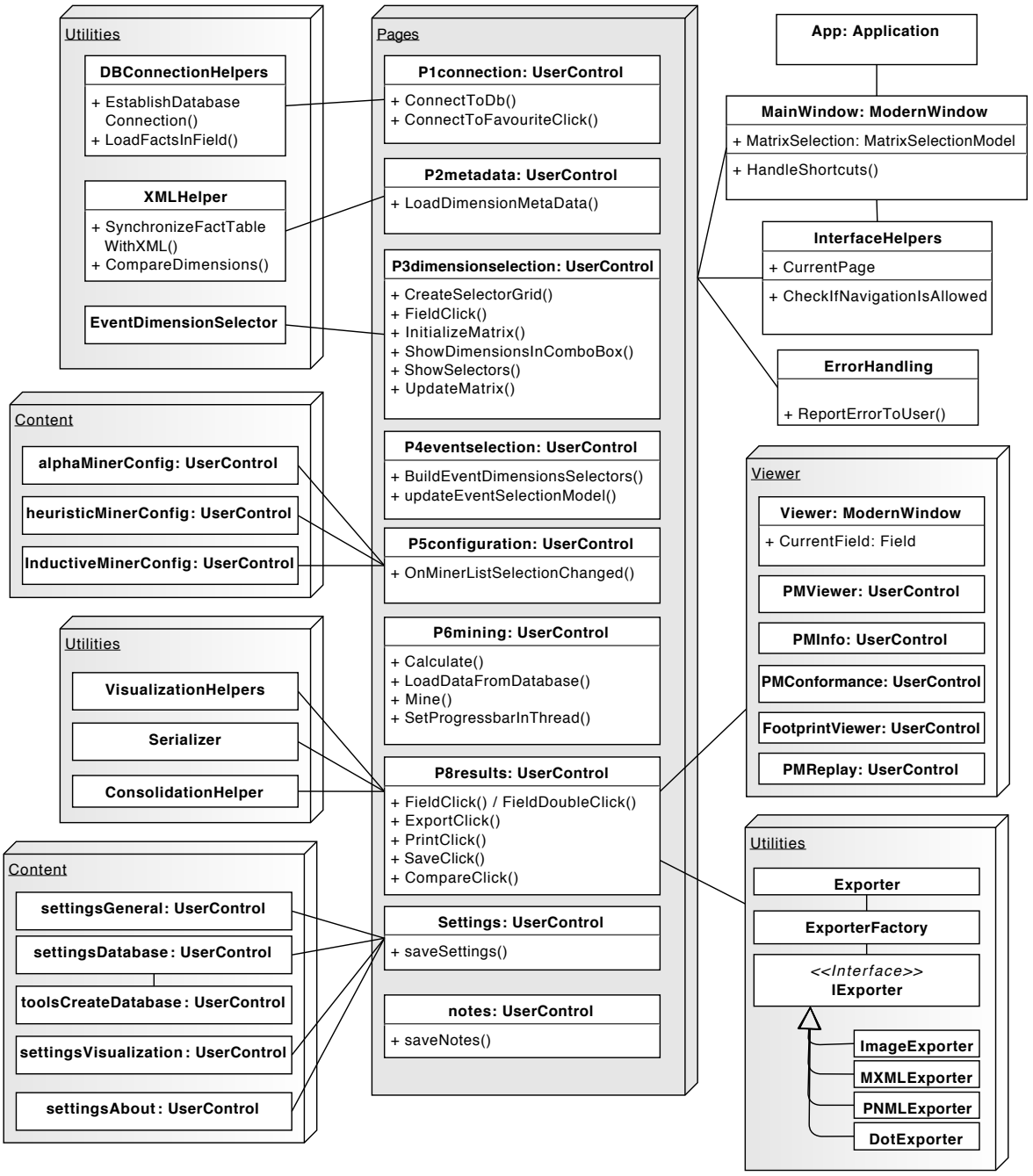


Abbildung 4.10: Die wichtigsten Klassen des *MainV2*-Pakets

5 Implementierung

Dieses Kapitel gewährt einen detaillierten Einblick in die Software. Zunächst wird beschrieben welche Technologien bei der Implementierung verwendet wurden. Anschließend wird das Data-Warehouse beschrieben das als Datengrundlage entwickelt und eingesetzt wurde. Insbesondere wird beschrieben, welche Voraussetzungen ein Data-Warehouse erfüllen muss, um mit der Software verwendet werden zu können. Danach wird darauf eingegangen wie die Anbindung einer Datenbank umgesetzt wurde und was zu tun ist, um eine neue Datenbank anzubinden. Im nächsten Abschnitt wird die Datenauswahl beschrieben, d.h. wie die Daten intern strukturiert sind und verarbeitet werden. Darüber hinaus wird auf die grafische Oberfläche für die Datenauswahl eingegangen. Als nächstes werden die implementierten Mining-Algorithmen und deren Besonderheiten dargestellt. Anschließend folgt die Beschreibung der Ergebnisdarstellung. Dort wird beschrieben wie die Ergebnisse vom Mining dargestellt werden, sowie welche Informationen und Funktionen für Analysen zur Verfügung stehen. Im vorletzten Abschnitt wird auf das Conformance Checking und die beiden implementierten Verfahren eingegangen. Abschließend kommt eine Beschreibung der zur Qualitätssicherung ergriffenen Maßnahmen.

5.1 Verwendete Technologien

In diesem Abschnitt werden in der Entwicklung eingesetzte Technologien und Werkzeuge beschrieben. Zunächst wird darauf eingegangen mit welchen Hilfsmitteln die Software implementiert wurde und warum diese verwendet wurden. Anschließend wird auf die Hilfsmittel eingegangen die beim ETL-Prozess verwendet wurden.

5.1.1 Software

Die Software wurde vollständig in der Programmiersprache C# entwickelt. Die Wahl fiel auf C#, weil die Sprache moderne Konzepte integriert, wie z. B. Garbage Collection, Propertys oder LINQ. Weiterhin ist die Sprache einfach in der Handhabung und objektorientiert. Für das grafische Interface wurde das ModernUI-Framework[1] und die Windows Presentation Foundation (WPF) verwendet. Dadurch konnte die Oberfläche schnell und optisch ansprechend entwickelt werden. Als Entwicklungsumgebung kam Visual Studio zum Einsatz, weil es eine umfassende Unterstützung in Programmierung mit C# bietet. Für die Versionskontrolle wurde Subversion[2] (kurz SVN) verwendet. Die Wahl fiel auf SVN, weil es zum einen Open Source ist und zum anderen sämtliche für die Projektgruppe benötigten Funktionen bereitstellt.

5.1.2 ETL-Prozess

Für den Aufbau des Data-Warehouse wurde primär das Softwarewerkzeug *Kettle*[3] eingesetzt. Damit wurden Daten aus Datenbanken ausgelesen, durch die Ausführung von SQL-Skripten transformiert und anschließend in ein Data-Warehouse geladen. Der gesamte Prozess wurde in Kettle modelliert

[1] ModernUI-Framework - `http://mui.codeplex.com/`
[2] Subversion - `http://subversion.apache.org`
[3] Kettle - `http://community.pentaho.com/projects/data-integration/`

und automatisiert durchgeführt. Als Hilfswerkzeuge wurden weiterhin *Oracle SQL Developer*[4] sowie *pgAdmin*[5] eingesetzt. Bei beiden Werkzeugen handelt es sich um Umgebungen, die den Zugriff auf und die Verwaltung von Datenbanken ermöglicht, wobei *Oracle SQL Developer* primär mit Oracle-Datenbanken und pgAdmin ausschließlich mit PostgresSQL-Datenbanken und davon abgeleiteten Datenbanken verwendet wird. Weiterhin wurde auch im ETL-Prozess SVN verwendet, um die Dateien untereinander auszutauschen.

5.2 Data-Warehouse

In diesem Abschnitt wird beschrieben welche Datenbank als Datenquelle für die Software implementiert wurde (Anforderung F1). Hierzu wird zunächst das Datenbankschema und dessen Konzept erläutert. Anschließend wird erklärt welche Regeln ein Datenbankschema einhalten muss, um als Datenquelle verwendet werden zu können. Zum Abschluss wird beschrieben, woher die in der Datenbank enthaltenen Daten stammen und wie diese Daten in das Data-Warehouse geladen wurden (Anforderung F2).

5.2.1 Datenbankschema

Das der Datenbank zugrundeliegende Schema wird in Abbildung 5.1 dargestellt. Es handelt sich um ein relationales Schema. Den Kern bilden die Tabellen FACT, CASE und EVENT. Wie der Name bereits verrät, enthält die Tabelle CASE sämtliche Cases. Die Cases werden anhand des Attributs case_id eindeutig identifiziert. Bei dem Attribut fact_id handelt es sich um einen Fremdschlüssel auf die Tabelle FACT. Diese enthält sämtliche Informationen über die Dimensionen der korrespondierenden Cases. Die in den Dimensionen enthaltenen Werte beschreiben die Eigenschaften des Cases, wie z. B. das Alter des Patienten oder die DRG des Cases. Diese Dimensionen werden anhand der jeweiligen Fremdschlüssel in der Tabelle FACT referenziert. FACT enthält ausschließlich Fremdschlüssel auf die Case Dimensionen sowie das Attribut fact_id. Die dritte zentrale Tabelle ist EVENT. Diese Tabelle enthält sämtliche Informationen über die Events des Cases. Ein Datensatz in EVENT repräsentiert immer ein Event. Jedes Event besitzt eine eindeutige id im Attribut event_id. Jedes Event referenziert über das Attribut case_id den Case zu dem es gehört. Weiterhin enthält jedes Event die Attribute activity, timestamp, volume und unit. Das Attribut activity beschreibt die durchgeführte Aktivität, timestamp liefert den Zeitpunkt als das Event stattgefunden hat. Die restlichen Attribute (eventtype_id, care_unit, id und care_giver_id) sind Fremdschlüssel auf die Dimensionen der Events. Diese Dimensionen beschreiben Eigenschaften eines Events, wie z.B. der Eventtyp des Events oder die Station auf der es stattgefunden hat.

Denkt man an die Metapher des Datenwürfels, so enthält die Tabelle FACT die Zellen des Datenwürfels. Die Dimension des Datenwürfels werden über die Dimensionen definiert, die in Tabelle FACT referenziert werden. Den Inhalt einer Zelle stellen die Cases mit ihren Events dar. Da Events ebenfalls Dimensionen aufweisen, können diese ebenfalls als Datenwürfel vorgestellt werden. Für jeden Case entsteht somit im Prinzip ein eigener Datenwürfel, in dessen Zellen sich die Events befinden und

[4] Oracle SQL Developer - `http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html?ssSourceSiteId=otnpt`
[5] pgAdmin - `http://www.pgadmin.org`

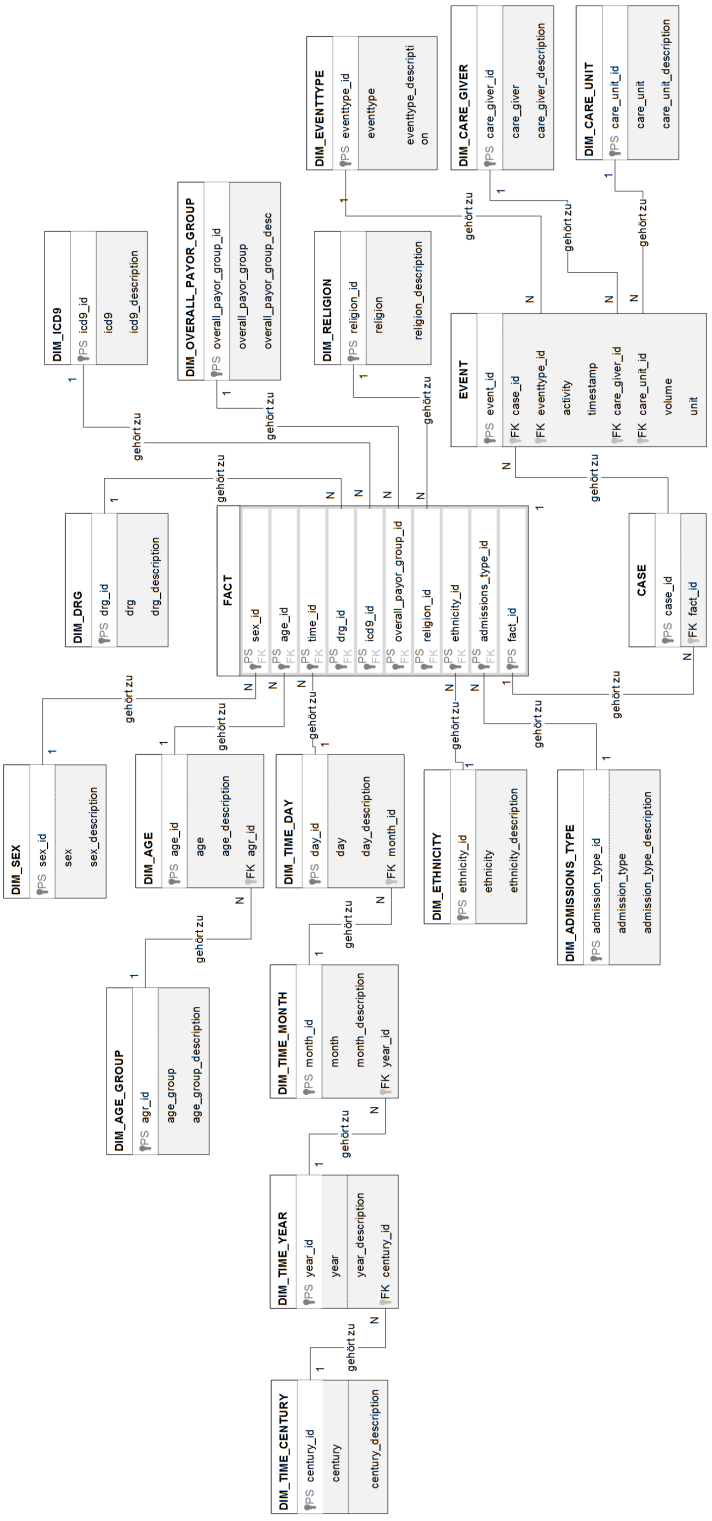


Abbildung 5.1: Datenbankschema

dessen Dimensionen sich aus den in Tabelle EVENT referenzierten Dimensionen ergeben. In der menschlichen Vorstellung bedeutet das ganze, dass in jeder Zelle eines Datenwürfels sich wiederum Datenwürfel befinden. Aus diesem Konzept ergeben sich weitreichende Analysemöglichkeiten. Es kann gezielt nach bestimmten Kriterien des Cases sowie Events abgefragt werden.

5.2.2 Einschränkungen des Datenbankschemas

Um eine Datenbank mit einem anderen Schema als dem Vorgestellten zu nutzen, müssen einige Regeln eingehalten werden. Andernfalls kann eine Datenbank nicht mit der Software verwendet werden. Entscheidend ist, dass die Tabellen FACT, CASE und EVENT vorhanden sind. Die Bezeichner für diese Tabellen können jedoch frei gewählt werden. Weiterhin sind die Fremdschlüsselbeziehungen sehr wichtig. Jede Tabelle die über einen Fremdschlüssel in Tabelle FACT referenziert wird, wird als Case Dimension interpretiert. Das Referenzieren von Tabellen die keine Dimensionen sind, ist in FACT laut den Regeln nicht erlaubt. Tabelle CASE darf genau ein Fremdschlüssel besitzen. Dieser muss auf Tabelle FACT referenzieren. Dadurch wird sichergestellt, dass die Tabelle FACT mit der CASE Tabelle referenziert wird. Unter Beachtung der beschriebenen referenziellen Integrität können FACT und CASE beliebige andere Attribute enthalten. Für die Tabelle EVENT gilt im Prinzip das gleiche. Jede Tabelle die über einen Fremdschlüssel aus der Tabelle EVENT heraus referenziert wird, wird als Event Dimension interpretiert. Einzige Ausnahme bildet hier das Attribut, dass auf die Tabelle CASE referenziert. Bei dem oben beschriebenen Schema wäre es das Attribut case_id. Die Tabelle EVENT muss zwangsläufig genau ein solches Attribut besitzen. Weiterhin muss die Tabelle ein Attribut beinhalten, dass sich als Classifier definieren lässt (z. B. eine Aktivität) und ein weiteres Attribut, dass sich zum Bestimmen der Reihenfolge der Events eignet (z. B. ein Timestamp). Die restlichen Attribute sind optional. Die Bezeichner für sämtliche Attribute aller Tabellen können frei gewählt werden. Einzige Ausnahme bilden beim aktuellem Stand die fact_id in FACT und fact_id in CASE (siehe Abbildung 5.1). Diese beiden Attribute müssen den gleichen Bezeichner tragen, dürfen aber anders heißen. Gleiches gilt für case_id in der Tabelle EVENT und case_id in der Tabelle CASE. Jede Dimension, d.h. sowohl von Cases als auch Events, muss im ersten Attribut den Primärschlüssel und im zweiten Attribut die eigentlichen Werte enthalten. Alle Attribute die danach kommen sind optional.

5.2.3 Datenquelle

Nachdem das Datenbankschema und das Konzept dahinter beschrieben wurde, wird an dieser Stelle darauf eingegangen, woher die Daten in der implementierten Datenbank stammen. Die Daten wurden vom PhysioNet bezogen. PhysioNet ist ein Zusammenschluss von Forschern diverser Universitäten mit dem Ziel physiologische Daten sowie Software zur Analyse solcher Daten bereitzustellen bzw. zu entwickeln. Im Rahmen des PhysioNet gibt es das sogenannte Mimic-II Projekt. Dieses Projekt läuft seit 2001 mit dem Ziel eine Datenbank mit physiologischen Daten aufzubauen. Die Datenbank enthält eine Vielzahl von Daten zu diversen Cases. Die Daten stammen aus dem Zeitraum 2001-2008 und beinhalten sowohl Eigenschaften des Cases (z. B. Alter des Patienten oder DRG) sowie Eventdaten (z. B. durchgeführte Prozeduren). Im Rahmen des Projekts wurden diese Daten, in stark anonymisierter Form, im Rahmen der Entwicklung der Software als Datengrundlage verwendet.

5.2.4 ETL

Die zuvor beschriebenen Daten wurden im Rahmen eines ETL-Prozesses in das Data-Warehouse importiert. Das Vorgehen gliedert sich in vier Schritte. Als erstes wurden die Daten aus der Mimic-II Datenbank vollständig und unverändert in eine eigene Datenbank kopiert. Anschließend wurde ausschließlich mit den kopierten Daten gearbeitet. Das wurde gemacht um im Rahmen des Entwicklungsprozesses nicht jedes Mal auf die Mimic-II Datenbank zugreifen zu müssen. Die nächsten Schritte wurden alle mit Kettle umgesetzt. Zunächst wurden die Dimensionen erzeugt, d.h. die Case- sowie Event-Dimensionen, wobei einige spezielle Dimensionen aufgrund praktischer Gründe im Rahmen des späteren Transformationsprozesses generiert wurden. Anschließend wurden relevante Daten in temporäre Tabellen extrahiert. Diese extrahierten Daten wurden schließlich transformiert. Transformiert bedeutet, dass die Daten syntaktisch angepasst wurden. Andere Maßnahmen waren nicht notwendig, weil die Daten bereits in entsprechender Qualität vorlagen. Im letzten Schritt wurden die Daten in das Data-Warehouse oder genauer gesagt in die Tabellen FACT, CASE und EVENT geladen.

5.3 Datenbankanbindung

Das Paket Database bildet die zentrale Schnittstelle zu den verschiedenen Datenbanksystemen (Anforderung F3). Unterstützt werden MySQL (Anforderung F11), Oracle SQL, MS-SQL und SQLite Datenbanken. Wie im Abschnitt 4.1 beschrieben sind alle Verbindungsklassen von der Abstrakten Klasse `MPMdBConnection` abgeleitet. In dieser sind die gemeinsamen Funktionen und Eigenschaften definiert. Die abgeleiteten Klassen wiederum enthalten spezielle Implementierungen für den jeweiligen Datenbanktyp.

Die Methode `public DataTable GetReferencingDataTable(String tablename)` führt beispielsweise eine SQL-Abfrage aus, die alle referenzierenden Tabellen auf die Tabelle `tablename` zurückgibt. Diese sind DB-Typ spezifisch, für Oracle werden beispielsweise die `all_constraints`, `all_cons_columns` Tabellen vereinigt, für MS-SQL werden `sys.tables`, `sys.colums`, `sys.foreign_keys` und `sys.foreign_key_columns` vereinigt. Zusätzlich werden die verschiedenen Datenbanktreiber der verschiedenen Hersteller über diese Klassen eingebunden.

Die Klasse DBWorker fungiert als Controller und Connection Factory, welches die Einbindung weiterer Datenbankverbindungen erleichtert (Anforderung F12). Basierend auf den Eigenschaften eines übergebenen `ConnectionParameter` Objekts wird die jeweilige `MPMdBConnection` Implementierung instanziiert. Des Weiteren werden Anfragen und Datenbankverbindungsoperationen über diese Klasse gesteuert.

Die Klasse `SQLCreator` und ihre Ableitungen dienen der korrekten Generierung von SQL Statements. Select, Join, Where und From Ausdrücke werden anhand übergebener Parameter generiert. Besonderheiten der verschiedenen DB-Typen werden über die DB-spezifischen Ableitungen des `SQLCreator` umgesetzt. Dies sind beispielsweise das Format von Tabellennamen oder die Formatierung von Stringausdrücken in SQL-Statements.

`CipherUtility` dient der symmetrischen Ver-/Entschlüsselung von Strings. Hierzu wird ein programmweiter Passcode, sowie für jeden zu verschlüsselnden String ein Salt, generiert. Diese werden dann AES ver-/entschlüsselt und der chiffrierte/decodierte Text wird zurück gegeben.

`ConnectionParameter` dient dem Speichern von Verbindungsinformationen wie Name, Passwort, IP Adresse, Port und Datenbanktyp. Das Propertie `Passwort` ver-/entschlüsselt automatisch Passwort-Strings um die interne Verarbeitung zu erleichtern.

5.4 Datenauswahl

Nachdem eine Verbindung zur Datenbank aufgebaut wurde kann mit der Datenauswahl begonnen werden. Die Datenauswahl dient der Selektion von Eventdaten aus einem multidimensionalen Event Log und ist die Grundlage für das anschließende Process Mining. Die Darstellung der Eventdaten erfolgt in der Software aus Gründen der Übersichtlichkeit als zweidimensionale Matrix. Grundbestandteile der Datenauswahl sind Dimensionen, Level und Filter die eine effiziente und präzise Navigation durch die Eventdaten ermöglichen. Nachdem die Navigation vollzogen wurde, werden die ausgewählten Filterdaten in der zweidimensionalen Matrix dargestellt. Anschließend kann mit der Konfiguration des Process Mining-Algorithmus fortgefahren werden, der schließlich auf die Datenauswahl angewandt wird.

5.4.1 Aufbau der Datenauswahl

Der Aufbau der Datenauswahl gliedert sich grundlegend in zwei Bereiche, einem Filterbereich sowie einem Darstellungsbereich. Im Darstellungsbereich werden die Eventdaten in einer zweidimensionalen Matrix dargestellt. Die Anzahl der Zellen in der Matrix richten sich dabei an die Anzahl der ausgewählten Daten im Filter der ersten beiden Dimensionen. Werden z. B. zwei Werte im Filter der ersten Dimension sowie zwei Werte im Filter der zweiten Dimension ausgewählt, entspricht dieses einer Matrix mit vier (2*2) Zellen. Die Filterdaten der restlichen Dimensionen werden in der zweidimensionalen Matrix nicht mit angezeigt. Die folgende Abbildung 5.2 stellt den Aufbau der Datenauswahl dar.

Die Anzahl der im Filterbereich verfügbaren Dimensionen wird in der Software, je nach Anzahl der vorhandenen Dimensionen im Data-Warehouse, dynamisch erzeugt. Ein Beispiel für eine Dimension wäre z. B. Zeit. Des Weiteren kann zu jeder Dimension ein Level ausgewählt werden, welches die Dimensionsdaten in verschiedene Granularitäten einteilt. Ein Level für die Dimension Zeit wären z. B. die Granularitäten Tag, Jahr oder Monat. Nach Auswahl eines Levels können schließlich im Filter die gewünschten Daten ausgewählt werden. Nachdem die Daten im Filter ausgewählt wurden, und dieses durch einen Klick auf den Update-Button bestätigt wurde, wird die zweidimensionale Matrix erzeugt.

5.4.2 Initialisierung der Datenauswahl

Sobald eine Verbindung zur Datenbank aufgebaut wurde, wird über die Funktion `DBWorker.BuildMetadataRepository` das Metadaten Repository aufgebaut. Der DBWorker lädt dabei alle Dimensionen sowie die Struktur der Case- und Event-Tabellen in einem Objekt der Klasse `MetaDataRepository`, welches die Grundstruktur der Metadaten für die Datenauswahl darstellt.

Nachdem das Metadaten Repository aufgebaut wurde, werden über die Funktion `InitializeMatrix` die Steuerelemente der Datenauswahl mit Werten initialisiert. Dabei wird für jede Dimension im Metadaten Repository über die Funktion `ShowDimensionInCombobox` die jeweilige Combo-

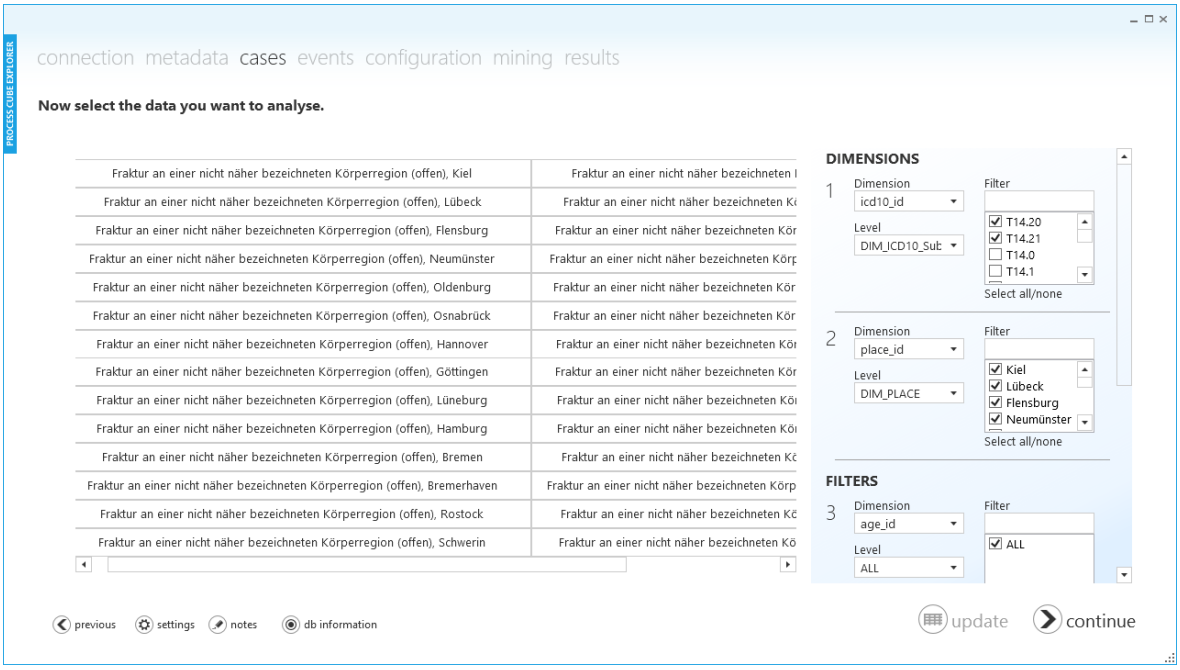


Abbildung 5.2: Datenauswahl

box gefüllt. Des Weiteren verfügt jede Combobox über ein `SelectionChanged` Event, welches ausgelöst wird sobald eine Änderung der Auswahl vorgenommen wird. Dieses Event führt schließlich dazu, dass die Comboboxen für `Level` und die Listboxen der `Filter` mit Werten aus dem Metadaten Repository passend zur Dimension gefüllt werden. Abschließend wird über die Funktion `UpdateMatrix` die zweidimensionale Matrix vom Typ `ItemsControl` aktualisiert und grafisch dargestellt.

5.4.3 Navigation in der Datenauswahl

Nachdem die Initialisierung der Datenauswahl vorgenommen wurde, kann mit der Navigation begonnen werden. Hierbei stehen dem Benutzer alle Dimensionen zur Verfügung die im Data-Warehouse vorhanden sind. Dabei kann jeder Dimension ein Level sowie ein Filter zugeordnet werden. Sobald eine Dimension ausgewählt wurde, wird das `SelectionChanged`-Event der jeweiligen Combobox aufgerufen. Dieses Event fügt der Combobox `Level` die verfügbaren Level der ausgewählten Dimension hinzu. Des Weiteren wird die ausgewählte Dimension in der Klasse `MatrixSelectionModel` hinterlegt. Die Combobox `Level` verfügt wiederum über ein `SelectionChanged`-Event welches ausgelöst wird, sobald ein Level ausgewählt wurde. Die Auswahl führt dazu, dass die Listbox `Filter` mit einer Liste von Objekten der Klasse `DimensionContent` gefüllt wird. Darüber hinaus wird auch das ausgewählte Level in der Klasse `MatrixSelectionModel` gespeichert. Auch die Listbox `Filter` besitzt ein `SelectionChanged`-Event. Dieses Event sorgt dafür, dass alle ausgewählten Elemente des Filters im `MatrixSelectionModel` hinterlegt werden. Des Weiteren ruft das Event die Funktion `UpdateMatrix` auf. In dieser Funktion werden die `Fields` für die in der `MatrixSelectionModel` hinterlegten Dimensionen erzeugt. Die `Fields` enthalten unter anderem die `SelectedDimensions` sowie den generierten SQL-Code der für Abfragen der ausgewählten Daten

aus dem Data Warehouse benötigt wird. Abschließend wird der zweidimensionalen Matrix vom Typ `ItemsControl` als `ItemSource` die `Fields` der Klasse `MatrixSelectionModel` über der Funktion `GetFields` zugewiesen.

5.5 Mining-Algorithmen

Die Mining-Algorithmen stellen einen wichtigen Bestandteil der Software dar. Es wurden drei Algorithmen implementiert: Der *Alpha Miner*, da er der einfachste und am weitesten verbreitete Algorithmus ist, der *Heuristic Miner*, da er bewährt ist und gute und vor allem übersichtliche Ergebnisse liefert, und der *Inductive Miner* mit der *infrequent*-Variante, da er der aktuellste ist (Anforderung F8).

Die Implementierung der drei Algorithmen wird in diesem Abschnitt genauer erläutert.

Neben einer soliden Auswahl an bereits implementierten Algorithmen war es jedoch sehr wichtig, eine einfache Erweiterbarkeit zu gewährleisten. Dafür wurde das Interface `IMiner` umgesetzt, das die Methode `Mine()` vorgibt, die ein Objekt der Klasse `Processmodel` zurückgibt. Durch die Klasse `MinerFactory` können nun alle Miner, die das Interface implementieren, ausgewählt werden (Anforderung F21). Listing 5.1 zeigt den Aufruf aus der Benutzungsoberfläche

```
IMiner miner =
  MinerFactory.CreateMiner(MinerSettings.MinerName, field);
ProcessModel resultingProcessModel = miner.Mine();
```

Listing 5.1: Erzeugen eines Miners und Mining-Prozess

Die Factory gibt je nach `MinerName` ein Objekt des richtigen Typs zurück. Der Name stammt aus dem Dictionary `MinerFactory.ListOfMiners`, in dem auch der Pfad zur Konfigurationsseite angegeben ist. Um einen weiteren Mining-Algorithmus hinzuzufügen muss also lediglich das Dictionary und die switch-case-Abfrage in `MinerFactory.CreateMiner()` aktualisiert werden. Die Miner bekommen dann ein `Field`-Objekt im Konstruktor übergeben, in dem sämtliche Daten enthalten sind. Die folgenden Abschnitte zeigen, wie darauf Mining-Algorithmen implementiert werden können.

5.5.1 Alpha Miner

Die Implementierung des Alpha Miners besteht aus drei groben Schritten. Im ersten Schritt werden alle Aktivitäten sowie die Start- und End-Aktivitäten ermittelt. Im zweiten Schritt werden in der `DetectAllPlaces` Methode alle möglichen Places ermittelt. Dabei kann es dazu kommen das einige Places die gleichen Vorgänger oder Nachfolger haben, diese Places werden in der `AddsPlacesTogether` Methode zu einem Place zusammengefasst. Das folgende Beispiel soll dies genauer verdeutlichen. Die Places {A} -> {B} und {A} -> {C} werden zu einem Place {A} -> {B,C} zusammengefasst. Zum Schluss des zweiten Schrittes werden noch alle redundanten Places entfernt. Im dritten und damit letzten Schritt wird nur noch das Petrinetz zusammengebaut und an die Visualisierung weitergegeben [Aal10], [Wes12]. Da der Standard Alpha Miner keine kurzen Loops (von der Länge 1 oder 2) darstellen kann, wurde der Standard Alpha Miners um den Alpha Miner+ und Alpha Miner++ erweitert.

Der Alpha Miner+ kann im Gegensatz zu dem Standard Alpha Miner, Loops der Länge 2 erkennen. Das Problem entsteht dadurch, dass der Standard-Algorithmus nicht zwischen parallelen Ausführungen und Loops unterscheiden kann, da dieser die Aktivitäten nur paarweise untersuchen kann. In dem folgenden beispielhaften Event Log ist eine Parallelität der Events a und b vorhanden, die durch das genannte Problem nicht entdeckt werden kann:

Der Alpha Miner+ kann im Gegensatz zu dem Standard Alpha Miner, Loops der Länge 2 erkennen. Das Problem entsteht dadurch, dass der Standard-Algorithmus nicht zwischen parallelen Ausführungen und Loops unterscheiden kann, da dieser die Aktivitäten nur paarweise untersuchen kann [Wes12]. In dem folgenden beispielhaften Event Log ist eine Parallelität der Events a und b vorhanden, die durch das genannte Problem nicht entdeckt werden kann:

<. . . , a, b, . . . > und <. . . , b, a, . . . > bzw. der Eintrag <. . . , a, b, a, . . . >. Um das Problem zu lösen, untersucht der Alpha+ Algorithmus in der `DetectAllPlaces` Methode, die Kausalität der Aktivitäten nicht paarweise sondern in dreier Gruppen, dadurch kann die Parallelität entdeckt werden. In der Methode `CorrectParallelPairs` werden anschließend eventuell fehlerhaft ermittelte Parallel-Verbindungen korrigiert [MDAW05].

Der Alpha Miner++ kann zusätzlich zu den Loops der Länge 2 auch die Loops der Länge 1 erkennen. Loops mit der Länge 1 sind Aktivitäten die im Event Log auf sich selbst verweisen, z. B. <. . . , a, a, . . . >.

Der Alpha Miner++ kann zusätzlich zu den Loops der Länge 2 auch die Loops der Länge 1 erkennen. Loops mit der Länge 1 sind Aktivitäten die im Event Log auf sich selbst verweisen, z. B. <. . . , a, a, . . . > [Wes12].

In der `Preprocessing` Methode werden alle diese Aktivitäten identifiziert und vorerst entfernt. Anschließend werden in der `Postprocessing` Methode die Loops an den richtigen Stellen eingefügt.

5.5.2 Heuristic Miner

Die Implementierung des *Heuristic Miners* richtet sich weitestgehend nach der Beschreibung in Abschnitt 2.3.2 und [WAM06], gleicht aber an einigen Stellen die Schwächen des Ansatzes aus.

Im ersten Schritt wird die Adjazenzmatrix des Event Logs aufgestellt. Dafür wird ein zweidimensionales, quadratisches Array erzeugt in dem jedes Event genau eine Spalte und Reihe hat. Jeder Eintrag stellt die Adjazenz zweier Events dar. Bei der Berechnung wird jedoch nur die obere Dreiecksmatrix berechnet, da $a \Rightarrow b = -1 \cdot (b \Rightarrow a)$ gilt (siehe Abschnitt 2.3.2). So sind nur etwa die Hälfte der Berechnungen nötig. Ein vereinfachter Codeausschnitt ist in Listing 5.2 zu sehen.

```
for (var row = 0; row < count; row++)
  for (var col = 0; col < count; col++)
   if (column < row)
     Matrix[row,col] = -Matrix[col,row];
   else
    Matrix[row,col] = GetAdjacency(Events[row], Events[col], Log);
```

Listing 5.2: Aufstellen der Adjazenzmatrix

Der zweite Schritt besteht darin, das Startevent zu finden. Hierfür wird die Spalte der Adjazenzmatrix gesucht, die nur negative Einträge enthält. In der Praxis können dies jedoch auch mehrere Spalten sein, weshalb die Implementierung nun für jedes Startevent ein Prozessmodell erstellt und diese später verknüpft.

Im dritten Schritt wird aus der Adjazenzmatrix ein Abhängigkeitsgraph erstellt. Hierfür gibt es eine eigene Datenstruktur `EventNode`, die ein `Event`, dessen Adjazenz zur vorherigen Node sowie eine Liste aller nachfolgenden Events speichert. Hierfür werden alle Events gewählt deren Eintrag in der Adazenzmatrix größer als der übergebene Schwellenwert (*Adjacency threshold*) ist.

Der Abhängigkeitsgraph wird repräsentiert durch die erste `EventNode`. Diese speichert sämtliche Informationen in der Liste der Nachfolger. Aus diesem Grund muss sie rekursiv durchlaufen werden.

Darauf folgt der vierte Schritt, in dem der Abhängigkeitsgraph in ein Petrinetz umgewandelt wird. Für einzelne aufeinanderfolgende Events ist dies sehr einfach, wird mit steigendem Parallelitätsgrad jedoch immer schwieriger. Die wichtigste Methode hier ist `HandleNode()`. Bei mehreren Nachfolgern zählt sie zunächst mit der zweiten Formel aus Abschnitt 2.3.2 die AND-Beziehungen der Nachfolger. Deckt sich diese mit der Anzahl der Nachfolger, wird ein AND-Split eingefügt und alle Nachfolger angehängt. Ist sie gleich Null stehen alle Events in einer XOR-Beziehung, also werden sie direkt an den aktuellen `Place` angehängt.

Unterscheidet sich die Anzahl der Nachfolger und die Anzahl der AND-Beziehungen gibt es eine Sonderbehandlung für alle Konstellationen in denen drei Events zueinander stehen können, bei mehr als drei Events werden alle in eine XOR-Beziehung gesetzt. Hier bietet der Algorithmus keine andere Lösung. Als Erweiterung zum Algorithmus werden in der Implementierung auch optionale Events erkannt, solange der Parallelitätsgrad zwei nicht übersteigt. Am Ende des vierten Schrittes steht ein fertiges Petrinetz das nun noch geordnet, aufgeräumt und dann zurückgegeben wird.

5.5.3 Inductive Miner infrequent

Die Implementierung des Mining-Algorithmus *Inductive Miner* wird in die Funktionen *Follower-Graph*, *Prozess-Baum* und *Divide-and-Conquer* unterteilt. Bei dem *Inductive Miner infrequent* kommt die Funktion *Filterung infrequent* dazu.

Wie im Abschnitt 2.3.3 beschrieben, basiert die Umsetzung auf dem Ansatz von S. Leemans [LFA13a, LFA13b]. Mit Ausnahme der Infrequence-Filterung ist der Inductive Miner und Inductive Miner-Infrequent identisch.

Follower-Graph

Durch den Aufruf der Methode `Mine()` werden zunächst ein *Directly-Follower-Dictionary* und ein *Eventualy-Follower-Dictionary* erstellt, welche die Beziehung zwischen zwei Events und ihrer Häufigkeit darstellen. Im Anschluss hieran werden beim Inductive Miner Infrequent alle Verbindungen eliminiert die unterhalb eines vom Benutzer definierten Schwellenwert liegen.

Für jedes Event wird ein `InductiveMinerGraphNode` erstellt. In den Listen `Follower List` und `EventualyFollowerList` werden die Beziehungen zu folgenden Knoten festgehalten. Dieses stellt den Eventual-Follow-Graph und Direct-Follow-Graph dar, die zur weiteren Verarbeitung im Algorithmus benötigt werden.

Filterung infrequent

Ein weiterer Aufruf in der Methode `Mine()` stellt die Funktion zur Filterung seltener Verbindungen zwischen den Events dar. Hierzu werden die Eventualy- und DirectlyFollower-Listen nach der Anzahl von Verbindungen aufsteigend sortiert. Die maximale Häufigkeit wird mit dem vom Benutzer definierten Schwellenwert multipliziert. Verbindungen die unter dieser minimalen Häufigkeitsgrenze liegen, werden eliminiert und der `EventualyFollowerGraph` dementsprechend angepasst. Im Anhang unter 10.2 befindet sich der Quellcode zur Eliminierung von infrequenten Verbindungen.

Divide-and-Conquer

Als nächster Schritt in der Mine()-Methode werden ein Petrinetz, der Graph-Node für das Start-Event sowie das Start-Event übergeben. In einem rekursiven Divide-and-Conquer-Verfahren werden die Graphen im Prozess-Baum so lange zerteilt (Loop-, XOR-, Sequence-, Parallelitäten-Identifizierung), bis nur noch einzelne Graph-Nodes übrig bleiben. Diese Leafs stellen einzelne Events dar. Das vollständige Divide-and-Conquer-Verfahren befindet sich unter 10.1 im Anhang. Des Weiteren sind die Funktionen, die das Divide-and-Conquer-Verfahren aufrufen, wie folgt im Anhang:

- Überprüfung nach Sequencen 10.5
- Überprüfung nach Loops 10.6
- Überprüfung nach Exclusive Choise 10.7
- Überprüfung nach Parallelitäten 10.8

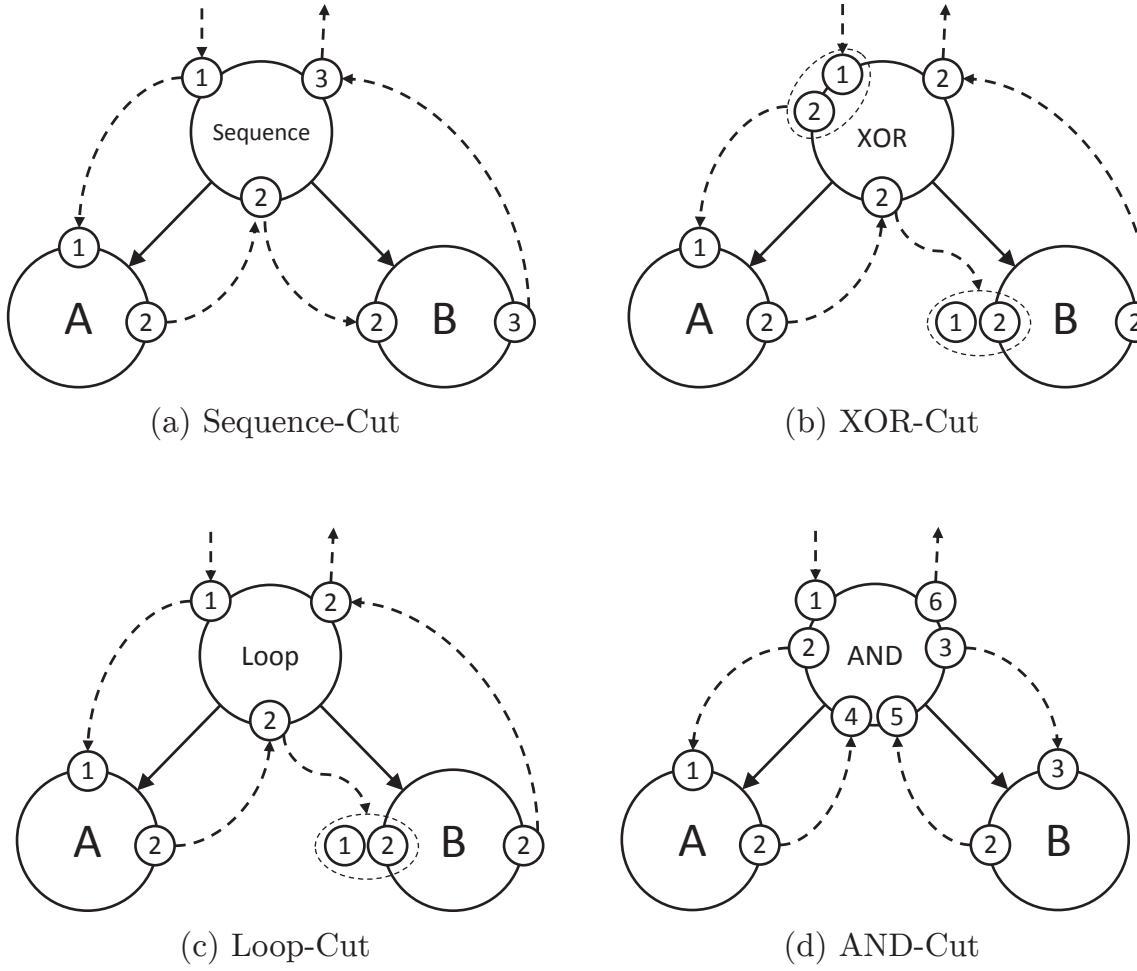
Petrinetz-Ausgabe

Eine rekursive Funktion baut das Petrinetz abhängig vom jeweiligen Prozessmodell-Operator auf. Hierfür wird ein Incoming- und/oder Outgoing-Place an die Leaf-Knoten weitergegeben. Diese liefern ihren Outgoing-Place zurück und fügen die Transitionen/Teilnetze im Prozessmodell ein. Mit Hilfe der Abbildung 5.3 wird das Vorgehen verdeutlicht.

- *Leaf*: Aus einem Leaf wird immer eine Transition erzeugt. Der Incoming-Place der Transition ist der Place, den das Leaf erhalten hat. Es wird ein neuer Place erzeugt, der zum einen der Outgoing-Place der Transition ist und zum anderen an die aufrufende Methode zurückgegeben wird.
- *Sequence*: (Abbildung 5.3 a) Der übergebende Place (1) wird als Start-Place für den linken Zweig weitergereicht. Der End-Place des linken Zweigs (2) wird als Start-Place an den rechten Zweig weitergegeben. Somit ist sichergestellt, dass der rechte Zweig auf den linken Zweig folgt. Die Sequence gibt den End-Place vom rechten Zweig (3) zurück.
- *Exclusive Choice*: (Abbildung 5.3 b) Der Start des linken und rechten Teilbaums ist identisch (1). Ebenso ist das Ende des linken und rechten Teilbaums identisch (2).
- *Loop*: (Abbildung 5.3 c) Der übergebene Place von Loop (1) wird an den linken Teilbaum als Startplace und den rechten Teilbaum als Endplace übergeben. Zudem ist das Ende des linken Teilbaums das Ende des rechten Teilbaums (2).
- *Parallel*: (Abbildung 5.3 d) Der übergebene Place (1) verweist auf eine neu erstellte And-Split-Transition. Diese erzeugt zwei neue Outgoing-Places. Eine für den linken (2) und eine für den

rechten (3) Teilbaum. Die And-Places des linken (4) und des rechten (5) Teilbaums zeigen wiederum auf eine neue And-Join-Transition. Als Ergebnis wird der End-Place (6) des And-Joins zurückgeben.

Im Anhang ist unter 10.3 der Quellcode für den Durchlauf durch den Prozessbaum, sowie der Quellcode zum Zeichnen der Transitionen 10.4 dargestellt.



(a) Sequence-Cut (b) XOR-Cut

(c) Loop-Cut (d) AND-Cut

Abbildung 5.3: Inductive Miner Petrinetz-Ausgabe

5.6 Ergebnisdarstellung

In diesem Abschnitt wird beschrieben, wie die Ergebnisdarstellung implementiert wurde (Anforderung F9). Zunächst wird auf die Generierung der Petrinetze, welche anschließend in der Prozessmodell-Darstellung verwendet wird, eingegangen. Anschließend wird die Möglichkeit des Anzeigens von Prozessmodell-Informationen zum ausgewählten Prozessmodell beschrieben. Daraufhin folgt die Beschreibung der Consolidation mit den implementierten Auswahlkriterien. Weiterhin wird der Prozessmodellvergleich und der Export von Prozessmodellen erläutert.

5.6.1 Petrinetze

Da die drei implementierten Mining-Algorithmen alle Petrinetze erzeugen, stellen diese eine wichtige Grundlage der Software dar (siehe Abschnitt 2.2 und 4.1). Die Abbildung 4.3 zeigt, dass zum einen die *PetiNet*-Klasse eine Liste aller *Transition*- und *Place*-Objekte speichert, zum anderen diese Objekte aber auch selbst auf ihre Nachfolger verweisen (in *OutgoingPlaces* bzw. *OutgoingTransitions*). Diese Verknüpfung macht sich der Algorithmus zum Zeichnen der Netze zu Nutzen.

Im ersten Schritt geht die *ColumnBuilder*-Klasse das Netz vom Anfang an durch und erstellt Spalten mit abwechselnd *Place*- und *Transition*-Objekten. Dies wird durch den abwechselnden Aufruf der Methoden *AddPlace()* und *AddTransition()* erreicht. Hierbei müssen einige Spezialfälle beachtet werden: Loops müssen erkannt werden, damit der Algorithmus nicht unendlich den selben Pfad zu zeichnen versucht. Außerdem wird versucht, besondere Transitionen wie AND-Splits und -Joins zu gruppieren.

Für jede Spalte im Netz wird ein *Column*-Objekt erzeugt, das neben der Spaltennummer eine beliebige Anzahl an *Node*-Objekten vorhalten kann. Am Ende werden alle *Column*-Objekte in einer Liste zurückgegeben. Die horizontale Verteilung der Objekte ist damit abgeschlossen.

Die Liste von *Column*s wird nun an den *RowBuilder* gegeben, der die vertikale Reihenfolge der Nodes bestimmt. Die Liste wird dabei drei Mal durchlaufen: Beim ersten Mal werden Nodes mit mehreren Nachbarn an diesen ausgerichtet. Danach werden die restlichen einzelnen Nodes aneinander ausgerichtet, zunächst von links nach rechts und im dritten Durchlauf von rechts nach links.

Nun kann in der *PetriNetVisualizer*-Klasse aus der Spaltennummer und dem eben berechneten *Row*-Attribut eine Koordinate berechnet werden. Über die *DrawNode()*-Funktion werden die Nodes nun entsprechend auf dem *Canvas*-Objekt gezeichnet. Hinterher werden alle Nodes erneut durchgegangen und die Verbindungspfeile mit *ConnectByArrow()* gezeichnet. Diese werden per *Binding* an die Nodes gebunden, damit sie beim Verschieben der Nodes automatisch korrekt platziert werden.

5.6.2 Prozessmodell-Darstellung

Nach dem Minen der zuvor ausgewählten Daten, gelangt der Benutzer auf die `result-Seite`. Hier werden alle Ergebnisse in der Matrix-Selection dargestellt. Durch einen Doppelklick auf das gewünschte Field, innerhalb der Matrix-Selection, wird das `Button-Click`-Event *FieldDoubleClick(object sender, MouseButtonEventArgs e)* in der `P8results`-Klasse ausgelöst. Hier wird wiederum die Methode *PMViewer()* in der partiellen Klasse `PMViewer` genutzt, um ein neues Fenster zu öffnen und dort das Prozessmodell darzustellen. Abbildung 5.4 zeigt ein Prozessmodell in dem Fenster, das durch die Klasse `PMViewer` aufgebaut wurde.

5.6.3 Prozessmodell-Informationen

Auf der Seite *results* können zu jedem Prozessmodell deren verfügbare Informationen angezeigt werden (Anforderung F14). Die Informationen werden eingeblendet, sobald in der Matrix-Selection auf ein Field geklickt wird. Die Abbildung 5.5 zeigt die Auswahl eines Fields (1. Punkt) und die dazugehörigen Prozessmodell-Informationen (2. Punkt).

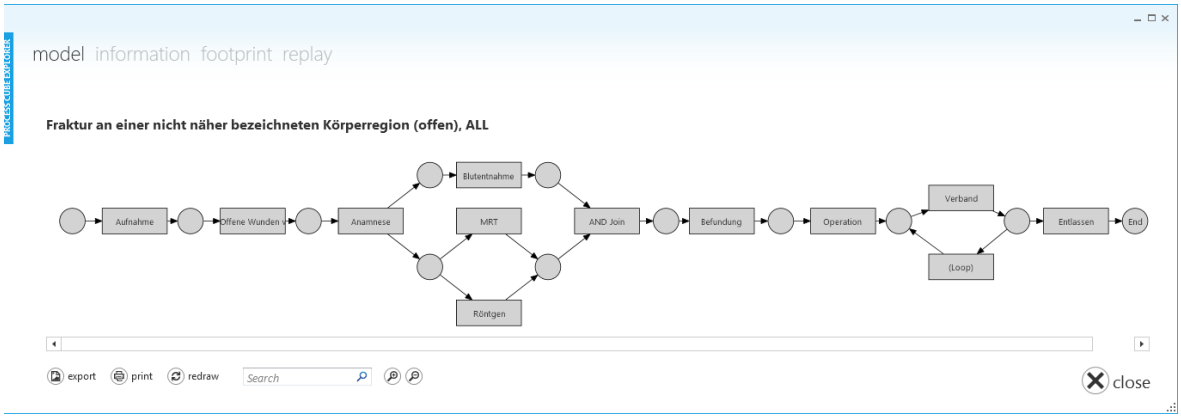


Abbildung 5.4: Prozessmodell-Darstellung

In den Fields werden je nach Zustand des dahinter liegendem Prozessmodell drei unterschiedliche Punkte in den Farben rot, gelb oder grün angezeigt. Ein grüner Punkt neben der Dimensionsbeschriftung gibt an, dass die Anzahl der verwendeten Events über dem vom Benutzer definierten Schwellwert liegt. Ein gelb markiertes Field bedeutet, dass die Anzahl der verwendeten Events unter dem Schwellenwert liegt. Konnte der Mining-Algorithmus für eine Dimensionsausprägung kein Prozessmodell erzeugen wird das entsprechende Field mit einem roten Punkt markiert.

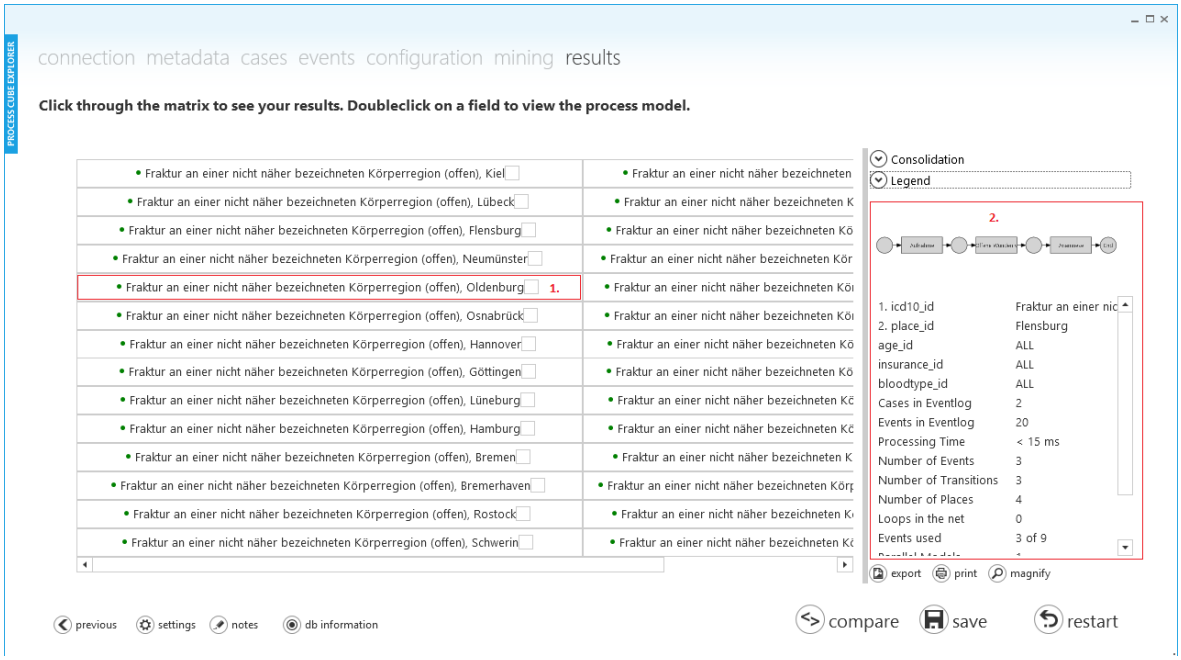


Abbildung 5.5: Informationen zum ausgewählten Prozessmodell “Fraktur an einer nicht näher bezeichneten Körperregion (offen), Oldenburg“

In der Abbildung 5.5 wurde das Field *Fraktur an einer nicht näher bezeichneten Körperregion (offen), Oldenburg* (1. Punkt) ausgewählt. Die zu diesem Prozessmodell dazugehörigen Informationen werden rechts (2. Punkt) angezeigt:

- Eine *Vorschau* des Prozessmodells.
- Die ausgewählten *Dimensionen* und *Filter*.
- *Cases in Eventlog* gibt die Anzahl des Cases in der Datenbank, die zu diesem Event-Log gehören, an.
- *Events in Eventlog* gibt die Anzahl aller Events (inklusive Duplikaten) im Event-Log an.

Die Mining-Algorithmen erstellen außerdem abhängig von ihrer Implementierung noch weitere Informationen, unter anderem:

- *Processing Time* ist die benötigte Dauer zur Berechnung des Prozessmodells.
- *Number of Events* besagt, wie viele einzigartige Events im Prozessmodell vorkommen.
- *Number of Transitions* gibt die Anzahl der Transitionen (ohne Loops und AND-Splits und Joins) im Prozessmodell an.
- *Number of Places* gibt die Anzahl der Places im Prozessmodell an.
- *Loops in net* gibt die Anzahl der Loops im Prozessmodell an.
- *Events used* gibt an, wie viele der möglichen (einzigartigen) Events tatsächlich im Prozessmodell vorkommen.
- *Parallel Models* erscheint, wenn der Heuristic Miner mehrere Startevents findet und daher mehrere Modelle miteinander verknüpft. Hier wird die Anzahl der parallelen Modelle angegeben.
- *Minimal Adjacency* gibt im Heuristic Miner an, wie groß die kleinste im Modell vorkommende Adjazenz (siehe Abschnitt 2.3.2) ist.
- *Average Adjancency* gibt die durschnittliche Adjazenz an.
- *Standard Deviation* gibt die Standardabweichung der Adjazenz an.

In der Implementierung verläuft es sich wie folgt: Nach einem Klick auf das jeweilige Field wird ein *Button-Click-Event* ausgelöst und die Methode *FieldClick(object sender, RoutedEventArgs e)* in der *P8results-Klasse* aufgerufen. In dieser Methode werden die gespeicherten Prozessmodell-Informationen, die zum angeklickten Field gehören, aus einem Dictionary mit dem Namen *Information* ausgelesen. Das Dictionary befindet sich in der *Field-Klasse*, das zu jedem Field die gesammelten Informationen bereit hält. Die Informationen werden unter anderem während des Aufrufs der Mining-Algorithmen in das Dictionary geschrieben.

5.6.4 Consolidation

Die Consolidation (deutsch: Konsolidierung) (Anforderung F22) ist im Kontext der OLAP-Operatoren das Zusammenfassen großer Datenmengen zu einem Datensatz. Für die entwickelte Software bedeutet das, dass sowohl die Daten innerhalb einer Dimension oder dimensionsübergreifend, als auch das mehrere Prozessmodelle zu einem Prozessmodell zusammengefasst werden. Für die aktuelle Software einigte sich die Projektgruppe darauf, die Anzahl der dargestellten Prozessmodelle anhand der in diesem Abschnitt beschriebenen Auswahlkriterien einzugrenzen.

Die Abbildung 5.6 zeigt die Einstellungen und das Ergebnis nach der Ausführung einer Consolidation. Auf der *results-Seite* ist das Menü *Consolidation* (1. Punkt) zu finden. Dort hat man die Möglichkeit, bestimmte Kriterien so auszuwählen, das nur die Prozessmodelle in der Matrix-Selection (5. Punkt) angezeigt werden, die diese Kriterien erfüllen. Die folgenden Kriterien stehen zur Auswahl:

- *Loop:* Es werden Prozessmodelle gesucht, die Loops enthalten.
- *Parallelism:* Es werden Prozessmodelle gesucht, die Parallelitäten enthalten.
- *Events:* Es werden Prozessmodelle gesucht, die die ausgewählten Events enthalten.
- *Min. Number of Events:* Es werden Prozessmodelle gesucht, die mindestens die ausgewählte Anzahl von Events enthalten.
- *OR-/AND-Operator:* Ist der OR-Operator ausgewählt, werden alle Prozessmodelle angezeigt, die eines oder mehrere der vorher ausgewählten Kriterien erfüllt. Wird der AND-Operator ausgewählt, werden nur die Prozessmodelle angezeigt, die alle ausgewählten Kriterien erfüllen.

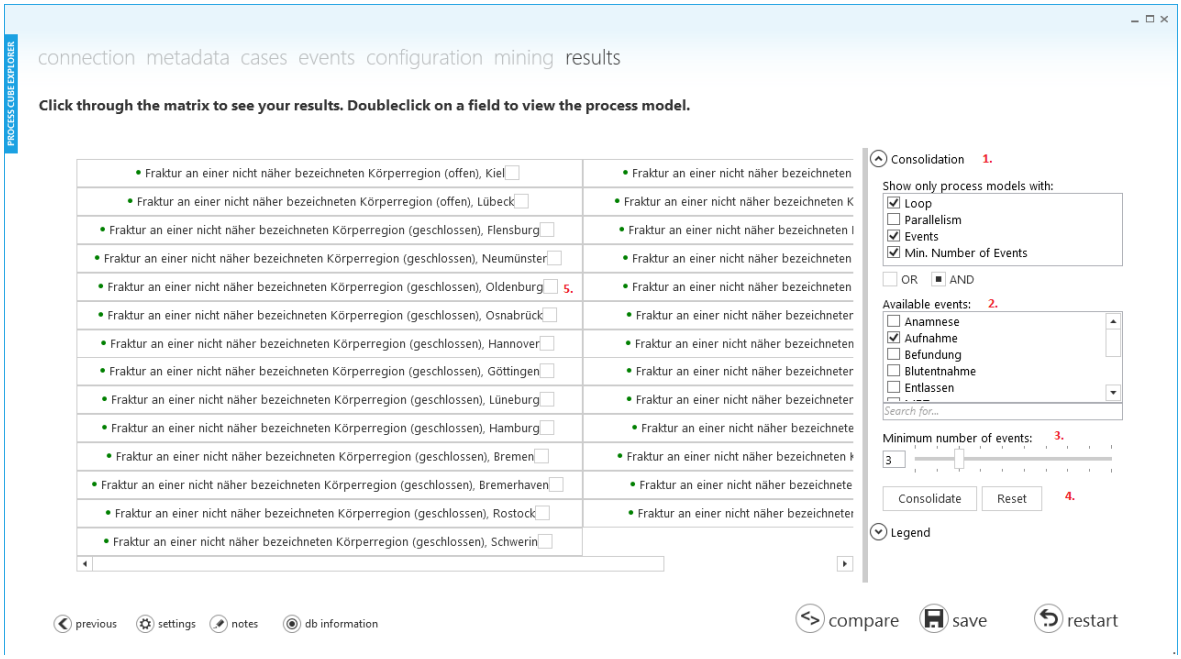


Abbildung 5.6: Ergebnis nach einer ausgeführten Consolidation

Abbildung 5.6 zeigt, dass die Kriterien *Loop, Events, Min. Number of Events und der OR-Operator* ausgewählt wurden. Da das Kriterium *Events* ausgewählt wurde, erscheint eine ListView (2. Punkt). Dort können die Events ausgewählt werden, die in den Prozessmodellen vorhanden sein müssen (In der Abbildung 5.6 wurde das Event *Aufnahme* ausgewählt). Direkt unter der ListView gibt es ein QuickSearch-Feld, mit dem die ListView nach Events durchsucht werden kann. Außerdem wurde das Kriterium *Min. Number of Events* ausgewählt. Dadurch wird ein Slider eingeblendet (3. Punkt). Dort kann geregelt werden, welche Anzahl an Events die Prozessmodelle beinhalten müssen (in der Abbildung 5.6 wurde der Slider auf zwei eingestellt). Nach einem Klick auf den *Consolidate*-Button (4. Punkt), wird das *Button-Click*-Event *StartConsolidationClick(object sender, RoutedEventArgs e)* in der *P8results-Klasse* ausgelöst. Dort werden die ausgewählten Kriterien ausgelesen und in

einer Liste der *ConsolidatorSettings-Klasse* gespeichert. Die verschiedenen Consolidator-Algorithmen haben anschließend Zugriff auf die gespeicherten Kriterien. Das Ergebnis der Consolidation wird anschließend in der Matrix-Selection angezeigt (5. Punkt).

Um die Erweiterbarkeit der Consolidation zu gewährleisten, wurde das sogenannte Factory-Pattern implementiert. Dafür wurde zunächst das *iConsolidator*-Interface erzeugt. Dies enthält die Methode *Consolidate()*, die von den erbenden Klassen zu implementieren ist. Um ein Objekt einer Klasse zu erzeugen wird die Methode *CreateConsolidator()* der statischen *ConsolidationFactory-Klasse* aufgerufen. Diese Methode erzeugt ein Objekt *consolidator* der Klasse *StandardConsolidator*. Danach wird mit *consolidator.Consolidate()* der Konsolidierungsprozess für den *StandardConsolidator* gestartet.

Implementiert wurde die bereits genannte *StandardConsolidator-Klasse*, die die Funktionen der Loop- und Parallelitätenerkennung, sowie des Suchens nach ausgewählten und der minimalen Anzahl von Events enthält.

5.6.5 Prozessmodellvergleich

Der implementierte Prozessmodellvergleich (Anforderung F23) basiert auf der Theorie des *Snapshot-Algorithmus*. Dieser legt fest, dass eines der ausgewählten Prozessmodelle als Referenzmodell dient. Dieses ist im Process Cube Explorer das zuerst ausgewählte Prozessmodell. Weiter legt der Snapshot-Algorithmus fest, dass für jedes Prozessmodell eine Liste mit allen Knoten bestehen soll, anhand dieser die Differenzen ermittelt werden. Für die Ermittlung der hinzugefügten Knoten, sprich Knoten, die nicht im Referenzmodell, jedoch im anderen Prozessmodell enthalten sind, wird Listing 5.3 ausgeführt. Die ermittelten Knoten werden als *added* gekennzeichnet.

```
var added = listofTransitionsInSecondProcessModel.Except(
   listofTransitionsInFirstProcessModel, new TransitionComparer()).
   ToList();
```

Listing 5.3: Differenzberechnung - Hinzugefügte Knoten

Die Klasse *TransitionComparer* überprüft, ob zwei Transitionen (Knoten) denselben Namen haben. Es wird angenommen, dass Transitionen mit gleichen Namen identisch sind. Knoten, die im Referenzmodell, jedoch nicht im anderen Prozessmodell enthalten sind, werden als *deleted* gekennzeichnet.

Knoten, bei denen der Vorgänger- und Nachfolgerknoten im Prozessmodell gleich ist werden als *changed* gekennzeichnet und der Name des Knoten verändert. Listing 5.4 zeigt die Implementierung der veränderten Knoten.

```
for (var index = 0; index < restOfFirstModel.Count(); index++)
 if (!restOfFirstModel[index].Name.Equals(restOfSecondModel[index].
    Name))
  {
  restOfFirstModel[index].DiffStatus = DiffState.Changed;
  restOfFirstModel[index].Name = "(" + restOfFirstModel[index].Name
      + ") " + restOfSecondModel[index].Name;
  }
```

Listing 5.4: Differenzberechnung - Veränderte Knoten

Die gekennzeichneten Knoten werden abschließend im Referenzmodell verändert, bzw. bei *added*-Transitionen neu am entsprechenden Platz eingefügt und mit dem *PMViewer* dem Benutzer angezeigt.

Abbildung 5.7 zeigt ein Ergebnis nach der Ausführung der Differenzberechnung. Der Benutzer wählt dazu in der Result-Ansicht der fertig erstellten Prozessmodelle zwei Prozessmodelle mit der Checkbox aus und klickt auf *Compare*. Die Darstellung des Differenz-Prozessmodells erfolgt in demselben Fenster, wie für die gewöhnlichen Prozessmodelle, sodass alle Funktionen, bis auf das Conformance Checking sowie der Export im MXML-Format, genutzt werden können.

Die Transitionen (Punkt 1 und 2) *Blutentnahme, MRT* wurden als *changed* markiert. Der Grund dafür ist, das die beiden Transitionen im zweiten Prozessmodell vertauscht sind.

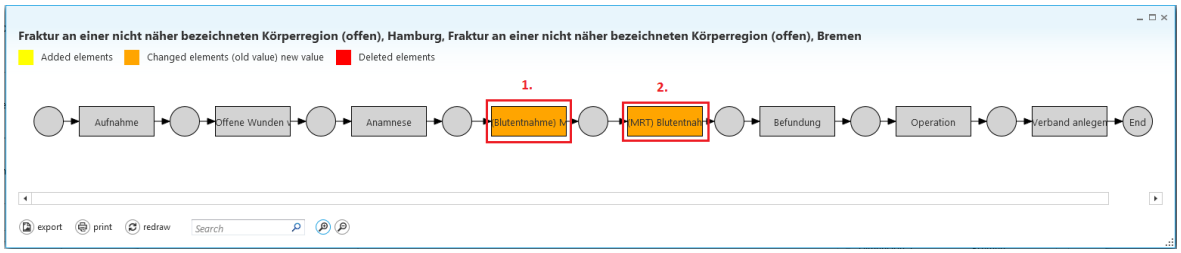


Abbildung 5.7: Prozessmodell resultierend aus der Differenzberechnung

5.6.6 Export

Auf der `result-Seite` ist es möglich, das ausgewählte Prozessmodell zu speichern. Dafür werden die Formate *.bmp, .gif, .jpeg, .png, .tiff, .dot, .pnml* und *.mxml* zur Verfügung gestellt. Bei dem Export als *.mxml* wird das Event Log aus dem das dargestellte Processmodel erstellt wurde exportiert. Die anderen Formate dienen zum Export des Processmodel. Dem Benutzer werden an verschiedenen Stellen die Möglichkeit zum Export des Prozessmodell angeboten, die Funktion ist jedoch immer dieselbe.

Durch einen Klick auf den *export*-Buttons, wird das *Button-Click-Event ExportClick(object sender, RoutedEventArgs e)* ausgelöst. Dort wird auf die Methode *Export()* der Klasse *Exporter* zugegriffen. In der Klasse werden mit der Methode *GetProperties()* Default-Parameter wie z. B. *ImageName, ImagePath* und der *ImageFiletype* aus der Klasse *Settings* ausgelesen. Diese Parameter sind für die anschließende Erzeugung des *SaveFileDialog*-Fensters notwendig. Sobald der Benutzer einen eigenen Pfad, Dateinamen und das Dateiformat gewählt und auf `Speichern` geklickt hat, werden diese Informationen in den entsprechenden Variablen gespeichert. Daraufhin kommt das implementierte Factory-Pattern für den Export zum Einsatz.

Um das Factory-Pattern umzusetzen, wurde das dafür notwendige Interface *IExporter*, die Factory-Klasse *ExporterFactory* und die eigentlichen Exporter-Klassen *ImageExporter*, *PNMLExporter* und *MXMLExporter* angelegt. Das Interface *IExporter* enthält die zu implementierende Methode *export(String filename)* für die erbenden Klassen. Nach dem Klick auf `Speichern` wird die Methode *createExporter(filetype, canvas, selectedField)* der Factory-Klasse *ExporterFactory* aufgerufen (Listing 5.5, 1. Zeile), um ein Objekt zu erzeugen, das abhängig vom ausgewählten Dateiformat ist. Wurde z. B. das *mxml*-Format gewählt, erzeugt die Klasse *ExporterFactory* ein Objekt der Klasse *MXMLExporter*. Dieses Objekt wird in der Variable *exporter* gespeichert. Danach wird die Methode

export(filename) des Objektes *exporter* aufgerufen (Listing 5.5, 2. Zeile) und das Prozessmodell gespeichert. Bei einer erfolgreichen Speicherung, liefert die Methode *export(filename)* ein *true* zurück. Erfolgt das Speichern korrekt, wird automatisch der Datei-Explorer geöffnet und in den Pfad gesprungen, der beim Speichervorgang ausgewählt wurde.

```
IExporter exporter = ExporterFactory.createExporter(filetype,
   canvas, selectedField);
bool success = exporter.export(filename);
```

Listing 5.5: Exporter-Erzeugung durch das Factory-Pattern

5.7 Conformance Checking

Die Software besitzt eine Komponente, mit der ein *Conformance Checking* durchgeführt werden kann (Anforderung F19). Um in der Software zu dem Bereich zu gelangen, kann auf ein *Field* doppelt geklickt werden. Darauf öffnet sich ein Fenster in dem es mehrere Bereiche gibt (siehe Abbildung 5.8). Es wurden die beiden Verfahren *Comparing Footprint* und *Token Replay* umgesetzt. Die beiden Verfahren sind in der *PMViewer-Ansicht* jeweils unter dem Bereich *footprint* und *replay* zu finden. Die Konzepte dahinter werden in den Kapiteln 5.7.1 und 5.7.2 genauer erläutert. Mit den beiden Verfahren lässt sich die Fitness berechnen, die eine Aussage über die Konformität gibt und es werden Informationen bereit gestellt mit denen bei Konformitätsproblemen die problematischen Stellen gefunden werden können. Zum einen werden dafür die betroffenen Stellen im Petrinetz markiert und zum anderen können die Informationen über Footprints abgelesen werden.

5.7.1 Comparing Footprint

In dem Abschnitt 2.4.1 wurden die Grundlagen vom *Comparing Footprint* Verfahren und die Symbole #, $\leftarrow$, $\rightarrow$ und || vorgestellt. Während der Implementierung des Verfahrens nach den theoretischen Ansätzen aus [Aal11] ist die Erkenntnis gekommen die Aussagekraft der Footprints durch weitere Symbole zu erweitern. Dafür wurden die beiden Symbole @ und X eingeführt. Das @-Symbol steht für eine Loop von einer Transition. Das X-Symbol hat eine ähnliche Bedeutung wie das #-Symbol. Während das #-Symbol bedeutet, dass eine Beziehung zwischen zwei Transitionen nicht vorhanden ist und damit wissentlich nicht vorhanden ist, bedeutet das X-Zeichen das keine Aussage gemacht werden kann, ob eine Beziehung zwischen zwei Transitionen vorhanden ist oder nicht. Das kann vorkommen, wenn zwei Footprints miteinander verglichen werden, bei denen es keine gemeinsame Schnittmenge von ein und den selben Transitionen gibt. Da in der Software ein Conformance Checking bisher nur zwischen einem Event Log und dem daraus generierten Petrinetz gemacht werden kann, kommt es nicht vor, dass das X-Symbol eine Verwendung findet. Für den Fall das die Software erweitert wird und ein Conformance Checking auch zwischen mehreren Event Logs oder zwischen mehreren Prozessmodellen durchgeführt werden soll, wird empfohlen dieses Symbol zu verwenden. Da hierbei mehr Informationen über die Konformität aus den Footprints abgelesen werden kann.

In der Comparing Footprint Ansicht (Abbildung 5.8) befinden sich zwei Unterbereiche *Conformance* und *Footprints*. In der Conformance-Ansicht kann oben die Fitness abgelesen werden, die mit dem Comparing Footprint Verfahren ermittelt wurde (Anforderung F18). Wie bereits im Abschnitt 2.3.5 in

dem Unterpunkt *Fitness* erwähnt wurde, kann die Fitness eine unterschiedliche Bedeutung haben. Die Definition der Fitness lehnt sich bei diesem Verfahren an die Event-Ebene.

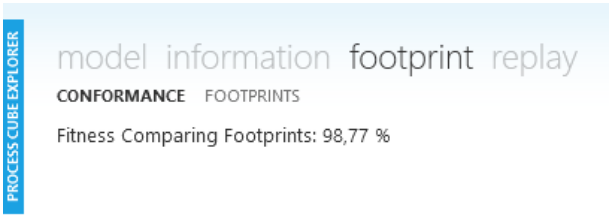


Abbildung 5.8: Comparing Footprint - Fitness

In dem Beispiel, das in diesem Abschnitt behandelt wird, wurde eine Fitness von 98,77% berechnet. In der gleichen Ansicht unterhalb der Fitness ist das Petrinetz dargestellt, dass mit dem Comparing Footprint Verfahren überprüft wurde. Von diesem Petrinetz ist ein Ausschnitt in der Abbildung 5.9 dargestellt. Eine Legende in der oberen rechten Ecke gibt an, welche Bedeutung die unterschiedlich farbig markierten Transitionen haben. Zwei Pfeile verdeutlichen die Bedeutung von zwei Transitionen. Während bei der Transition *Operation* mit der Konformität zwischen dem Petrinetz und dem Event Log alles in Ordnung ist, gibt es ein Konformitätsproblem bei der Transition *Verband*.

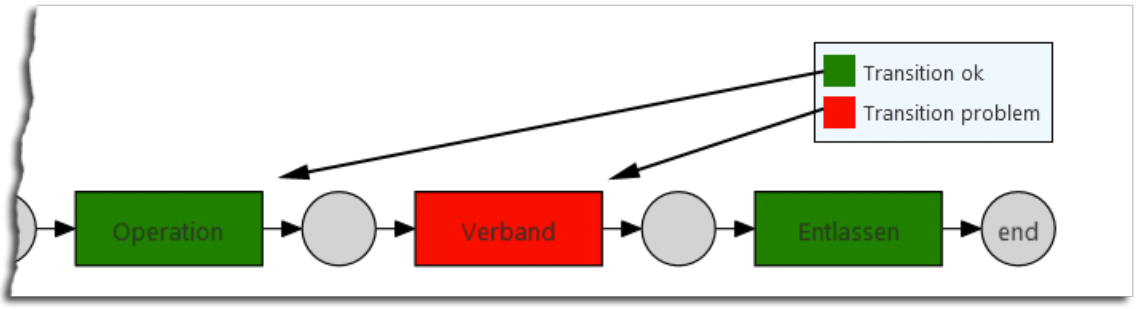


Abbildung 5.9: Comparing Footprint - Transition Problem

Um nun mehr Informationen über das Konformitätsproblem zu bekommen, muss zu der erwähnten Unteransicht *Footprints* gewechselt werden. Auf dieser Ansicht befinden sich drei Footprints. Ein Footprint von dem Event Log (Tabelle 9.1), Petrinetz (Tabelle 9.2) und ein Footprint, dass die Unterschiede zwischen den ersten beiden Footprints darstellt (Tabelle 5.1). In dem Footprint, das die Unterschiede anzeigt, kann der Grund für das Konformitätsproblem abgelesen werden. Auf diesem befindet sich nur ein einziger Eintrag. In der Spalte und Zeile *Verband* steht in der entsprechenden Zelle @:#. Das bedeutet das laut Event Log die Transition *Verband* sich wie eine Loop verhält (@), aber in dem Petrinetz dies nicht dargestellt ist (#). Es kann unterschiedliche Gründe dafür geben. Bei diesem Beispiel wurde der Alpha-Miner verwendet, um aus dem Event Log ein Petrinetz zu generieren. Der Alpha Miner ist jedoch nicht in der Lage Loops von einzelnen Transitionen zu erkennen. Dafür müsste z. B. der Alpha Miner++ oder der Heuristic Miner verwendet werden.

Bei der Verwendung eines Mining-Algorithmus, der Loops von Transitionen erkennen kann, würde der entsprechende Ausschnitt von dem Petrinetz wie in der Abbildung 5.10 aussehen. Daraus ergibt sich eine Fitness von 100%. Daher gibt es keinen Unterschied zwischen dem Event Log und dem daraus generierten Petrinetz. Somit sind sämtliche Cases aus dem Event Log auf das Petrinetz abspielbar.

	Aufnahme	Anamnese	Blutentnahme	Röntgen	MRT	Befundung	Operation	Verband	Entlassen
Aufnahme									
Anamnese									
Blutentnahme									
Röntgen									
MRT									
Befundung									
Operation									
Verband								@:#	
Entlassen									

Tabelle 5.1: Footprint von dem Vergleich zwischen Event Log und Petrinetz

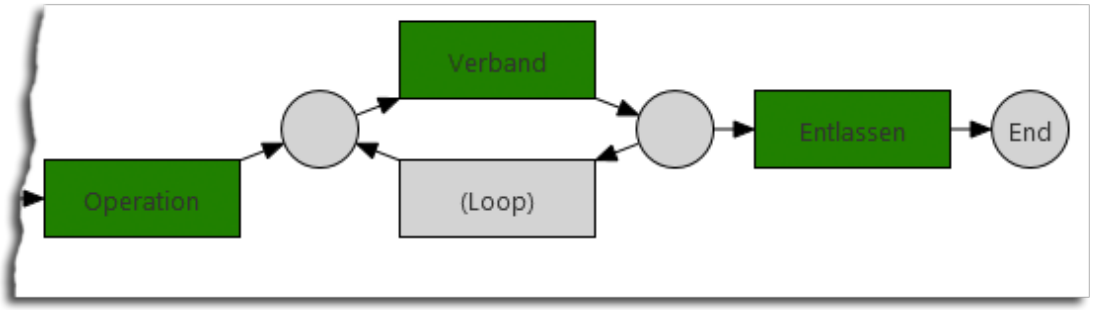


Abbildung 5.10: Loop erkannt

Das Comparing Footprint Verfahren befindet sich in dem Projekt *ConformanceChecking*. Dazu gehören die Klassen *ComparingFootprintAlgorithm*, *ComparingFootprint*, *ComparingFootprintResultMatrix*, *ComparingFootprintResultMatrixCell* und *CellType*. In der Klasse *ComparingFootprintAlgorithm* befindet sich die Logik für die Erstellung eines Footprints. Ein Footprint wird durch die Klasse *ComparingFootprint* abgebildet. Wenn zwei Footprints miteinander verglichen werden, dann kann dieser Ergebnis-Footprint durch die Klasse *ComparingFootprintResultMatrix* abgebildet werden. Bei der Umsetzung des Verfahrens wurden als erstes unterschiedliche Methoden für die Erstellung von Footprints implementiert. Die Methode *CreateFootprint(EventLog eventLog)* erstellt einen Footprint aus einem Event Log und liefert ein Objekt des Typs *ComparingFootprint* zurück, daher einen Footprint. Dafür werden als erstes von jedem Case aus dem Event Log einzelne Footprints erstellt und in einer Liste zwischengespeichert. Um einen Footprint von einem Case zu erstellen wird ein neues Objekt vom Typ `ComparingFootprint` angelegt. In diesem werden zunächst alle Namen von den unterschiedlichen Events in die Liste `HeaderWithEventNames` eingetragen. Danach wird ein zweidimensionales Array `ResultMatrix` mit der Größe von der Anzahl der Event-Namen jeweils für jede Dimension initialisiert. In einer zweifach verschachtelten Schleife wird das Array `ResultMatrix` durchgegangen. Die äußere Schleife zählt den Index für die Spalte und die innere Schleife zählt den Index für die Zeile. Dieses Konstrukt geht Zelle für Zelle von dem zu erstellenden Footprint durch. In der inneren (zweiten) Schleife gibt es eine dritte Schleife, die sämtliche Events von dem aktuellen Case behandelt. Wenn dabei gefunden wurde, das in der aktuellen Zelle eine Transition auf die andere folgt, dann wird ein Rechtspfeil zu der Zelle zugewiesen und die Überprüfung geht weiter. Bei der fort laufenden Überprüfung könnte nun festgestellt werden, dass die beiden Transitionen ebenfalls von der anderen Richtung folgen, daher ein Linkspfeil. Da in der Ergebniszelle jedoch bereits ein Rechtspfeil steht, resultiert daraus eine Parallelität und in die Zelle wird daher der Zellentyp Parallelität (||) eingetragen. Die Erstellung eines Footprints von einem Petrinetz erfolgt sehr ähnlich. Dabei wird

anstatt, dem überprüfen der aufeinander folgenden Events aus den Cases, das Petrinetz durchlaufen und so die Zusammenhänge der Events ermittelt und in die Zellen der Footprints eingetragen.

Danach wird diese Liste mit den Footprints der Methode `MergeFootprints(List<Comparing Footprint> footprintList)` übergeben und die einzelnen Footprints werden zu einem einzigen zusammen geführt. Beim Zusammenführen von mehreren Footprints wird in einer Schleife die Liste mit den Footprints Zelle für Zelle durch gegangen. Dabei wird in jeder Zelle gezählt wie oft welcher Zellentyp in den Footprints des Cases vorkommt. Die einzelnen Symbole haben eine unterschiedliche Gewichtung, diese ist in der Tabelle 5.2 aufgelistet. Wenn nun also nur ein einziges Symbol in der Zelle von den Footprints mit einer höheren Gewichtung vorkommt als ein Symbol mit einer niedrigeren Gewichtung, dann ist das Symbol mit der höheren Gewichtung ausschlaggebend. Daher wird bei drei Footprints, in denen in der ersten Zelle einmal eine #, ein $\rightarrow$ und das ||-Symbol vorkommt, in dem zusammengeführten Footprint das ||-Symbol in die Zelle eingetragen. Falls bei einer anderen Zelle der Pfeil nach links und der Pfeil nach rechts vorkommt, dann kann daraus zurück geschlossen werden, dass der Zellentyp eine Parallelität darstellt. Daher wird darauf aus den beiden Pfeilen das ||-Symbol. Das ||-Symbol kann nur bei Transitionen vorkommen die unterschiedlich sind. Also bei einer Zelle bei der das Verhältnis von ein und der selben Transition dargestellt wird z. B. *Verband* und *Verband*, kann es bei dieser Zelle entweder kein Verhältnis (#) oder eine Loop (@) geben.

Um nun zwei Footprints miteinander zu vergleichen, also z. B. dem Footprint von einem Event Log und dem eines Petrinetzes wird ein Objekt der Klasse `ComparingFootprintResultMatrix` instantiiert. Als Parameter erwartet der Konstruktor zwei Objekte vom Typ `ComparingFootprint`. Wenn Unterschiede zwischen den Zellen festgestellt wurden, dann wird dieser Unterschied (z. B. #:||) in die Zelle des Ergebnis-Footprints eingetragen.

Für die Berechnung der Fitness wird die Anzahl der Zellen mit Unterschieden, sowie die Anzahl der Zellen aus dem Ergebnis-Footprint gezählt und der Methode `CalculateFitness(int numDifferences, int numOpportunities)` übergeben. Diese liefert den Wert der Fitness zurück.

Symbol	Gewichtung
X	0
#	1
$\leftarrow$	2
$\rightarrow$	2
||	3
@	3

Tabelle 5.2: Gewichtung der Footprint Zellen Symbole

5.7.2 Token Replay

In der Abbildung 5.11 ist die Token Replay Ansicht zu sehen. Dabei wurde das gleiche Beispiel analysiert, mit dem gleichen Problem wie in der Abbildung 5.9. Bei *Successful Replays* steht die Anzahl an erfolgreich abgespielten Cases auf das Petrinetz. Unter *Success Rate* steht die Fitness, die mit dem Token Replay Verfahren berechnet wurde. Wie bereits im Kapitel 2.3.5 in dem Unterpunkt

Fitness erwähnt wurde, kann die Fitness eine unterschiedliche Bedeutung haben. Die Definition der Fitness lehnt sich bei diesem Verfahren an die Case-Ebene.

Unter dem Punkt *Transitions Not Found* werden sämtliche Cases aufgelistet, bei denen eine Transition nicht abgefeuert werden konnte, da die Transition nicht gefunden wurde. Bei *Transitions Not Enabled* werden die Cases aufgelistet, die aufgrund einer falsch dargestellten Reihenfolge nicht abgefeuert werden konnten. Da in dem Petrinetz aus dem Beispiel (Abbildung 5.9) die Transition *Verband* nicht als Loop dargestellt wurde, können die Cases nicht richtig auf das Petrinetz abgespielt werden. In jedem dieser Cases wird das Event *Verband* mehrmals hintereinander aufgerufen, daher müsste dieses Event als Loop im Petrinetz dargestellt werden. Bei einer richtig dargestellten Loop, wie bei dem Beispiel aus der Abbildung 5.10, sieht die Token Replay Ansicht wie in der Abbildung 9.1 aus.

Abbildung 5.11: Token Replay - Fitness

5.8 Qualitätssicherung

Die Funktionsfähigkeit und Korrektheit des Programms wurden in umfangreichen Tests überprüft. Im Verlauf der Entwicklung wurden Komponententests (Unit-Tests) zum Testen der verschiedenen Klassen und ihrer Methoden erstellt. Diese bieten die Basis zur Ermittlung von Fehlern im Rahmen der Implementierung. Die Ergebnisse der Tests wurden bei Bedarf protokolliert und etwaige Bugs im Jira als Bug-Ticket aufgenommen. Hierin sind Art, Schwere, Stelle des Fehlers sowie Reporter und Priorität festgehalten. Fehler wurden im Rahmen der weiteren Entwicklung analysiert, bearbeitet und behoben. In diesem Kapitel werden die eingesetzten Maßnahmen zur Qualitätssicherung beschrieben.

5.8.1 Unit-Tests

Während der Entwicklung wurden Komponententests (Unit-Tests) zum Prüfen der korrekten Funktionsfähigkeit von Klassen und Methoden erstellt (Anforderung F25). Diese stellten die Basis zur Ermittlung von Fehlern im Rahmen der Implementierung dar. Für alle Teilprojekte (mit Ausnahme *Example Data*) wurden Unit-Tests erstellt. Je nach Anforderung an den Test wurden simple oder komplexe Prüfmethoden erstellt. Die Methode `public void CreateMinerTest()` (siehe Programmcode 5.6) prüft beispielsweise die Funktion der `MinerFactory`. Hierzu werden Strings mit Minernamen an die Factory übergeben. Die Assert-Methode vergleicht im Anschluss den Typ der zurückgelieferten Minerinstanz mit dem erwarteten Objekt-Typ.

```
[TestMethod]
public void CreateMinerTest()
{
  var field = new Field();
  var miner = MinerFactory.CreateMiner("Alpha Miner", field);
  Assert.IsInstanceOfType(miner, typeof(AlphaMiner));
}
```

Listing 5.6: public void CreateMinerTest())

Als Teil der Entwicklung wurden Tests teilweise von Code-Verantwortlichen und Code-Fremden Teammitgliedern erstellt. So wurde ermöglicht, erwartete Ergebnisse von tatsächlich gelieferten Ergebnissen zu unterscheiden. In der Regel wurden vor jedem Commit die Tests ausgeführt um negative Wechselwirkungen zwischen neu entwickelten Funktionen und bestehendem Code zu identifizieren. Hierdurch konnten Probleme im Code leichter identifiziert werden.

Insgesamt wurden (Stand 30.03.2014) 284 Unit-Tests implementiert und somit für die Relevanten Pakete(exclusive MainV2) eine Testabdeckung von 61,07 % erreicht. Die Abbildung 5.12 zeigt eine Coverage Übersicht der durchlaufenen Codeblocks. Im Anhang befindet sich eine weiter aufgeschlüsselte Übersicht 8.1

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Bernd_BN-MOBIL 2014-03-30 21_03_46.coverage	2903	38,95 %	4550	61,05 %
conformancechecking.dll	148	18,66 %	645	81,34 %
consolidationalgorithm.dll	26	11,66 %	197	88,34 %
database.dll	636	33,60 %	1257	66,40 %
diff.dll	136	54,18 %	115	45,82 %
matrixselection.dll	335	43,73 %	431	56,27 %
miningalgorithm.dll	1497	59,38 %	1024	40,62 %
processmodel.dll	125	12,43 %	881	87,57 %

Abbildung 5.12: Code Coverage Results

5.8.2 User-Tests

Für die Gewährleistung der Funktionsfähigkeit der Mining-Algorithmen, wurde die Software *Process Log Generator*[6] (PLG) verwendet, diese wird unter [BS13] beschrieben. Mit dem PLG wurden größere

[6] Process Log Generator - `http://www.processmining.it/sw/plg`

Petrinetze erstellt, woraus anschließend ein Event Log exportiert werden konnte. Einige solcher Event Logs wurden in einer Testdatenbank eingefügt und anschließend wurde manuell überprüft ob die Mining-Algorithmen aus den Event Logs ebenfalls die Petrinetze generieren konnten, wie sie Ursprünglich in dem Process Log Generator abgebildet waren. Falls dies nicht der Fall war, wurden die Mining-Algorithmen solange verbessert bis diese mit den unterschiedlichen Event Logs umgehen konnten.

6 Vorgehen im Projekt

Die Projektzeit begann zunächst mit einer Seminarphase, um die notwendigen Grundlagen für das Thema Multidimensionales Process Mining zu erarbeiten. Dafür wurden die folgenden Themen auf die einzelnen Projektmitglieder verteilt:

- Übersicht und Vergleich von Prozessmodellen
- Data-Warehouse und OLAP
- Event Cube
- Grundlagen Process Mining
- Alpha Algorithmus und Fuzzy Mining
- Heuristic Miner und genetisches Process Mining
- Weitere Process Mining Perspektiven
- Conformance Checking
- ProM-Framework
- Anwendungsgebiete von Process Mining
- Versorgungsforschung und Versorgungsprozesse
- Scrum

Zu jedem Thema wurde ein zwölfseitiges Paper erstellt und zum Abschluss der Seminarphase der Projektgruppe in einem Vortrag präsentiert.

Für die weitere Projektplanung, -abwicklung und -kontrolle wurde das agile Vorgehensmodell Scrum ausgewählt, welches im folgenden Abschnitt näher erläutert wird. Dadurch konnte auf eine große Planungsphase in der ersten Hälfte der Projektlaufzeit verzichtet werden. Vielmehr wurde so die schnelle Umsetzung eines einfachen und lauffähigen Prototyps realisiert.

Zu den verschiedenen Verantwortlichkeiten der Projektorganisation wurden Rollen definiert. Zudem wurde für die Bildung einer Arbeitsgrundlage die Projektgruppe zunächst in kleine Teams aufgeteilt. Deren Rollenver- und Teamaufteilung sowie die Aufgabenbereiche werden im Abschnitt 6.2 näher erläutert.

Dieses Kapitel schließt mit einer Beschreibung der Projektwerkzeuge ab. Es wird der Einsatz der webbasierten Anwendungen Confluence und Jira sowie der Versionsverwaltung Apache Subversion beschrieben.

6.1 Scrum

Scrum ist einer von mehreren existierenden agilen Prozessen für Softwareentwicklung und Projektmanagement – andere sind z. B. Crystal, Extreme Programming oder Feature Driven Development. Durch Scrum wird die Entwicklung einer Software im Team unterstützt und regelmäßige Projekt-Sitzungen ermöglichen allen Beteiligten einen aktuellen Informationsstand. In den folgenden Abschnitten wird erläutert, worum es sich bei Scrum handelt und wie es in der Projektgruppe eingesetzt wurde.

6.1.1 Agile Softwareentwicklung

Bei der agilen Softwareentwicklung handelt es sich um häufige Rückkopplungsprozesse und ein zyklisches Vorgehen auf allen Ebenen: bei der Programmierung, im Team und beim Management.

Anders als in der klassischen Vorgehensweise wird das neue System nicht im Voraus in allen Einzelheiten genau geplant und dann in einem einzigen langen Durchgang entwickelt. Das liegt darin begründet, dass sich die Anforderungen während der Projektlaufzeit noch ändern können und oft zu Beginn gar nicht vollständig bekannt sind.

Beim agilen Vorgehen wechseln sich stattdessen kurze Planungs- und Entwicklungsphasen ab. Im Anschluss an die Seminarphase des Projekts wurde der Projektgruppe die Idee des multidimensionalen Process Mining vorgestellt. Dadurch sollte ein Eindruck von der Thematik vermittelt werden, um Ziele und mögliche Anforderungen definieren zu können. Diese wurden in dem sogenannten Product Backlog (siehe Abschnitt 6.1.4) gesammelt. Zudem wurden kleine Teams zu den Themen Datenmodell, Schnittstellen/Interne Repräsentation und GUI/Funktionen (siehe Abschnitt 6.2)gebildet. Die Aufgabe der Teams bestand darin, in diesen Bereichen Ideen für einen Prototypen zu sammeln. Die Ergebnisse jedes Themenbereichs wurden in der großen Gruppe präsentiert und diskutiert. Aus diesen Ideen wurden schließlich Ziele für den Entwurf eines ersten Prototyps erarbeitet, besprochen und die Entwicklung begonnen.

6.1.2 Eigenschaften von Scrum

Das Vorgehensmodell Scrum soll die beteiligten Menschen in den Mittelpunkt stellen und dem sozialen Aspekt der Gruppenarbeit große Bedeutung einräumen. Kommunikation zwischen den Teammitgliedern bildet ein zentrales Element bei der Bearbeitung der zu erfüllenden Aufgaben [Sch].

Wöchentliche Sitzungen ermöglichten während der Projektlaufzeit einen regelmäßigen, persönlichen Kontakt zwischen den Teammitgliedern. Im Anschluss an die Sitzungen war ebenfalls Zeit, Gespräche in kleinen Gruppen zu führen, um einen gegenseitigen Austausch zu ermöglichen. Zusätzlich zu den wöchentlichen Sitzungen wurden ab der zweiten Projekthälfte Kernarbeitszeiten eingeführt. Innerhalb dieser Zeiten konnten sich die Teammitglieder in Gruppen zusammenfinden um gemeinsam Aufgaben zu lösen und das weitere Vorgehen zu besprechen.

Die regelmäßig stattgefundenen Sitzungen werden Scrums genannt. Aufgetretene Probleme und Fortschritte konnten so zeitnah besprochen werden. Große Softwareprojekte sind für den einzelnen Entwickler oft schwer zu überblicken. Scrums helfen dabei, Probleme frühzeitig zu erkennen.

Zudem wird das Projekt in kleinen Etappen bearbeitet. In jeder Etappe sind alle Entwicklungsschritte des Gesamtprojekts – Anforderungsanalyse, Design, Implementierung, Test – enthalten. Diese Etappen werden Sprints genannt und schriftlich festgehalten. Dadurch haben alle Teammitglieder die Möglichkeit sich über die Fortschritte der anderen Mitarbeiter zu informieren. Dies dient ebenfalls der Vorbeugung von Fehlern.

In diesem Projekt wurde die Sprint-Länge auf zwei Wochen festgelegt. Dadurch sollte ein ausreichendes Zeitfenster geschaffen werden, um die gestellten Aufgaben bearbeiten zu können. In der zweiten Projekthälfte wurde eine einwöchige Länge der Sprints ausprobiert, da sich bei zweiwöchigen Sprints gezeigt hat, dass die Aufgaben zu groß geplant und nicht immer abschließend bearbeitet werden

konnten. Die einwöchigen Sprints haben die Planung von kleineren Aufgaben unterstützt und eine gleichmäßigere Bearbeitung ermöglicht.

In den nachfolgenden Abschnitten werden grundlegende Praktiken von Scrum und die Umsetzung in der Projektgruppe näher beschrieben [Kim13].

6.1.3 Sprint

Bei der Softwareentwicklung mit Scrum stehen die Sprints im Mittelpunkt. Die gesammelten Anforderungen des Kunden werden mit den Sprints Schritt für Schritt abgearbeitet. Dafür werden die aufgestellten Anforderungen in Sprint Backlogs (siehe Abschnitt 6.1.5) eingetragen. So wird festgehalten, welche Anforderungen in einem bestimmten Sprint-Zeitraum erfüllt werden sollen. Die Anforderungen werden vom Kunden nach Wichtigkeit eingestuft und in der so entstehenden Reihenfolge umgesetzt. Die Anforderungen aus dem Sprint Backlog werden vom Team aufgeteilt und jeder Entwickler schätzt und bewertet die ihm zugewiesenen Aufgaben.

Die Entwicklung eines funktionsfähigen, vertikalen Prototyps stand während der ersten Hälfte der Projektlaufzeit im Vordergrund. Die Anforderungen aus dem Product Backlog (siehe Abschnitt 6.1.4) waren darauf ausgelegt und wurden in zweiwöchigen Sprints abgearbeitet. In der zweiten Projekthälfte sollte der vertikale Prototyp horizontal erweitert werden. Dafür wurden im Product Backlog weitere Anforderungen gesammelt und vom Product Owner – in diesem Fall der Projektbetreuer – priorisiert. Daraus ergaben sich die Aufgaben für die folgenden, nun einwöchigen, Sprints. Zu Beginn jeder Sprint-Planung wurde gemeinsam besprochen, welche Aufgaben aus dem Product Backlog in den zu planenden Sprint gezogen werden sollten. Jedes Projektmitglied wählte selbstständig für sich zu bearbeitende Aufgaben aus und hatte dafür Sorge zu tragen, dass diese während der Sprint-Laufzeit erledigt wurden. Die dafür benötigte Zeit wurde vom Teammitglied oder in Beratung mit der Gruppe geschätzt.

6.1.4 Product Backlog

Das Product Backlog ist eine Auflistung der vom Kunden gewichteten Anforderungen. So war eine grobe Entwicklungsrichtung für das Softwareprodukt erkennbar. Die Anforderungen wurden von der Projektgruppe erarbeitet, mit dem Projektbetreuer durchgesprochen und von ihm in die Prioritäten hoch, mittel und niedrig gewichtet. Das Besondere hierbei war, dass die Anforderungen zu Projektbeginn nicht vollständig waren. Während der ersten Projekthälfte waren die Anforderungen auf den vertikalen Prototypen zugeschnitten. Nach Fertigstellung des vertikalen Prototyps und mit den Erfahrungen aus dem ersten Projektabschnitt wurden die Anforderungen um zusätzliche, für die Erweiterung zu einem horizontalen Prototypen notwendigen, Anforderungen ergänzt. Während der zweiten Projekthälfte ergaben sich weitere wichtige Anforderungen, die in das Product Backlog mit aufgenommen und priorisiert wurden.

6.1.5 Sprint Backlog

Im Sprint Backlog werden die Anforderungen aufgelistet, die innerhalb eines Sprints abgearbeitet werden sollen. Für die Sprint-Planung wurden die Anforderungen aus dem Product Backlog in

Aufgaben umformuliert und im Jira-Backlog (siehe Abschnitt 6.3) der Projektgruppe gesammelt. Daraus wurden die Aufgaben in den jeweils zu planenden Sprint gezogen. Das sich daraus ergebende Sprint Backlog wurde in der Gruppe besprochen und gemeinsam beschlossen.

6.1.6 Daily Scrum

Bei jeder Projekt-Sitzung findet ein kurzes (15-minütiges) Daily Scrum statt. Dabei beschreibt jeder einzelne oder jede Arbeitsgruppe kurz die eigenen Ergebnisse und Probleme. Für gewöhnlich sollen diese Sitzungen täglich stattfinden. Aufgrund der unterschiedlichen Studien-Zeitpläne der Projektmitglieder war ein tägliches Treffen nicht umsetzbar. Die Sitzungen fanden wöchentlich statt und das Daily Scrum wurde in ein Weekly Scrum umfunktioniert. In den ersten 15 Minuten jeder Sitzung wurde von jedem Teammitglied kurz berichtet, welche Aufgaben in der vergangenen Woche bearbeitet wurden und wie der Fortschritt war. Kleinere Probleme wurden sofort besprochen, größere Anliegen in die Tagesordnung mit aufgenommen oder im Anschluss an die Sitzung besprochen.

6.1.7 Sprint Review

Nach dem Beenden eines Sprints findet eine Sprint Review statt. Dabei können dem Kunden die abgearbeiteten Anforderungen eines Sprints vorgestellt werden. Diese Möglichkeit wurde in diesem Projekt ebenfalls genutzt. So wurden den Projektbetreuern neu implementierte Funktionen vorgestellt und es wurde beidseitig auf Fragen eingegangen.

6.1.8 Sprint Retrospective

Die Sprint Retrospective ist eine Projekt-Sitzung nur für die Entwickler des Softwareprodukts, bei dem folgende Fragen geklärt werden sollen:

- Welche Ereignisse sind eingetreten?
- Was lief gut?
- Was ist verbesserungswürdig?

Nach Abschluss jedes Sprints wurde in der Sitzung eine Bewertung des Sprints vorgenommen. Diese Bewertung wurde in einer Positives-Negatives-Tabelle in dem jeweiligen Sitzungsprotokoll festgehalten. Eine gesonderte Sitzung wurde für die Sprint Retrospective nicht eingeführt.

6.2 Rollenverteilung

Die Festlegung und Einteilung der verschiedenen Rollen erfolgte innerhalb der ersten Wochen. Tabelle 6.1 zeigt die Rollenverteilung während der Projektlaufzeit.

Die Positionen wurden möglichst doppelt besetzt, um eine bessere Aufteilung der organisatorischen Arbeit zu gewährleisten. Des Weiteren konnte damit sichergestellt werden, dass stets eine zweite Person ansprechbar war. Im Folgenden werden die Aufgaben der verschiedenen Rollen kurz erläutert:

Rolle	Verantwortliche
Projektmanager	Jannik Arndt, Thomas Meents
Gruppenleiter	Markus Holznagel, Christopher Licht
Scrum Master	Moritz Eversmann
Konfigurationsbeauftragter	Naby Moussa Sow, Bernd Nottbeck
Dokumentationsbeauftragter	Andrej Albrecht, Bernhard Bruns

Tabelle 6.1: Rollenverteilung

- Der *Projektmanager* überwacht den Projektfortschritt und fungiert als Bindeglied zwischen dem Team und den Betreuern.
- Aufgabe des *Gruppenleiters* ist es die Teamsitzungen zu moderieren.
- Der *Scrum Master* ist für die Einhaltung der Scrum-Richtlinien zuständig. Darüber hinaus kümmert er sich um den Review und die Retrospektive der Sprints.
- Der *Konfigurationsbeauftragte* ist für die Einrichtung des Datenbankservers und der Versionsverwaltung (SVN) zuständig. Zudem ist er für die Hilfe und Behebung technischer Probleme verantwortlich.
- Aufgabe des *Dokumentationsbeauftragten* ist die Erstellung und Pflege der Dokumentenvorlagen, des Weiteren ist er Ansprechpartner bei dokumentationsbedingten Problemen.

Neben der Einteilung in verschiedene Rollen erfolgte für die ersten Wochen des Projekts eine Aufteilung in die drei Kleinteams „Datenbank“, „Schnittstellen“ und „GUI“.

Das Ziel des Datenbank-Teams war es aus der gegebenen MIMIC II Datenbank, ein multidimensionales Data-Warehouse aufzubauen. Um einen Gesamtüberblick zu erhalten, wurde zunächst die bestehende Datenbasis analysiert und es wurden Entwürfe in Form eines ER-Modells und eines Star-Schemas erstellt. Nach Umsetzung der Entwürfe wurden schließlich die bestehenden Daten durch einen ETL-Prozess in das Data-Warehouse importiert.

Die Aufgabe des Schnittstellen-Teams lag in der Erarbeitung und Entwicklung der internen Repräsentation der Daten. Die Hauptaufgabe war neben der Bereitstellung von Schnittstellen, die Erstellung von Klassen zur internen Darstellung von Event Logs und Prozessmodellen. Des Weiteren beschäftigte sich das Team mit der Datenbankanbindung.

Das GUI-Team war für die Entwicklung der grafischen Benutzungsoberfläche zuständig. Dafür wurde zunächst ein Mock-up erstellt, welches das Grundgerüst für die grafische Benutzungsoberfläche darstellte. Anschließend begann die Umsetzung des Modells mit dem Grafik-Framework WPF.

Nachdem die Grundlagen geschaffen waren, erfolgte die Auflösung der Kleinteams und die Aufgaben wurden teamübergreifend verteilt. Die Bearbeitung dieser Aufgaben erfolgte bei kleineren Arbeitspaketen in Einzelarbeit, während für größere Arbeitspakete neue Kleinteams gebildet wurden. Dies geschah beispielsweise bei den in Abschnitt 5.5 vorgestellten Mining-Algorithmen.

6.3 Projektwerkzeuge

Zur Unterstützung des operativen Projektmanagements sowie zur Kommunikation innerhalb der Projektgruppe wurden die webbasierten Anwendungen Confluence und Jira eingesetzt.

Confluence ist ein Wiki, welches hauptsächlich für die Kommunikation und den Wissensaustausch eingesetzt wurde. Es diente als zentrale Wissensbasis, auf die das Projektteam durchgehend zugreifen konnte. Für die Projektlaufzeit wurden zwei Bereiche eingerichtet, um die Übersichtlichkeit zu erleichtern.

Der erste Bereich beinhaltete weitestgehend organisatorische Beiträge, welche in die folgenden Kategorien gegliedert wurden:

- *Backlog:* Enthielt eine erste Version der Anforderungen an die Software sowie die Ziele für die zweite Projekthälfte.
- *C# So wird es gemacht:* Umfasste Codebeispiele und Vorgehensweisen für die Implementierung mit C#.
- *Glossar:* Vereinheitlichte das Wissen in den Bereichen Process Mining, Scrum und der MIMIC-2-Datenbank.
- *Ideen / Konzepte:* Wurde beispielsweise für die Namensfindung der Software eingesetzt, um Ideen zu sammeln und darüber abzustimmen.
- *Konfiguration:* Enthielt Anleitungen sowie Zugangsdaten für die Einrichtung begleitender Software (z. B. Oracle, SVN, . . .).
- *Organisatorisches:* Zusammenfassung von Informationen, wie z. B. die Kontaktdaten sämtlicher Teammitglieder, Kernarbeitszeiten, Richtlinien für die Projektarbeit oder die Urlaubsplanung.
- *Protokolle:* Bereich für sämtliche Protokolle der wöchentlichen Sitzungen.
- *Seminarthemen:* Übersicht der Seminarthemen sowie Verknüpfungen zu den Ausarbeitungen und Präsentationen.
- *Stundenzettel:* Wöchentliche Stundenübersicht für jedes Mitglied der Projektgruppe.

Der zweite Bereich wurde für die begleitende Dokumentation verwendet und enthielt die folgenden Kategorien: Anwendungsfälle, Allgemeine Codestruktur, Aufbau der GUI, Datenbankanbindung, Datenbank, Analyse und Verarbeitung der Daten sowie Unittests.

Weiterhin stellte Confluence einen Kalender bereit, der für alle wichtigen Termine, wie z. B. Teamsitzungen oder Urlaubsphasen, genutzt wurde.

Neben Confluence wurde als zweite webbasierte Software Jira eingesetzt. Jira ist eine Projektmanagementplattform und wird vorwiegend für die Softwareentwicklung eingesetzt. Es unterstützt das Anforderungsmanagement, die Statusverfolgung sowie den Fehlerbehebungsprozess.

Zur Umsetzung des agilen Vorgehensmodells Scrum (siehe Abschnitt 6.1) wurde Jira um das Addon Jira Agile erweitert. Dieses Addon diente ausschließlich zur Verwaltung von Scrum.

Jira stellt eine einfache Ticketverwaltung bereit, mit der während der Projektlaufzeit die einzelnen Sprints geplant wurden. Darüber hinaus war ersichtlich, welches Teammitglied mit welcher Aufgabe beschäftigt war. Ebenso war der Fortschritt innerhalb des Sprints für jedes Mitglied der Projektgruppe

erkennbar. Zudem konnten neue Funktionen, Bugs oder Aufgaben direkt dokumentiert sowie einem Benutzer zugewiesen werden. Alle erstellten Tickets wurden automatisch im Backlog von Jira verwaltet. Bei der Sprint-Planung wurden die erforderlichen Tickets in den entsprechenden Sprint-Abschnitt gezogen. Nachdem der Sprint gestartet wurde, wurden alle Tickets im Aufgabenbereich von Jira angezeigt. In diesem Bereich konnten sich die Mitglieder der Projektgruppe Tickets zuweisen sowie den Verlauf dokumentieren. Des Weiteren wird in Jira die Möglichkeit geboten, den Sprint auch nach dem Start um Tickets zu erweitern. Diese Möglichkeit wurde hauptsächlich in Anspruch genommen, falls während der Entwicklung Bugs auftraten. Nach Beendigung eines Sprints wurde von Jira ein Report erzeugt, der einen Überblick über alle bearbeiteten Tickets lieferte. Darüber hinaus wurde durch ein Burndown-Diagramm die geleistete Arbeit visualisiert, indem Planungs- und Realfall gegenübergestellt wurden.

Als Projektwerkzeug für die Versionsverwaltung von Dateien und Verzeichnissen wurde Apache Subversion (SVN) eingesetzt. Dieses System diente während der Projektlaufzeit als zentraler Speicherplatz für den Quellcode des Softwareprototypen sowie den Zwischen- und Abschlussbericht.

7 Fazit

7.1 Zusammenfassung und Rückblick

Das Projekt *Multidimensionales Process Mining* begann mit einer Seminarphase, durch die eine solide Wissensbasis für den weiteren Verlauf geschaffen wurde. Im Anschluss daran begann die Planung zur Umsetzung des Forschungsframeworks. Grundlage für die Umsetzung war das Vorgehensmodell *Scrum* (siehe Abschnitt 6.1), welches ein agiles Vorgehen in der Softwareentwicklung anstrebt.

Die Herausforderung der ersten Projekthälfte bestand in der Entwicklung eines vertikalen Prototypen. Für ein strukturiertes Vorgehen wurden zunächst kleine Teams gebildet, die grundlegende Ideen zu den Bereichen Datenbank, Schnittstellen und Benutzungsoberfläche ausarbeiteten. Diese Zuordnung wurde für die Implementierungsarbeit wieder aufgelöst. Die verschiedenen Vor- und Nachteile der vorgestellten Mining-Algorithmen aus der Seminarphase wurden von der Projektgruppe diskutiert und abgewogen. Das Ergebnis war der Entschluss als ersten Process Mining-Algorithmus den *Heuristic Miner* zu implementieren. Zur Hälfte der Projektlaufzeit konnte schließlich ein funktionsfähiger, vertikaler Prototyp präsentiert werden. Dieser bot die Möglichkeit, aus selektierten Eventdaten ein Prozessmodell in Form eines Petrinetzes zu erstellen.

Für die zweite Projekthälfte bestand die Herausforderung nun darin, den vertikalen Prototypen horizontal auszubauen. Dafür wurden zunächst Überlegungen angestellt, welchen Funktionsumfang der Softwareprototyp bekommen sollte. Eine naheliegende Idee war die Implementierung des *Alpha Miners*, da dieser Algorithmus einer der ersten in diesem Bereich war und die Ergebnisse eine gute Vergleichsbasis für den Heuristic Miner darstellen würde. Zudem wurde die Implementierung des *Heuristic Miners* weiter verbessert. Ebenso sollte es eine Möglichkeit geben, die generierten Prozessmodelle auf ihre Korrektheit in Bezug auf das Event Log zu überprüfen. Dazu wurde das *Conformance Checking* in den Funktionsumfang mit aufgenommen. Während der zweiten Projekthälfte kamen noch zwei weitere Bereiche hinzu. Zum einen sollte die zu Beginn als leere Schnittstelle geplante *Consolidation* mit einfachen, grundlegenden Funktionen gefüllt werden. Zum anderen erfuhr das Projektteam von einem neuen Mining-Algorithmus, dem *Inductive Miner - infrequent* bei dem sich die Gruppe dafür entschied diesen umzusetzen.

Als Ergebnis der einjährigen Projektzeit kann der Softwareprototyp des *Process Cube Explorer* präsentiert werden. Das erweiterbare, der Software zugrunde liegende Data-Warehouse-Schema bietet zahlreiche Analysemöglichkeiten. Dieses war für den multidimensionalen Ansatz (siehe Abschnitt 2.7) notwendig. Neben der Dimensionsauswahl auf Case-Ebene ist zusätzlich eine Auswahl und Aggregation auf Event-Ebene möglich. Wie in dem Abschnitt 5.5 erläutert, verfügt die Software über die drei Mining-Algorithmen *Heuristic Miner*, *Alpha Miner* und *Inductive Miner*, sowie ihre Erweiterungen *Alpha Miner +*, *Alpha Miner ++* und *Inductive Miner - infrequent*. Die *Consolidation*-Schnittstelle wurde umgesetzt und grundlegende Filtermethoden implementiert. Die Ergebnisse aus den Mining-Algorithmen werden als Petrinetz visualisiert und zusätzliche Informationen auf der *results-Seite* dargestellt. Diese Prozessmodelle können mithilfe der *Conformance Checking*-Verfahren *Comparing Footprint* und *Token Replay* auf die Übereinstimmung mit dem zugrunde liegenden Event Log überprüft werden. Unterschiede zwischen zwei Prozessmodellen können mithilfe des implementierten Prozessmodellvergleichs ermittelt werden. Bei der gesamten Implementierung wurde Wert auf den Einsatz von Schnittstellen sowie die funktionale Trennung von Softwarepaketen gelegt. Dies soll eine Erweiterbarkeit der Software gewährleisten. Zudem wurden Unit-Tests und User-Tests zur Ver-

besserung der Programmqualität eingesetzt. Insgesamt wurden somit die Anforderungen der ersten Projekthälfte sowie die Anforderungen F11–F25 der zweiten Projekthälfte erfüllt (siehe 3.1). Im Rahmen des Projektes konnte eine funktionsreiche Software entwickelt werden, welche den Benutzer bei der multidimensionalen Analyse von Prozessen unterstützt. Der *Process Cube Explorer* wurde im Rahmen der Informatiktage 2014 der Gesellschaft für Informatik vorgestellt. Das hierzu eingereichte Paper wurde von der Jury als eines der drei besten Paper geehrt.

7.2 Ausblick

Wie bereits erwähnt handelt es sich beim *Process Cube Explorer* um einen Prototypen. Obwohl eine Vielzahl an Funktionen bereits implementiert wurde besitzt der Prototyp noch Funktionen die Potenzial für Erweiterungen und Optimierungen bieten.

Erweiterungsmöglichkeiten in Form von weitere Mining-Algorithmen, wie beispielsweise dem *Fuzzy-* oder dem *Genetic-Miner*, können den Funktionsumfang der Software erweitern. Zudem können die bestehenden Mining-Algorithmen im Hinblick auf Laufzeit, Rekursionstiefe und Ressourceneinsatz optimiert werden. Im Bereich *Model Enhancement* und *Consolidation* gibt es verschiedene Forschungsansätze die in das Programm einfließen könnten. Auch der Bereich der Visualisierung bietet Erweiterungsmöglichkeiten. Verbesserte Anordnungsalgorithmen und eine optisch ansprechendere Darstellung können die Lesbarkeit der Prozessmodelle erhöhen. Neben der Optimierung der bestehenden Visualisierung bietet sich die Möglichkeit, weitere Visualisierungsformen wie das *Causal-* oder das *BPMN-Netz* zu implementieren. Des Weiteren können Funktionen, die sich aus den nicht umgesetzten *Kann-Kriterien* 3.1 ableiten lassen, den Funktionsumfang der Software erweitern. Dies wäre zum Beispiel bei der Umsetzung eines *Decision Minings* der Fall, welches dem Benutzer weitere Informationen bezüglich der getroffenen Entscheidungen innerhalb des Prozessmodells zur Verfügung stellt. Auch Schnittstellen für *EventLog-Operationen* und *Processmodel-Operationen* sind möglich. Aufgrund der prototypischen Implementierung bieten sich Möglichkeiten zur Performance- und Ressourcenoptimierung in einigen Bereichen der Software. Hierzu zählen beispielsweise parallele Datenbankabfragen und Mehrkernoptimierungen. Die während der Entwicklung genutzte Datenbasis stammt aus dem Bereich der medizinischen Versorgung. Das Anwendungskonzept ist auf diese nicht beschränkt. Daher ist die Nutzung der Software in anderen Anwendungsdomänen erstrebenswert und das Konzept der Software weiter zu überprüfen.

Glossar

Activity

Eine Aktivität (Activity) ist die Tätigkeit in einem Event.

Addon

Ein Addon ist ein optionales Modul, welches bestehende Hard- oder Software erweitert.

Alpha Miner

Der Alpha Algorithmus durchsucht das Event Log nach speziellen Mustern. Die Durchsuchung des Event Log erfolgt paarweise, wodurch der Alpha Algorithmus keine kurzen Loops der Größe eins und zwei darstellen kann.

Alpha Miner +

Der Alpha Miner + ist eine Verbesserung des Alpha Miners und in der Lage, Loops der Größe zwei zu erkennen.

Alpha Miner ++

Der Alpha Miner ++ ist eine Verbesserung des Alpha Miners und in der Lage, Loops der Größe eins zu erkennen.

Backlog

Ein Backlog ist eine Liste mit Dingen, die noch erledigt werden müssen, z.B. ein Anforderungskatalog.

Behavioral Appropriateness

Die Behavioral Appropriateness ist eine Methode zur Messung der Precision und sagt aus wie genau ein modelliertes Verhalten (Prozessmodell) das Event Log wiedergibt.

Bug

Ein Bug ist ein Softwarefehler und bezeichnet das Fehlverhalten von Computerprogrammen.

Case

Ein Case ist im Process Mining ein Fall, z. B. der Aufenthalt in einem Krankenhaus. In jedem Fall sind Ereignisse enthalten, von denen jedes zu genau einem Fall gehört. Jeder Fall beinhaltet verschiedene Aktivitäten und wird innerhalb eines Ereignisprotokolls als Prozessinstanz dargestellt.

Classifier

Ein Classifier innerhalb des Process Cube Explorers entspricht einer DB-Tabellenspalte, der zur Identifizierung von Events verwendet wird. Der Classifier wird ebenfalls für die Beschriftung von Transitionen in einem Petrinetz genutzt.

Comparing Footprint

Comparing Footprint ist ein Conformance Checking Verfahren zur Berechnung der Fitness eines Prozessmodells bezogen auf das zugrunde liegende Event Log.

Computational Intelligence

Computational Intelligence ist ein Gebiet der künstlichen Intelligenz.

Confluence

Confluence ist ein webbasiertes Wiki.

Conformance Checking

Conformance Checking ist eine Process Mining Technik für den Vergleich eines Prozessmodells mit einem Event Log desselben Prozesses.

Consolidation

Consolidation ist die Reduzierung der Anzahl von Prozessmodellen auf ein für den Benutzer relevantes Ergebnis.

Data Mining

Data Mining ist die systematische Anwendung statistischer Methoden auf einen Datenbestand mit dem Ziel, neue Muster zu erkennen.

Data-Warehouse

Ein Data-Warehouse ist eine Datenbank, in der Daten aus unterschiedlichen Quellen in einem einheitlichen Format zusammengefasst werden.

Datenwürfel

Ein Datenwürfel stellt eine mehrdimensionale Darstellung von Kennzahlen dar.

Event

Ein Event ist im Process Mining ein Ereignis innerhalb eines Falls, z. B. die Untersuchung mit einem Ultraschallgerät.

Event Log

Ein Event Log ist eine Ereignisprotokoll, in dem eine Menge an Ereignissen abgelegt ist.

Factory-Pattern

Das Factory-Pattern ist ein Entwurfsmuster aus der Softwareentwicklung, eine Factory-Methode erzeugt Objekte basierend auf übergebenen Parametern.

Fitness

Die Fitness ist eine Qualitätskennzahl und besagt auf der Case-Ebene, wie viele Cases aus einem Event Log auf ein Prozessmodell abspielbar sind. Auf der Event-Ebene gibt die Fitness den Anteil der Events an, die sich an das Verhalten aus dem Event Log halten.

Flower Model

Ein Flower Model stellt alle Abläufe eines Event Logs, aber auch jedes anderen Event Logs, das sich auf die selbe Menge an Events bezieht, dar. Eine Stelle verweist auf viele Transitionen, die wiederum auf die selbe Stelle zurück verweisen. Das Model ähnelt einer Blume.

Footprint

Ein Footprint drückt die Beziehungen zwischen Events eines Event Logs oder Prozessmodells mit Symbolen in einer Matrix aus.

Garbage Collection

Als Garbage Collection wird in der Softwaretechnik die automatisierte Speicherbereinigung bezeichnet.

Generalization

Die Generalization ist eine Qualitätskennzahl und gibt an in wie weit das Verhalten aus dem Event Log in dem Prozessmodell reduziert dargestellt wurde.

GUI

GUI steht für Graphical User Interface, also das, was das System dem Nutzer anzeigt (Fenster, Label, Button) und mit dem er interagieren kann.

Heuristic Miner

Der Heuristic Miner ist ein Process Mining Algorithmus (siehe auch Heuristic Mining).

Inductive Miner

Siehe Inductive Mining.

Inductive Miner - infrequent

Der Inductive Miner - infrequent ist eine Erweiterung des Inductive Miners, der die Schritte des Inductive Miners um Filter für seltenes Verhalten ergänzt.

Jira

Jira ist ein Werkzeug für die Verwaltung von Aufgaben.

LINQ

LINQ steht für Language Integrated Query und ist ein programmtechnisches Verfahren von Microsoft für den Zugriff auf Daten.

Long Distance Dependency

Beschreibung ergänzen!

Loop

Eine Loop ist eine innerhalb eines Petrinetzes künstlich erzeugte Transition, um ein sich wiederholendes Vorgehen anzuzeigen.

MIMIC II Database

Die MIMIC II Database ist eine Datenbank des PhysioNet und beinhaltet physiologische Daten.

Mining-Algorithmus

Ein Mining-Algorithmus ist eine Technik des Process Mining.

Mock-up

Ein Mock-up ist eine Attrappe, z. B. eine Nachbildung für Präsentationszwecke.

Model Enhancement

Model Enhancement ist ein Bereich des Process Mining zur Erweiterung und Verbesserung von Prozessmodellen.

Petrinetz

Ein Petrinetz ist ein gerichteter Graph bestehend aus Knoten und Kanten.

Precision

Die Precision ist eine Qualitätskennzahl und gibt an, wie präzise ein Prozessmodell auf die im Event Log vorhandenen Ausführungen passt.

Process Discovery

Process Discovery ist ein Bereich des Process Mining und fasst die verschiedenen Methoden und Techniken für die Erkennung von Prozessen zusammen.

Process Mining

Process Mining ist eine Technologie des Prozessmanagements zur Rekonstruktion und Analyse von Prozessen anhand digitaler Spuren in IT-Systemen.

Product Backlog

Das Product Backlog ist eine Auflistung der vom Kunden gewichteten Anforderungen.

Property

Properties beinhalten sowohl Aspekte von Feldern als auch von Methoden.

Rauschen

Rauschen (engl. Noise) bezeichnet fehlerhafte Einträge in einem Event Log.

Salt

Salt bezeichnet in der Kryptographie eine zufällig gewählte Zeichenfolge.

Scrum

Scrum ist ein agiles Vorgehensmodell für die Softwareentwicklung und das Projektmanagement.

Simplicity

Die Simplicity ist eine Qualitätskennzahl und gibt die Komplexität des resultierenden Prozessmodells an.

Sprint

Der Sprint ist bei Scrum-Projekten ein sich wiederholendes Zeitfenster, in dem eine Menge an Aufgaben abgearbeitet wird.

Sprint Backlog

Im Sprint Backlog werden die Anforderungen aufgelistet, die innerhalb eines Sprints abgearbeitet werden sollen.

Sprint Retrospective

Die Sprint Retrospective ist eine Projekt-Sitzung nur für die Entwickler eines Softwareprodukts.

Sprint Review

In der Sprint Review werden die abgearbeiteten Anforderungen eines Sprints dem Kunden vorgestellt.

Star-Schema

Das Star-Schema liegen die Dimensionstabellen denormalisiert vor, was eine bessere Verarbeitungsgeschwindigkeit zu Lasten der Datenintegrität und des Speicherplatzes mit sich bringt.

Structural Appropriateness

Die Structural Appropriateness ist eine Methode zur Messung des Grads der strukturellen Angemessenheit zwischen einem modellierten und tatsächlichen Verhalten, d. h. zwischen Event Log und Prozessmodell.

Token Replay

Das Token Replay ist ein Verfahren des Conformance Checking zur Ermittlung der Fitness eines Prozessmodells.

Wiki

Ein Wiki ist ein Hypertext-System für Webseiten, deren Inhalte von den Benutzern nicht nur gelesen, sondern auch online direkt im Webbrowser geändert werden können.

Workflow-Netz

Ein Workflow-Netz ist eine besondere Form eines Petrinetzes und besitzt genau eine Start- und eine End-Stelle. Am Anfang bestitzt nur die Start-Stelle eine Markierung, nachdem das Netz durchgelaufen wurde besitzt nur noch die End-Stelle eine Markierung.

Abkürzungsverzeichnis

DRG	Diagnosis Related Groups (deutsch: diagnosebezogene Fallgruppen)
DWH	Data-Warehouse
ER-Modell	Entity-Relationship-Modell
ETL	Extract, Transform, Load
HOLAP	Hybrid Online Analytical Processing
ICD	International Statistical Classification of Diseases and Related Health Problems
IMi	Inductive Miner-infrequent
MDDBMS	Multidimensional Datenbank Management System
MOLAP	Multidimensional Online Analytical Processing
OLAP	Online Analytical Processing
RDBMS	Relational Datenbank Management System
ROLAP	Relational Online Analytical Processing
SQL	Structured Query Language
SVN	Subversion
WPF	Windows Presentation Foundation

Abbildungen

Tabellenverzeichnis

Listings

Literatur

[Aal10] AALST, Will van d.: Process Discovery: Capturing the Invisible. In: *IEEE Computational Intelligence Magazine* 5 (2010), Februar, Nr. 1, 28–41. http://dx.doi.org/10.1109/MCI.2009.935307. – DOI 10.1109/MCI.2009.935307. – ISSN 1556–603X

[Aal11] AALST, Wil M. d.: *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag Berlin Heidelberg, 2011. – ISBN 9783642193446

[AB09] ANDREAS BAUER., H. Günzel: *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung.* dpunkt, Heidelberg., 2009

[Amr12] AMREIN, C.: *Business- und IT-Development*. Compendio Bildungsmedien, 2012 http://books.google.de/books?id=wou8xHAUMDwC. – ISBN 9783715595665

[BS13] BURATTIN, Andrea ; SPERDUTI, Alessandro: *PLG: a Framework for the Generation of Business Process Models and their Execution Logs*. http://www.processmining.it/public/publications/2010-bpi.pdf. Version: 2013. – Letzter Zugriff am 29.3.2014

[CA07] CHRISTIAN, W G. ; AALST, Wil M P Van D.: *Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics*. 2007

[Cor13] CORDES, Carsten: *Entwicklung eines Werkzeugs zur Visualisierung von Differenzen zwischen Prozessmodellen*. 2013. – Bachelorarbeit

[DMV+05] DONGEN, Boudewijn F. ; MEDEIROS, Ana Karla A. ; VERBEEK, HMW ; WEIJTERS, AJMM ; VAN DER AALST, Wil M.: The ProM framework: A new era in process mining tool support. In: *Applications and Theory of Petri Nets 2005*. Springer, 2005, S. 444–454

[Far11] FARKISCH, Kiumars: *Data Warehouse Systeme kompakt: Aufbau, Architektur, Grundfunktionen*. Springer, 2011. – 11–44 S.

[GRC09] GÓMEZ, Jorge M. ; RAUTENSTRAUCH, Claus ; CISSEK, Peter: *Einführung in Business Intelligence mit SAP NetWeaver 7.0*. Springer Berlin, Berlin., 2009. – ISBN 9783540795360

[KH06] KEMPER H., Unger C. Mehanna W. W. Mehanna W.: *Business Intelligence - Grundlagen und praktische Anwendungen. Eine Einführung in die IT-basierte Managementunterstützung.* Vieweg, Wiesbaden, 2006

[KHL+10] KOEGEL, Maximilian ; HERRMANNSDOERFER, Markus ; LI, Yang ; HELMING, Jonas ; DAVID, Jörn: Comparing State- and Operation-Based Change Tracking on Models. In: *EDOC*, IEEE Computer Society, 2010. – ISBN 978–0–7695–4163–1, 163-172

[Kim13] KIM, Don: The state of scrum: Benchmarks and guidelines. (2013). http://www.scrumalliance.org/scrum/media/ScrumAllianceMedia/Files%20and%20PDFs/State%20of%20Scrum/2013-State-of-Scrum-Report_062713_final.pdf

[Kra12] KRAHN, Tobias: *Wissensbasierte Auswahl von Klassifikationsknoten in Analytischen Informationssystemen*. 2012. – Masterarbeit, Universität Oldenburg

[LFA13a] LEEMANS, Sander J. J. ; FAHLAND, Dirk ; AALST, Wil M. P. d.: Discovering Block-

Structured Process Models from Event Logs - A Constructive Approach. In: COLOM, José M. (Hrsg.) ; DESEL, Jörg (Hrsg.): *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings* Bd. 7927, Springer, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–38696–1, S. 311–329

[LFA13b] LEEMANS, Sander J. ; FAHLAND, Dirk ; AALST, Wil M. d.: Discovering Block-Structured Process Models From Event Logs Containing Infrequent Behaviour. (2013). http://fluxicon.com/blog/wp-content/uploads/2013/09/Discovering-Block-Structured-Process-Models.pdf

[LU07] LESER U., Naumann F.: *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* Dpunkt-Verl., Heidelberg, 2007

[Mad00] MADER, Sven: *SAP Business Information Warehouse als neue Dimension des Informationsmanagements: Theoretische Grundlagen und praktische Ausführungen an der SAP R/3-Beispielapplikation Konstenstellenrechnung*. Diplom.de, 2000. – ISBN 9783832428464

[MDAW05] MEDEIROS, A.K.A. de ; DONGEN, B.F. van ; AALST, W.M.P. van d. ; WEIJTERS, A.J.M.M.: *Process Mining: Extending the alpha-algorithm to Mine Short Loops*. http://wwwis.win.tue.nl/~wvdaalst/publications/p221.pdf. Version: 2005. – Department of Technology Management , Eindhoven University of Technology

[PM12] PROCESS MINING, IEEE T.: Process Mining Manifesto. In: DANIEL, Florian (Hrsg.) ; BARKAOUI, Kamel (Hrsg.) ; DUSTDAR, Schahram (Hrsg.): *Business Process Management Workshops(1), Jg. 99 von Lecture Notes in Business Information Processing*. Springer-Verlag, 2012, S. 169–194

[Sch] SCHWABER, Ken: *Scrum - it's about common sense*. http://www.controlchaos.com. – Letzter Zugriff am 25.9.2013

[Sch04] SCHIMM, Guido: *Process Mining*. http://www.processmining.de/1001.html. Version: 2004. – Letzter Zugriff am 16.9.2013

[SSG13] SATTLER, Kai-Uwe ; SAAKE ; GUNTER, Köppen Veit: *Data-Warehouse-Technologien*. http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/dw/dwt1314/03-Modellierung.pdf. Version: 2013

[Tie09] TIEMEYER, E.: *Handbuch IT-Management: Konzepte, Methoden, Lösungen und Arbeitshilfen für die Praxis*. Hanser Fachbuchverlag, 2009 http://books.google.de/books?id=9R2QIzYF_PkC. – ISBN 9783446418424

[VBDA10] VERBEEK, H.M.W. ; BUIJS, J.C.A.M. ; DONGEN, B.F. van ; AALST, W.M.P. van der: ProM 6: The Process Mining Toolkit. 615 (2010), S. 34–39

[VK12] VEIT KOEPPEN, Sattler Kai U. Saake Günter G. Saake Günter: *Data Warehouse Technologien*. mitp, Verl.Gruppe Huethig, Jehle, Rehm, 2012

[Vog13] VOGELGESANG, Thomas: *Multidimensional Process Mining. A flexible analysis approach for health services research*. 2013

[W.10] W., Reisig: *Petrinetze - Modellierungstechnik, Analysemethoden, Fallstudien.* Vieweg + Teubner, Wiesbaden, 2010

[WAM06] WEIJTERS, A.J.M.M. ; AALST, W.M.P. van d. ; MEDEIROS, A.K. A.: *Process mining with the heuristics miner-algorithm*. http://dx.doi.org/10.1.1.118.8288. Version: 2006

[Wes12] WESKE, Prof. Dr. M.: *Prozessorientierte Informationssysteme II*. https://www.tele-task.de/archive/series/overview/900/. Version: 2012. – Hasso-Plattner-Institut, tele-TASK E-Lectures

Index

8 Anhang zu Tests

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Bernd_BN-MOBIL 2014-03-30 21_03_46.coverage	2903	38,95 %	4550	61,05 %
conformancechecking.dll	148	18,66 %	645	81,34 %
pgmpm.ConformanceChecking	148	18,66 %	645	81,34 %
consolidationalgorithm.dll	26	11,66 %	197	88,34 %
pgmpm.Consolidation	0	0,00 %	24	100,00 %
pgmpm.Consolidation.Algorithm	26	13,07 %	173	86,93 %
database.dll	636	33,60 %	1257	66,40 %
pgmpm.Database	455	30,01 %	1061	69,99 %
pgmpm.Database.AggregationConfiguration	25	100,00 %	0	0,00 %
pgmpm.Database.Exceptions	0	0,00 %	87	100,00 %
pgmpm.Database.Helper	135	84,38 %	25	15,63 %
pgmpm.Database.Model	0	0,00 %	46	100,00 %
pgmpm.Database.Properties	21	35,59 %	38	64,41 %
diff.dll	136	54,18 %	115	45,82 %
pgmpm.Diff	0	0,00 %	4	100,00 %
pgmpm.Diff.DiffAlgorithm	136	55,06 %	111	44,94 %
matrixselection.dll	335	43,73 %	431	56,27 %
pgmpm.MatrixSelection	133	91,72 %	12	8,28 %
pgmpm.MatrixSelection.Dimensions	87	40,65 %	127	59,35 %
pgmpm.MatrixSelection.Fields	115	28,97 %	282	71,03 %
pgmpm.MatrixSelection.Properties	0	0,00 %	10	100,00 %
miningalgorithm.dll	1497	59,38 %	1024	40,62 %
pgmpm.MiningAlgorithm	474	32,05 %	1005	67,95 %
pgmpm.MiningAlgorithm.Exceptions	0	0,00 %	8	100,00 %
pgmpm.MiningAlgorithm.InductiveV2	1023	98,94 %	11	1,06 %
processmodel.dll	125	12,43 %	881	87,57 %
pgmpm.Model	3	20,00 %	12	80,00 %
pgmpm.Model.PetriNet	122	12,31 %	869	87,69 %

Abbildung 8.1: Code Coverage Results ausführliche Aufstellung

9 Anhang - Conformance Checking

	Aufnahme	Anamnese	Blutentnahme	Röntgen	MRT	Befundung	Operation	Verband	Entlassen
Aufnahme	#	→	#	#	#	#	#	#	#
Anamnese	←	#	→	→	→	#	#	#	#
Blutentnahme	#	←	#	||	||	→	#	#	#
Röntgen	#	←	||	#	#	→	#	#	#
MRT	#	←	||	#	#	→	#	#	#
Befundung	#	#	←	←	←	#	→	#	#
Operation	#	#	#	#	#	←	#	→	#
Verband	#	#	#	#	#	#	←	@	→
Entlassen	#	#	#	#	#	#	#	←	#

Tabelle 9.1: Footprint Event Log

	Aufnahme	Anamnese	Blutentnahme	Röntgen	MRT	Befundung	Operation	Verband	Entlassen
Aufnahme	#	→	#	#	#	#	#	#	#
Anamnese	←	#	→	→	→	#	#	#	#
Blutentnahme	#	←	#	||	||	→	#	#	#
Röntgen	#	←	||	#	#	→	#	#	#
MRT	#	←	||	#	#	→	#	#	#
Befundung	#	#	←	←	←	#	→	#	#
Operation	#	#	#	#	#	←	#	→	#
Verband	#	#	#	#	#	#	←	#	→
Entlassen	#	#	#	#	#	#	#	←	#

Tabelle 9.2: Footprint Petrinetz

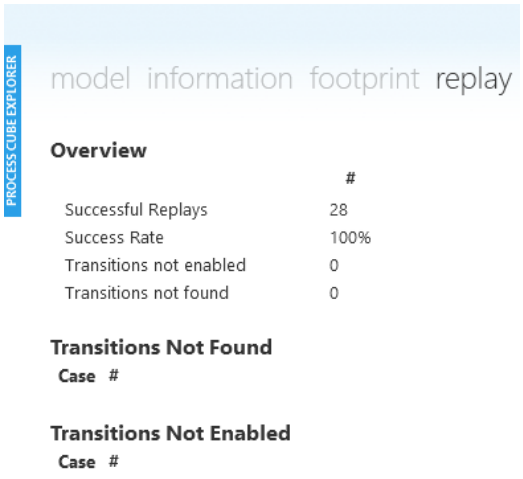


Abbildung 9.1: Token Replay - Fitness bei erkannter Loop

10 Anhang - Inductive Miner

```
public void DivideAndConquer()
  {
   if (Operation == OperationsEnum.isUnkown)
   {
    GraphNode.ReBuildeEventualFollower(null);
    GraphNode.CleanUpHelperList(null);
    foreach (InductiveMinerGraphNode nody in GraphNode.
       GetMyEventualNodes())
     {
      nody.ReBuildeEventualFollower(null);
      nody.CleanUpHelperList(null);
     }
     if (GraphNode.EventualFollowerList.Count <= 1)
     {
       Operation = OperationsEnum.isLeaf;
     }
   }
   if (Operation != OperationsEnum.isLeaf)
     {
         if (CheckSequenceCut())
         {
           newStart.ReBuildeEventualFollower(null);
           newStart.CleanUpHelperList(null);

           foreach (InductiveMinerGraphNode nody in newStart.
              GetMyEventualNodes())
            {
             nody.ReBuildeEventualFollower(null);
             nody.CleanUpHelperList(null);
            }

           LeftLeaf = new InductiveMinerTreeNode(petriNet,
              GraphNode, startEvent);
           RightLeaf = new InductiveMinerTreeNode(petriNet,
              newStart, newStart.Name);
           Operation = OperationsEnum.isSequence;
         }
         else if (CheckXorCut(GraphNode))
         {
          newStart.ReBuildeEventualFollower(null);
          newStart.CleanUpHelperList(null);

          foreach (InductiveMinerGraphNode nody in newStart.
             GetMyEventualNodes())
```

```
            {
             nody.ReBuildeEventualFollower(null);
             nody.CleanUpHelperList(null);
            }

          LeftLeaf = new InductiveMinerTreeNode(petriNet,
             GraphNode, startEvent);
          RightLeaf = new InductiveMinerTreeNode(petriNet,
             newStart, newStart.Name);
          Operation = OperationsEnum.isXOR;
        }
        else if (CheckLoopCut(GraphNode))
        {
         newStart.ReBuildeEventualFollower(null);
         newStart.CleanUpHelperList(null);
         foreach (InductiveMinerGraphNode nody in newStart.
            GetMyEventualNodes())
          {
          nody.ReBuildeEventualFollower(null);
          nody.CleanUpHelperList(null);
          }

          LeftLeaf = new InductiveMinerTreeNode(petriNet,
             GraphNode, startEvent );
          RightLeaf = new InductiveMinerTreeNode(petriNet,
             newStart, newStart.Name );
          Operation = OperationsEnum.isLoop;
         }
         else if (CheckAndCut(GraphNode))
         {
          newStart.ReBuildeEventualFollower(null);
          newStart.CleanUpHelperList(null);
          foreach (InductiveMinerGraphNode nody in newStart.
             GetMyEventualNodes())
           {
           nody.ReBuildeEventualFollower(null);
           nody.CleanUpHelperList(null);
         }

          LeftLeaf = new InductiveMinerTreeNode(petriNet,
             GraphNode, startEvent);
          RightLeaf = new InductiveMinerTreeNode(petriNet,
             newStart, newStart.Name);
          Operation = OperationsEnum.isParallel;
       }
    }
```

```
        else
        {
        if (GraphNode.FollowerList.Count > 0)
         Event = GraphNode.FollowerList[0].ToNode.Name;
       }
    }
```

Listing 10.1: Devide-and-Conquer-Verfahren

```
public void EliminateInfrequent()
  {
   if (FollowerList.Count > 1)
    {
     InductiveMinerRow lastRow = FollowerList.Last<
        InductiveMinerRow>();
     int ThreshHoldValue = (int)Math.Round(lastRow.Count *
        threshHold);
     bool SomethingWasRemoved = true;
     List<InductiveMinerRow>.Enumerator e = FollowerList.
        GetEnumerator();
     List<InductiveMinerRow> deleteList = new List<
        InductiveMinerRow>();
     e.MoveNext();
       do
       {
        if (e.Current.Count < ThreshHoldValue)
        {
        deleteList.Add(e.Current);
        }
       else
       {
       SomethingWasRemoved = false;
       }
      if (!e.MoveNext())
      {
        SomethingWasRemoved = false;
      }

      } while (SomethingWasRemoved);
      e.Dispose();
      foreach (InductiveMinerRow deleteRow in deleteList)
        {
         FollowerList.Remove(deleteRow);
        }
     }
   }
```

Listing 10.2: Filterung von infrequence

```
private Place TraverseTreePetrinet(InductiveMinerTreeNode node,
   Place relayedPlace, Place relayedOutgoingPlace = null, Place
   relayedIncomingPlace = null)
 {
  if (node.Operation.Equals(OperationsEnum.isLeaf))
  {
  return DrawLeafPetrinet(node, relayedPlace,
     overwriteIncomingPlace: relayedIncomingPlace,
     overwriteOutgoingPlace: relayedOutgoingPlace);
  }

   if (node.Operation.Equals(OperationsEnum.isSequence))
     {
      if (node.LeftLeaf != null)
      {
       Place temp = TraverseTreePetrinet(node.LeftLeaf,
          relayedPlace);
       if (node.RightLeaf != null)
         temp = TraverseTreePetrinet(node.RightLeaf, temp);
           return temp;
       }
      }
      else if (node.Operation.Equals(OperationsEnum.isXOR))
       {
        if (node.LeftLeaf != null)
         {
           Place tempXORExitPlace = TraverseTreePetrinet(node.
              LeftLeaf, relayedPlace);

            if (node.RightLeaf == null)
              return null;

            Place newPlace = TraverseTreePetrinet(node.RightLeaf,
               tempXORExitPlace, relayedIncomingPlace: relayedPlace
               , relayedOutgoingPlace: tempXORExitPlace);

            if (tempXORExitPlace != null)
               return tempXORExitPlace;
            return newPlace;
           }
         }
         else if (node.Operation.Equals(OperationsEnum.isLoop))
         {
```

```
        Place tempLoopEntrancePlace = relayedPlace;

         if (node.LeftLeaf != null)
         {
          Place temp = TraverseTreePetrinet(node.LeftLeaf,
              relayedPlace);

        Place tempLoopExitPlace = temp;
        if (node.RightLeaf != null)
          {
           //prevents a Nullpointer-Exception if the first Place
               is a Loop.
           if (tempLoopEntrancePlace != IMPetriNet.Places[0])
           {
            temp = TraverseTreePetrinet(node.RightLeaf, temp);
            IMPetriNet.AddTransition("Loop", incomingPlace: temp
               , outgoingPlace: tempLoopEntrancePlace,
            isLoop: true);
           }
           else
           {
           temp = TraverseTreePetrinet(node.RightLeaf, temp,
              relayedOutgoingPlace: tempLoopEntrancePlace);
           if (tempLoopExitPlace != null)
             temp = tempLoopExitPlace;
           }
          }
         return temp;
        }
    }
    else if (node.Operation.Equals(OperationsEnum.isParallel))
    {
     Transition ANDSplit = new Transition("AND-Split");
     ANDSplit.AddIncomingPlace(relayedPlace);
     IMPetriNet.Transitions.Add(ANDSplit);
     Place NewPlaceLeft = new Place();
     Place NewPlaceRight = new Place();
     IMPetriNet.Places.Add(NewPlaceLeft);
     IMPetriNet.Places.Add(NewPlaceRight);
     ANDSplit.AddOutgoingPlace(NewPlaceLeft);
     ANDSplit.AddOutgoingPlace(NewPlaceRight);
     Transition ANDJoin = new Transition("AND-Join");
     Place AndJoinOutgoingPlace = new Place();
     ANDJoin.AddOutgoingPlace(AndJoinOutgoingPlace);
     IMPetriNet.Places.Add(AndJoinOutgoingPlace);
     IMPetriNet.Transitions.Add(ANDJoin);
```

```
        if (node.LeftLeaf != null)
         {
          Place temp = TraverseTreePetrinet(node.LeftLeaf,
             relayedPlace, relayedIncomingPlace: NewPlaceLeft);
          ANDJoin.AddIncomingPlace(temp);

           if (node.RightLeaf != null)
            {
             temp = TraverseTreePetrinet(node.RightLeaf,
                NewPlaceRight);
             ANDJoin.AddIncomingPlace(temp);
             }
            return AndJoinOutgoingPlace;
          }
         }
         else
         {
          throw new Exception("Something in the process tree is
             wrong.");
         }
       return relayedPlace;
   }
```

Listing 10.3: Aufbau des Petrinet

```
private Place DrawLeafPetrinet(InductiveMinerTreeNode node, Place
   relayed, Place overwriteOutgoingPlace = null, Place
   overwriteIncomingPlace = null)
 {
   if (!node.Operation.Equals(OperationsEnum.isLeaf))
     throw new Exception("Only leafs can be drawn.");

    Place outgoing = new Place();
    petriNet.Places.Add(outgoing);

    if (overwriteIncomingPlace != null)
      relayed = overwriteIncomingPlace;

    Transition transition = new Transition(node.Event.Name) {
       IsDrawn = false };
    transition.AddIncomingPlace(relayed);
    transition.AddOutgoingPlace(outgoing);
    if (overwriteOutgoingPlace != null)
      {
       transition.OutgoingPlaces.Remove(outgoing);
       transition.OutgoingPlaces.Add(overwriteOutgoingPlace);
```

```
      }
    petriNet.Transitions.Add(transition);

   return outgoing;
 }
```

Listing 10.4: Zeichnen der Transition im Prozess-Baum

```
private bool CheckSequenceCut()
 {
  HashSet<InductiveMinerRow> sequenceList = SequenceCutHelper(
     GraphNode);
  bool isSequence = sequenceList.Any();
  if (isSequence)
   {
   int middle = IsEven(sequenceList.Count) ? (sequenceList.Count -
      1) / 2 : sequenceList.Count / 2;
   InductiveMinerRow middleRow = sequenceList.ElementAt(middle);
   List<InductiveMinerRow> sequenceNodes = sequenceList.Where(k =>
      k.FromNode.Equals(middleRow.FromNode) ||

      k.ToNode.Equals(middleRow.ToNode)).ToList();

    var fromNodeQuery = from row in sequenceNodes
            where row.FromNode != middleRow.FromNode
            select row.FromNode;

     var toNodeQuery = from row in sequenceNodes
            where row.ToNode != middleRow.ToNode
            select row.ToNode;

      List<InductiveMinerGraphNode> fromNodes = fromNodeQuery.
         ToList();
      List<InductiveMinerGraphNode> toNodes = toNodeQuery.ToList();
      sequenceNodes.AddRange(sequenceList.Where(k => fromNodes.
         Contains(k.FromNode) && toNodes.Contains(k.ToNode)).ToList
         ());
      sequenceList = new HashSet<InductiveMinerRow>(sequenceNodes);

      foreach (InductiveMinerRow sequenceRow in sequenceList)
        {
          newStart.AddDirectFollower(sequenceRow.ToNode);
          foreach (InductiveMinerGraphNode eventllyNode in
             sequenceRow.ToNode.GetMyEventualNodes())
          {
             newStart.AddEventualFollower(eventllyNode);
          }
```

```
      sequenceRow.FromNode.WasCut = true;
      sequenceRow.FromNode.DeleteFollower(sequenceRow);
      sequenceRow.FromNode.ReBuildeEventualFollower(null);
      sequenceRow.FromNode.CleanUpHelperList(null);
    }

   newStart.ReBuildeEventualFollower(null);
   newStart.CleanUpHelperList(null);
   }
   return isSequence;
 }
```

Listing 10.5: Sequence Cut

```
private bool CheckLoopCut(InductiveMinerGraphNode graphNode)
 {
  bool foundSth = false;
  if (!visitedLoopCheckNodes.Contains(graphNode))
   {
   visitedLoopCheckNodes.Add(graphNode);
   List<InductiveMinerRow> followerList = graphNode.FollowerList;
   foreach (InductiveMinerRow row in followerList)
    {
     var query = from SearchRow in row.ToNode.EventualFollowerList
                          where SearchRow.ToNode == row.FromNode
                          select SearchRow;

    InductiveMinerRow currentRow = query.FirstOrDefault();
    bool and = false;
    if (currentRow != null)
     {
       foreach (InductiveMinerGraphNode andCheckNode in currentRow.
          FromNode.GetMyDirectNodes())
       if (andCheckNode.GetMyDirectNodes().Contains(currentRow.
          FromNode)) and = true;
      }
     if (currentRow != null && !and)
      {
       foundSth = true;
       bool cutFound = executeLastCutInLoop(row.FromNode, row.
          ToNode);
       if (cutFound)
        {
         executeFirstCutInLoop(row.FromNode, row.ToNode);
         return true;
         }
```

```
        }
        if (foundSth) { }
        else
        {
         if (CheckLoopCut(row.ToNode)) return true;
         }
        }
       }
   return foundSth;
 }
```

Listing 10.6: Loop Cut

```
private bool CheckXorCut(InductiveMinerGraphNode graphNode)
 {
  List<InductiveMinerGraphNode> followerList = graphNode.
     GetMyDirectNodes();
  List<InductiveMinerRow> deleteList = new List<InductiveMinerRow
     >();
  bool bo = false;
  bool foundone = false;
  bool goOn = true;
  if (graphNode.FollowerList.Count > 1)
     {
     List<InductiveMinerRow>.Enumerator e = graphNode.FollowerList.
        GetEnumerator();
     e.MoveNext();
     do
      {
       if (!e.Current.ToNode.FollowerContains(followerList))
        {
         if (foundone)
         {
          goOn = false;
          deleteList.Add(e.Current);
         }
        foundone = true;
       }
     if (goOn)
      {
       goOn = e.MoveNext();
      }
    } while (goOn);
    e.Dispose();

    foreach (InductiveMinerRow deleteRow in deleteList)
    {
```

```
      newStart.AddDirectFollower(deleteRow.ToNode);
      foreach (InductiveMinerRow row in deleteRow.ToNode.
         EventualFollowerList)
      {
      if (!newStart.GetMyEventualNodes().Contains(row.ToNode))
        newStart.EventualFollowerList.Add(new InductiveMinerRow(
           newStart, row.ToNode));
     }

    graphNode.WasCut = true;
    graphNode.DeleteFollower(deleteRow);

    bo = true;
   }
  if (bo)
  {
   newStart.ReBuildeEventualFollower(null);
   newStart.CleanUpHelperList(null);
   graphNode.ReBuildeEventualFollower(null);
   graphNode.CleanUpHelperList(null);
   }
   return bo;
  }
 else return false;
 }
```

Listing 10.7: Exclusive Choise Cut

```
private bool CheckAndCut(InductiveMinerGraphNode graphNode)
 {
 bool wasSplit = false;
 if (graphNode.FollowerList.Count > 1)
 {
 List<InductiveMinerRow> followerList = graphNode.FollowerList;
 List<InductiveMinerRow> deleteList = new List<InductiveMinerRow>()
    ;
 foreach (InductiveMinerRow row in followerList)
  {
     InductiveMinerRow currentRow = row.ToNode.FollowerList.
        FirstOrDefault();
     if (currentRow.ToNode.GetMyDirectNodes().Contains(currentRow.
        FromNode))
      {
       deleteList.Add(currentRow);
       }
   }
   wasSplit = deleteList.Any();
```

```
  if (wasSplit)
    {
    foreach (InductiveMinerRow row in deleteList)
    {
     row.FromNode.DeleteFollower(row);
    }

   InductiveMinerRow lastRow = deleteList.Last();
   newStart.AddDirectFollower(lastRow.FromNode);
   newStart.AddEventualFollower(lastRow.FromNode);
   graphNode.DeleteFollower(graphNode.GetRowWithFollower(lastRow.
      FromNode));
   graphNode.ReBuildeEventualFollower(null);
   graphNode.CleanUpHelperList(null);
  }
 }
 return wasSplit;
}
```

Listing 10.8: Parallel Cut