ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG
Technische Fakultät
Institut für Informatik und Gesellschaft
Prof. Dr. Günter Müller

Bachelorarbeit

Workflow Mining: Ein Überblick mit Gegenüberstellung verschiedener Verfahren

vorgelegt von:

Student:	Cornelius Aurel Amzar
Matrikelnummer:	2316690
E-Mail:	amzar@informatik.uni-freiburg.de

Freiburg, den 05.02.2010

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Freiburg, den 05.02.2010

_____

Cornelius Amzar

Zusammenfassung. *Workflow Mining ist eine noch recht unbekannte Technik. Vielen ist nicht klar, wie ausgedehnt das Anwendungsgebiet ist und wie mächtig die Fähigkeiten des Verfahrens sind. Es gibt noch viele offene Fragen und Unstimmigkeiten, die von bisherigen Arbeiten nicht oder nur unvollkommen gelöst werden konnten. Diese Arbeit soll die Grundlagen erklären und einen Überblick über den aktuellen Stand der Forschung geben. Ein solcher Überblick fehlt bislang. Eine Auswahl von Arbeiten verschiedener Autoren wird zusammengefasst. Dazu zählen Ansätze für die Erkennung von Anomalien, impliziten Abhängigkeiten und Verfahren um das erzeugte Modell besser zu visualisieren. Die im Workflow Mining verwendeten Verfahren haben fast alle ihre Schwächen und so sollen hier Kriterien vorgestellt werden, anhand denen man die Ansätze vergleichen und ihre jeweiligen Vor- und Nachteile bewerten kann. Auch das Thema Sicherheit im Zusammenhang mit Workflow-Mining wird nicht ausgespart.*

Danksagung. Für die tolle Unterstützung und die vielen Tipps möchte ich mich bei meinem Betreuer, Dr. Rafael Accorsi, bedanken. Außerdem geht mein Dank an Michael Leukert der die Korrektur dieser Arbeit übernommen hat.

Inhalt

Abbildungsverzeichnis

Verzeichnis der Tabellen

1 Einführung

1.1 Motivation

Das Verfahren des *Workflow Mining* (auch *Process Mining* genannt) ist bereits seit ca. fünfzehn Jahren bekannt und wird sowohl in der Softwareentwicklung als auch im Geschäftsleben eingesetzt. Als Prozess bzw. Workflow bezeichnet man in diesem Zusammenhang jede Art von Ablauf. Das kann die (Online-)Bestellung von Waren, das Ausfüllen der Steuererklärung, die Produktion eines komplizierten Bauteils oder die Aktivitäten in einem Sozialen Netzwerk sein.

Eine *Log-Datei* ist – vereinfacht gesagt – eine Aufzeichnung von Informationen, wann wer welche Aufgabe ausgeführt hat. Der genaue Aufbau einer solchen Datei wird in Abschnitt 2.1 erläutert. Log-Dateien sind heute von praktisch allen Programmen und vielen Geschäftsabläufen verfügbar. Sie werden in der Regel von der verwendeten Software automatisch aufgezeichnet. Mit Hilfe eines geeigneten Verfahrens kann ein Modell des Prozesses aus den Log-Dateien rekonstruiert werden (Abbildung 1). Modelle von Prozessen werden heutzutage immer wichtiger, da die Abläufe immer komplizierter werden und man trotzdem den Überblick behalten muss. Dadurch kann man die Abläufe viel besser nachvollziehen und damit auch optimieren. Ein Einsatzgebiet ist beispielsweise das *Enterprise Resource Planning* (ERP), welches eingesetzt wird, um die Verwendung von Ressourcen wie Personal, Maschinen und Material zu optimieren [2]. Unter anderem bietet die Firma SAP eine (gleichnamige) Software zu diesem Zweck an. Weitere Einsatzgebiete sind *Customer Relationship Management* (CRM), *Business-to-Business* (B2B, Beziehungen zwischen zwei oder mehr Unternehmen) und *Software Configuration Management* (SCM). Letzteres beschäftigt sich mit den Abläufen bei der Softwareentwicklung, von der Planung über die Entwicklung bis zur Bereitstellung von Patches für das fertige Produkt.

Sinn eines Modells ist nicht nur das Redesign des Prozesses, sondern auch das bessere Verständnis der Abläufe – ein Bild sagt eben mehr als tausend Worte. Ein Redesign des Prozesses kann auch nur durchgeführt werden, wenn man einen Überblick über alle Abläufe und Abhängigkeiten hat. Da aber Geschäftsprozesse häufig je nach Nachfrage, Bedarf, etc. im Detail angepasst werden, kann der Überblick schnell verloren gehen. Manager werden sich für Details wie den häufigsten Arbeitsablauf, den minimalen/maximalen Durchsatz oder die Mitarbeiter, die an einem bestimmten Arbeitsablauf beteiligt sind, interessieren. Das Workflow Mining bietet nun eine Möglichkeit ein Modell des Prozesses zu erstellen. Dieses Modell kann dann – abhängig davon welche Daten in dem Log enthalten sind – genutzt werden, um den Managern die geforderten Informationen zu liefern. Anhand diesen Informationen kann nun der Ablauf überarbeitet (redesigned) bzw. angepasst werden.

Anfangs waren die Algorithmen nur für stark eingeschränkte und vereinfachte Log-Dateien zu gebrauchen, da sie z.B. nicht mit nebenläufigen Prozessen, Rauschen oder bestimmten Abhängigkeiten umgehen konnten und produzierten unter widrigen Umständen sehr unübersichtliche Modelle. Die manuelle Überwachung des Workflow Minings und die Nachbesserung des Modells von Hand war unabkömmlich. Die Forschung hat hier aber Fortschritte gemacht und für all diese Probleme geeignete Algorithmen zur Erkennung und zum Umgang mit solchen Logs entwickelt. Ein weiteres Einsatzgebiet ist

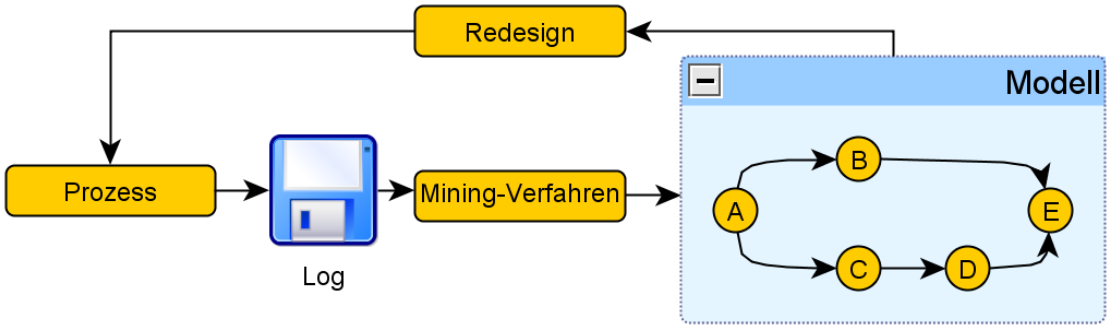


Abbildung 1: *Das Prinzip des Process Mining*

die Sicherheitsüberwachung. Hat man erst einmal ein Modell eines Prozesses, so kann man in Echtzeit prüfen, ob irgendwelche Aktivitäten vom Modell abweichen. Solche Aktivitäten können Angriffe darstellen. Diese Thematik wird in Abschnitt 6.2 genauer verfolgt.

Bislang fehlt ein Überblick über den aktuellen Stand des Workflow-Minings. Genau das soll nun in der vorliegenden Arbeit erarbeitet werden. Die Arbeit beschäftigt sich mit der Vorstellung und dem Vergleich unterschiedlichen Mining-Verfahren und den dadurch erzeugten Modellen. Außerdem wird eine Einführung in das Gebiet des Workflow Mining gegeben. Auch die Sicherheit wird in diesem Zusammenhang behandelt. Das Redesign des Prozesses und die Erstellung des Logs selbst sind jedoch nicht Thema dieser Arbeit.

1.2 Aufgabenstellung und Methodik

Es gibt inzwischen viele verschiedene Methoden, um aus einer Menge von aufgezeichneten Arbeitsabläufen ein aussagekräftiges und präzises Modell des zugrunde liegenden Prozesses zu erhalten. Dabei handelt es sich einerseits um grundlegende Algorithmen zum Workflow Mining, andererseits aber auch um spezielle Problemlösungen.

Diese Arbeit soll zuerst einmal eine Einführung in das Gebiet des Workflow Mining bieten und des Weiteren einen Überblick über die verschiedenen Verfahren, ihre Funktionsweise und zugrunde liegenden Modelle geben und die Vor- und Nachteile abwägen. Ein Vergleich soll zum Schluss zeigen, welcher Ansatz mit welchen Problemen umgehen kann und mit welchen nicht. Beim Vergleich werden unterschiedliche Kriterien angelegt. Zuerst einmal wird untersucht, welche Teilmenge von Workflows der Algorithmus überhaupt verarbeiten kann. Zu den Kriterien zählen hier also die Fähigkeit, Nebenläufigkeit, Schleifen, Anomalien/Rauschen und implizite Abhängigkeiten zu verarbeiten. Anschließend wird die Nutzerfreundlichkeit der Verfahren verglichen. Die Ansätze haben teils sehr viele Parameter, die einem praktischen Einsatz eher im Wege stehen. Die Möglichkeit, das erzeugte Modell zu vereinfachen, kommt immer dann zum Tragen, wenn komplexe Abläufe zu analysieren sind. Manche Algorithmen wurden nur theoretisch entwickelt und auf dem Papier getestet. Andere dagegen wurden mit sehr großen Log-Dateien konfrontiert. Der Umfang der Tests ist ebenfalls entscheidend für die Nutzerfreundlichkeit. Zum Schluss wird ein Überblick über Programme zum Workflow-Mining gegeben und die Frage untersucht, ob sich der Aufwand für spezielle Problemlö-

sungen überhaupt lohnt bzw. zu einer spürbaren Verbesserung des Modells führt. Die meisten Aufsätze, die hier zusammengefasst wurden, stammen aus den Jahren 2006-2008. Einige stammen auch von 2001 und 2004. Es handelt sich also um durchaus aktuelle Forschungsergebnisse.

1.3 Struktur der Arbeit

Abschnitt 2 schildert die gemeinsamen Grundlagen der Verfahren und den Aufbau von Log-Dateien. Die Zielsetzungen des Workflow Minings werden festgelegt. Dazu gehört auch die Frage, was überhaupt ein *gutes Modell* ausmacht. Nachdem drei verschiedene Möglichkeiten vorgestellt wurden, Modelle darzustellen, werden grundsätzliche Ansätze eingeführt und der am weitesten verbreitete Algorithmus, der als Grundlage für viele aktuelle Ansätze dient, vorgestellt.

In den Abschnitten 3 bis 7 werden einschlägige Forschungsarbeiten zusammengefasst und ihre Vor- und Nachteile dargelegt. Die Arbeiten reichen von völlig unterschiedlichen Ansätzen zur Gewinnung eines Modells aus dem Log, bis zu recht speziellen Problemlösungen. Im einzelnen sind es das statistische Workflow Mining von Weijters und Van der Aalst, das Abductive Workflow Mining von Buffett und Hamilton, die Erkennung von Abhängigkeiten und Anomalien in Log-Dateien von Bezerra und Wainer bzw. Van der Aalst und De Medeiros und die Aufteilung des Modells in Gruppen mithilfe von disjunktiven Workflow-Modellen von Greco et al.

Eine Gegenüberstellung der zuvor beschriebenen Ansätze findet sich in Abschnitt 8. Dort wird auch auf Programme eingegangen, mit denen man die vorgestellten Algorithmen simulieren kann. Außerdem wird ein abschließender Blick auf Sicherheitsaspekte geworfen. Im letzten Abschnitt findet sich schließlich eine Zusammenfassung der gesamten Arbeit und eine Auswertung der festgestellten Probleme.

2 Grundlagen

2.1 Grundbegriffe

Log-Dateien werden heute von vielen unterschiedlichen Systemen erzeugt. Dazu zählen sowohl Software-Systeme als auch Geschäftsprozesse. Hier soll der Aufbau einer typischen Log-Datei beschrieben werden. Eine Log-Datei besteht aus Einträgen, die einem bestimmten Fall (*Case*) eine Aufgabe (*Task*, manchmal auch *Aktivität* oder *Event Class*) zuordnen. Die Menge der *Events* ist das, was im Log aufgezeichnet wird. Jede Aufgabe wird zusammen mit dem dazugehörigen Fall gespeichert. In Abbildung 2 wäre die Anzahl der Events also 19, die der Tasks fünf. Die einzelnen Events müssen dabei aber nicht nacheinander abgearbeitet werden, sondern dies kann auch parallel geschehen. Dadurch sind die Einträge für die einzelnen Cases aber miteinander vermischt. Nun kann man aus diesem Log den Arbeitsablauf für einen einzelnen Fall ablesen. Diesen Arbeitsablauf bezeichnet man als *Trace* oder *Sequenz*.

Case	1	2	3	3	1	1	2	4	2	2	5	4	1	3	3	4	5	5	4
Task	A	A	A	B	B	C	C	A	B	D	A	C	D	C	D	B	E	D	D

Abbildung 2: *Beispiel-Logdatei. Das entsprechende Modell findet sich in Abbildung 3*

Dieses Log enthält fünf Traces, nämlich {*ABCD*, *ACBD*, *ABCD*, *ACBD*, *AED*}, d.h. für jeden der fünf Cases genau einen Trace. Manche Traces treten mehrfach auf, andere nur einfach. Zudem kann es – je nach Anwendung – für einen Case mehrere Traces geben, die nacheinander ausgeführt werden. Aus der Abfolge der Tasks in den Traces kann man bestimmte Schlüsse ziehen. So sieht man auf den ersten Blick, dass alle Traces mit *A* beginnen und mit *D* enden und dass *B* sowohl vor als auch nach *C* ausgeführt werden kann. Genau so funktioniert im Prinzip Workflow Mining. Durch die Analyse von Log-Dateien soll ein möglichst aussagekräftiges und vollständiges Modell erstellt werden, das den aufgezeichneten Prozess möglichst genau wiedergibt. Die Aussagekraft wird durch unnötige Zustände verschlechtert, die Vollständigkeit wird erreicht, wenn das Modell alle im Log enthaltenen Traces akzeptiert, d.h. es gibt einen Pfad durch das Modell, der dem jeweiligen Trace entspricht. Das Modell zu dem in Abbildung 2 gezeigten Log findet sich in Abbildung 3.

Als realitätsnahes Beispiel für einen Workflow kann man die Bestellung von Waren bei einem Online-Shop anführen. Bei jedem Vorgang (Case) legt ein Kunde gewisse Güter in seinen Warenkorb und führt bestimmte Schritte (Tasks) durch, je nachdem ob er ein neuer Kunde ist oder nicht und ob er per Kreditkarte oder Bankeinzug zahlt. Dabei muss sich ein bereits registrierter Kunde immer zuerst einloggen, bevor er die Bestellung abschicken kann, und ein Neukunde muss zuerst seine Adresse angeben. Diese Reihenfolge ist festgelegt und darf niemals verändert werden. Ein Trace ist nun der Bestellablauf für einen einzelnen Kunden und das Log enthält die Aktivitäten aller Kunden in der Reihenfolge, wie sie auftreten. Das Log wird also von einem zentralen Server aufgezeichnet und enthält nicht die Aktivitäten eines Kunden und dann die Aktivitäten des nächsten, sondern alle Aktivitäten sind miteinander vermischt.

Die Aufgabe des Workflow Minings wäre es nun, aus diesen Logs ein Modell wie in Abbildung 3 zu schaffen, das für alle Kunden zutrifft und daher den Ablauf bei der Bestellung in diesem Online-Shop widerspiegelt. Je nach Anwendung ist eine Einschränkung der Menge von Kunden sinnvoll, da sonst zu viele – völlig unterschiedliche – Traces das Modell sehr unübersichtlich und wenig hilfreich machen würden. Dies kann in den Implementierungen durch Filter über die Logs umgesetzt werden.

Das Log kann darüber hinaus auch Zeitstempel und personenbezogene Daten enthalten; dies kann zur Erkennung bestimmter Verhaltensweisen und zur Gewinnung von Informationen über die Dauer der Abläufe und das darin eingebundene Personal genutzt werden. Diese Zusatzinformationen sind aber für das allgemeine Workflow Mining nicht zwingend erforderlich und werden hier, soweit nicht anders erwähnt, nicht vorausgesetzt.

2.2 Modellvarianten

Die Algorithmen nutzen unterschiedliche Meta-Modelle für die Verarbeitung und Ausgabe. Häufig verwendet werden *Workflow-Netze* (WF-Netze), eine Teilmenge der Petri-Netze. Einfacher sind *Endliche Automaten* (Finite State Machine, FSM) und *Workflow Schemata*. Im Prinzip stellen sie alle eine spezielle Form von gerichteten Graphen dar. Die Knoten entsprechen den Tasks und die Kanten den gefundenen Übergängen zwischen den Tasks. Zusätzlich gibt es meist noch eine Kennzeichnung für AND/OR-Blöcke. Endliche Automaten und Workflow Schemata sind zwar weniger mächtig als ein WF-Netz, aber dafür einfacher zu gewinnen. Ein Nachteil ist allerdings, dass eine FSM keine Nebenläufigkeit modellieren kann, und ein Workflow Schema keine Schleifen. Von diesen Eigenheiten abgesehen ist es aber egal, welches Modell man verwendet. Die Modelle sind in der Regel übertragbar, es gibt Verfahren zur Transformation eines Meta-Modells in ein anderes. Im Folgenden werden kurz die Grundlagen zu diesen Modellen zusammengefasst. Genauere Beschreibungen finden sich in (Van der Aalst et al., 2003) für Workflow-Netze und in (Cook und Wolf, 1996) für die FSM. Die Workflow-Schemata werden zum Beispiel in (Greco et al, 2006) vorgestellt.

Eine zentrale Frage ist, was denn überhaupt ein „gutes“ Modell ausmacht. Das Modell soll natürlich alle im Log vorhandenen Sequenzen erfassen können. Das bedeutet, dass jeder Trace aus dem Log das Modell durchlaufen kann und dann in einem Endzustand stehen bleibt. Von jedem Task im Modell muss ein Übergang zum darauf folgenden Task im Trace vorhanden sein. Im Umkehrschluss sollten aber möglichst keine Traces akzeptiert werden, die nicht im Log enthalten sind. Dieser Thematik widmet sich besonders Abschnitt 7. Das Verhalten des Prozesses soll also bestmöglich abgebildet werden. Durch das Rauschen (zufällige Fehler in den Aufzeichnungen) und durch Varianten des Prozesses (kleine Abweichungen des Arbeitsablaufs) wird dies erschwert. Damit beschäftigen sich wiederum die Abschnitte 6.1 und 7. Diese Konformität reicht aber nicht, denn ein Modell sollte auch für einen Menschen gut verständlich sein und daher möglichst wenige nutzlose oder doppelte Zustände und Übergänge haben. Das bedeutet, dass der Algorithmus solche Zustände und Übergänge möglichst erkennen und selbstständig entfernen sollte ohne dass dadurch Traces nicht mehr akzeptiert werden. Die Algorithmen werden darin zwar immer besser, dennoch ist es manchmal noch nötig, dass ein Fachmann sich das Modell anschaut und eventuell durch das Verschmelzen von Zuständen vereinfacht. Ein Ansatz, der unnötigen Zuständen und Übergängen besondere Beachtung schenkt, findet sich in Abschnitt 5.

2.2.1 Workflow-Netze

Workflow-Netze basieren auf den sogenannten *Petri-Netzen*. Ein Petri-Netz besteht aus drei Mengen: *Places* (*P*), *Transitions (T*) und *Flow Relations* (*F*). Es gilt $P \cap T = \emptyset$ und $F \subseteq (P \times T) \cup (T \times P)$. Die Menge *F* stellt also die aus gerichteten Graphen gewohnten Kanten dar. Zwischen zwei Transitionen muss immer ein Place liegen. Eine Besonderheit der Petri-Netze sind die *Tokens.* Ein Token stellt die Bearbeitungsfolge eines Traces dar. Zu Beginn liegt nur ein einziges Token im Input-Place. Von hier geht also jede Aktivität aus. Eine Transition, bei der an jedem Eingangs-Place ein Token liegt, heißt *schaltbereit* oder *aktiviert*. Die Tokens werden dann mithilfe der Schaltregel (*Firing Rule*) von Place zu Place weitergegeben (genauer: sie werden verbraucht und dann in den Ausgangs-Places einer Transition neue erzeugt). Die Tasks werden durch die Transitionen *T* repräsentiert. Weitere Transitionen sind AND-Split, AND-Join, (X)OR-Split und (X)OR-Join. Die AND-Splits geben also zwei oder mehr Tokens aus während ein AND-Join zwei oder mehr Tokens annimmt und nur eines ausgibt. Die Verwendung von AND/OR-Blöcken ist nicht zwingend, aber übersichtlicher. Manche Autoren verzichten aus Gründen der Einfachheit darauf. Die Workflow-Netze unterscheiden sich von Petri-Netzen dadurch, dass es genau einen *Input-* und einen *Output-Place* gibt. Außerdem muss das Netz *stark zusammenhängend* sein. Das bedeutet, dass es für jedes Paar von Transitionen einen Pfad gibt, der die beiden Knoten verbindet. Da ein solches WF-Netz trotzdem noch unerwünschte Eigenschaften[1] haben kann, wird darüber hinaus die Zuverlässigkeit (*Soundness*) definiert. Ein WF-Netz ist *zuverlässig*, genau dann, wenn

- die Terminierung garantiert ist,
- bis zur Terminierung keine Tokens zurückgelassen werden,
- es keine unerreichbaren (toten) Tasks enthält.

Abbildung 3 zeigt ein solches zuverlässiges WF-Netz. Ein Token wird durch den Punkt dargestellt. Die Places entsprechen den Kreisen, Transitionen den Quadraten.

2.2.2 Workflow Schemata

Ein – verglichen mit WF-Netzen – einfacheres und weniger ausdrucksstarkes Modell ist das Workflow-Schema (Greco et al., 2006). Workflow-Schemata sind besonders für komplexe Prozesse mit vielen Aktivitäten geeignet. Ein Workflow Schema *WS* besteht aus einer Menge *A* von Aktivitäten (Tasks), einer Menge *E* von Kanten, einem Anfangszustand a_0, einer Menge von Endzuständen A_F und zwei Funktionen *Fork* und *Join*, welche die Art der Übergänge (AND, OR, XOR) festlegen. Diese beiden Funktionen haben also die Grundmenge $A - A_F$ bzw. $A - \{a_0\}$ und die Bildmenge {AND, OR, XOR}. Ein solches Schema ist zusammen mit einer Beschreibung der Zustände in Abbildung 6 dargestellt. Jedes Mal, wenn ein Workflow-Schema in einem Workflow-Management-System ausgeführt wird, wird eine Instanz $I = (A_I, E_I)$ erzeugt. Diese Instanz ist ein Subgraph des Schemas und enthält alle Aktivitäten, die in dem Trace ausgeführt werden und die deswegen ihre ausgehenden Kanten aktivieren. Zu beachten ist aber, dass die Ecken, zu denen diese Kanten führen, nicht zwingend ausgeführt werden müssen. Die Ausführung kann abgebrochen werden, weil in einem anderen Zweig des Graphen in der Zwischenzeit ein Endzustand erreicht wurde. In der Instanz in Abbildung 6 wird das durch die gestrichelten Linien angedeutet. Im Task *f* wird die Bonität des Benutzers geprüft

[1] Ein nicht zuverlässiges WF-Netz kann beispielsweise Deadlocks enthalten. Außerdem kann es Transitionen geben, die niemals erreicht werden können.

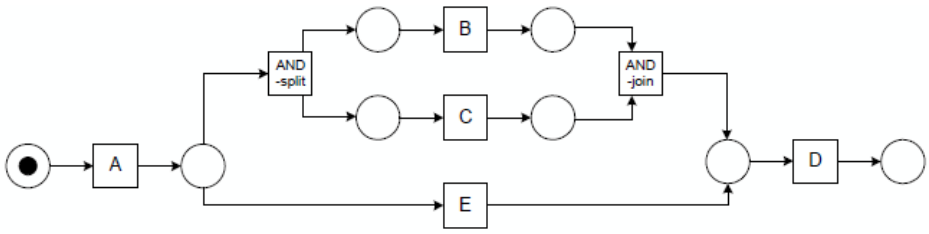


Abbildung 3: *Ein Modell (WF-Netz) für das Log in Abbildung 2*

und dann im Task *h* eine Fortführung der Bestellung unterbrochen. Deswegen wird die Kante zu *i* gar nicht erst ausgeführt, obwohl sie aktiviert ist. Ein Trace *s* ist konform zu einem Schema *WS* durch eine Instanz *I*, wenn *s* eine topologische Sortierung von *I* ist. Im Beispiel ist *abfcgh* konform zur Instanz, aber *afbcgh* und *abfjk* nicht. *s* ist konform zum gesamten Workflow-Schema, $s \models WS$ wenn es eine Instanz dieses Schemas gibt, so dass *s* zu dieser Instanz *I* konform ist.

In diesem Modell sind Schleifen nicht möglich, Will man einen Task mehrmals ausführen, so kann man dies erreichen, indem man dem gleichen Task unterschiedliche Bezeichner gibt.

2.2.3 Endliche Automaten

Ein nichtdeterministischer, endlicher Automat, engl. *Finite State Machine*, FSM, besteht aus Zuständen, Zustandsübergängen und Aktionen [3]. Der Begriff „endlich“ bezieht sich auf die Anzahl der Zustände. Die Zustandsübergänge entsprechen den Tasks. Die Zustände selbst haben keine Bedeutung, werden aber zur Unterscheidung nummeriert. In Abbildung 4 ist eine FSM dargestellt. Es gibt drei Tasks (Edit, Review und Checkin). Das Modell enthält alle Übergänge, die auch in den Traces auftreten. Das Problem, eine FSM aus einem Workflow-Log zu erzeugen, ist vergleichbar mit dem Problem, eine Grammatik für eine reguläre Sprache zu finden (Cook und Wolf, 1996). Dabei handelt es sich um ein klassisches Problem der Informatik, bei dem die Tasks den Buchstaben der Sprache, die Traces den Wörtern und das gesamte Log der Sprache entsprechen. Nun soll eine Grammatik, d.h. eine Menge von Regeln gefunden werden, die alle Wörter dieser Sprache beschreibt, aber keine Wörter, die nicht zur Sprache gehören.

Die Unfähigkeit der FSM, Nebenläufigkeit zu modellieren, liegt im Fehlen von AND-Bausteinen begründet. Alle Zustandsübergänge beschreiben immer ein XOR. Aus einem Zustand können nicht gleichzeitig zwei Übergänge verfolgt werden. Man kann dies aber umgehen, indem man Nebenläufigkeit während der Vorverarbeitung erkennt und dann für nebenläufige Prozesse getrennte FSMs berechnen lässt. Die endlichen Automaten wurden in frühen Arbeiten zum Thema Workflow Mining verwendet, weil ihre Erzeugung bereits in anderen Problemen der Informatik untersucht wurde. Sie verlieren aber aufgrund ihrer Einschränkungen an Bedeutung. In aktuellen Arbeiten wird meist ein WF-Netz oder ein WF-Schema eingesetzt.

2.3 Verfahren zur Modellrekonstruktion

In diesem Abschnitt soll zunächst einmal vorgestellt werden, welche Verfahren in der Vergangenheit verfolgt wurden, um aus einem Log ein Modell des Prozesses zu erzeugen. Anschließend wird der bewährte α-Algorithmus kurz vorgestellt. Er bildet die Grundlage vieler hier vorgestellter Arbeiten.

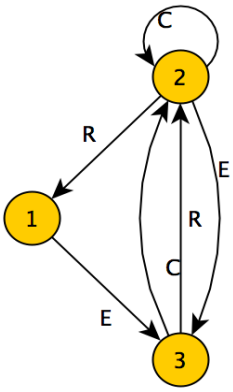


Abbildung 4: *Ein endlicher Automat*

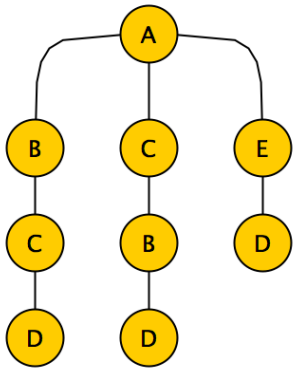


Abbildung 5: *Suffix-Baum für das Log in Abbildung 2*

2.3.1 Allgemeine Vorgehensweise

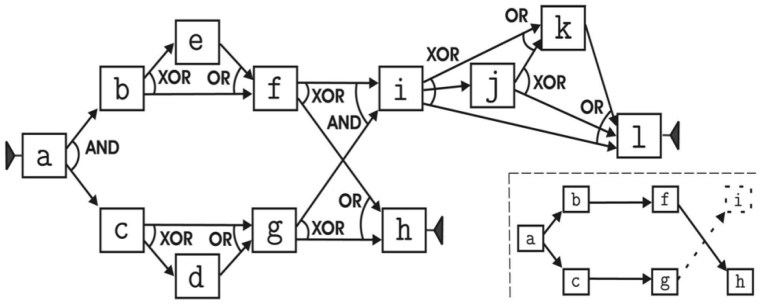
Wie in (Cook und Wolf, 1996) beschrieben, wurden bisher im wesentlichen drei verschiedene Ansätze verfolgt, um aus einer Log-Datei mit mehreren Traces ein passendes Modell des Prozesses zu gewinnen:

- mit künstlicher Intelligenz (Neuronales Netz),
- algorithmisch, d.h. durch reine Berechnungen,
- statistisch. d.h. durch Analyse der Häufigkeiten verschiedener Muster im Log.

Mischungen dieser Verfahren sind natürlich möglich. Diese bezeichnet man dann als *hybrid*. Sie werden zunehmend verwendet, da sie die Vorteile der beteiligen Ansätze vereinen können. Der Fokus dieser Arbeit liegt auf den beiden letztgenannten Ansätzen, da der Ansatz mit neuronalen Netzwerken sich aufgrund des erforderlichen Trainings als zu langsam erwiesen hat (Cook und Wolf, 1996). Ein algorithmischer Ansatz bedient sich zum Beispiel der Datenstruktur eines Suffix-Baums. Dies funktioniert wie folgt: Der Algorithmus erhält als Eingabe einen Array von Strings. Jeder String repräsentiert einen Trace im Log. Entscheidend ist nun, dass der Zustand dadurch festgelegt wird, welche zukünftigen Ereignisse aus diesem Zustand heraus erreicht werden können. Für das in Abbildung 2 gezeigte Log würde der Suffix-Baum aussehen wie in Abbildung 5 gezeigt. Zwei oder mehr solche Strings können sich ein Präfix teilen, sich aber im Suffix, d.h. in der „Zukunft“ unterscheiden. Die "Zukunft" wird dabei meist durch einen Parameter definiert, der angibt, wie weit der Algorithmus vorausschauen soll. Zwei Traces, die sich bisher unterschieden haben, aber in der Zukunft gleich verhalten, befinden sich in der gleichen Äquivalenzklasse. Diese Äquivalenzklassen repräsentieren nun die Zustände des Modells. Der Algorithmus erzeugt also den Suffix-Baum der Traces und leitet aus diesem Baum ein Modell ab.

Der sehr bekannte und vielfach überarbeitete α-Algorithmus wird unter anderem in (Van der Aalst et al., 2003) und (Van der Aalst und de Medeiros, 2004) beschrieben. Er ist ebenfalls algorithmischer Natur, funktioniert jedoch ohne eine spezielle Datenstruktur, sondern erstellt - basierend auf bestimmten Ordnungsrelationen - schrittweise aus den Log-Dateien direkt ein Modell.

Statistische Algorithmen berechnen aus den Traces eine Tabelle mit verschiedenen statistischen Werten. Beispielsweise, wie oft dieser Task insgesamt auftritt und wie oft er von einem anderen Task gefolgt wird bzw. wie oft ein anderer Task in der Sequenz



a	Erhalt der Bestellung	*g*	Auswertung des Produktionsplans
b	Authentifizierung des Kunden	*h*	Ablehnen der Bestellung
c	Ist das Produkt auf Lager?	*i*	Akzeptieren der Bestellung
d	Ist es extern verfügbar?	*j*	Schicke Mail an Zulieferer → schnellere Lieferung
e	Registrierung des Kunden	*k*	Rabatte abziehen
f	Prüfung der Bonität	*l*	Rechnung vorbereiten

Abbildung 6: *Oben links ein Workflow-Schema, rechts eine Instanz dieses Schemas; Unten eine Beschreibung der Zustände*

vor ihm steht. Daraus können dann auch Nebenläufigkeit, Schleifen und Entweder-Oder-Konstrukte abgelesen werden. So entsteht ein Modell.

Als Beispiel für einen hybriden Algorithmus kann man den Markov-Algorithmus heranziehen (Cook und Wolf, 1996). Er basiert auf den sogenannten Markov-Modellen. Der Algorithmus hat sowohl einen statistischen als auch einen algorithmischen Anteil. Zuerst wird eine einfache Häufigkeitstabelle erstellt, anschließend wird anhand dieser Tabelle ein gerichteter Graph erstellt. Dieser Graph ist aber noch kein gutes Modell. Es kann nämlich vorkommen, dass die Ecke zu viele Kanten besitzt, die Sequenzen repräsentieren, die eigentlich nicht erlaubt sind. Diese werden „entschärft“, indem die betreffende Ecke aufgeteilt wird. Der duale Graph des so erhaltenen Graphen entspricht dann einer *Finite State Machine* (siehe nächsten Abschnitt).

Vielfach bedient man sich einer Heuristik, wenn die Berechnungen zu aufwändig sind. Das bedeutet, man sucht nicht mehr nach *der* optimalen Lösung, sondern gibt sich mit einer Approximation zufrieden. Eine Heuristik bedeutet also einen Kompromiss zwischen der Laufzeit einer Problemlösung und seiner Güte [4].

Die Log-Dateien, die zum Test des Verfahrens benutzt werden, stammen in der Regel nicht von realen Prozessen, sondern von einem zuvor erstellten Modell. Dieses Modell wird dann so oft simuliert, bis genügend Traces vorliegen. Anschließend wird der Workflow-Mining-Algorithmus gestartet, um aus den Traces ein neues Modell zu erstellen. Nun kann man durch den Vergleich des alten Modells mit dem neuen Modell auf die Güte des Algorithmus schließen.

Vorab bekanntes Wissen, z.B. Start- und Endtasks über den zugrunde liegenden Prozess, kann den Mining-Vorgang unabhängig vom verwendeten Verfahren weiter beschleunigen. Dies wird aber in den hier besprochenen Ansätzen nicht vorausgesetzt.

2.3.2 Der α-Algorithmus

Der in (Van der Aalst et al., 2003) vorgestellte Algorithmus bildet heute die Basis vieler Verfahren, die ihn sowohl weiterentwickeln als auch darauf aufbauen und mit dem vom

α-Algorithmus erzeugten Modell weiterarbeiten. Das liegt nicht zuletzt daran, dass er viele Workflows problemlos erkennen kann und damit vielfältig einsetzbar ist. Eingesetzt wird das Modell der WF-Netze. Dieses ist zwar relativ kompliziert, kann dafür aber auch Nebenläufigkeit und Schleifen abbilden. Da viele der in diesem Aufsatz vorgestellten Papers auf den α-Algorithmus Bezug nehmen, soll er hier kurz eingeführt werden. Die hier vorgestellte Grundvariante des α-Algorithmus kann nicht mit Rauschen umgehen. Es wird also angenommen, dass das Log rauschfrei ist und ausreichend Daten enthält.

Der Algorithmus ist an sich algorithmischer Natur, er wertet die kausalen Abhängigkeiten zwischen den Tasks aus. Es werden jedoch nur direkte Abhängigkeiten beachtet, indirekte Abhängigkeiten werden ignoriert. Zur Unterscheidung zwischen direkten und indirekten Abhängigkeiten siehe Abschnitt 5. Zuerst berechnet der Algorithmus verschiedene Relationen, welche die Abhängigkeit bzw. Unabhängigkeit der Tasks untereinander widerspiegeln. Es gibt vier Relationen über der Menge der Tasks.

- $a > b$, wenn a der direkte Vorgänger von b ist.
- $a \rightarrow b$, wenn a der direkte Vorgänger von b ist, aber umgekehrt b nie vor a auftritt. Dies ist daher die direkte kausale Abhängigkeit.
- $a\#b$, wenn weder $a > b$ noch $b > a$, das heißt es gibt keine direkten kausalen Abhängigkeiten. Die beiden Tasks sind wahrscheinlich unabhängig und es ist unwahrscheinlich, dass sie parallel ablaufen.
- $a \| b$, wenn a und b parallel (nebenläufig) ablaufen. Das bedeutet, dass die beiden Tasks in jeder beliebigen Reihenfolge aufeinander folgen.

Diese Relationen basieren alle auf der Relation >, d.h. sie können alle aus dieser einen Relation gewonnen werden. Der Algorithmus erhält als Eingabe nur das Workflow-Log W, welches sich aus Traces σ zusammensetzt. Daraus berechnet er dann schrittweise das entsprechende WF-Netz. Es werden zunächst mehrere Mengen gebildet. Die genauen Definitionen finden sich in (3) – (8). Die Menge T_W der Transitionen ergibt sich direkt aus der Menge der Tasks. Nun werden zuerst die Input- und die Output-Transitionen, T_I und T_O, bestimmt und anschließend vier Mengen berechnet: X_W, Y_W, P_W und F_W.

$$T_W = \{t \in T \mid \exists \sigma \in W \; t \in \sigma\} \tag{1}$$

$$T_I = \{t \in T \mid \exists \sigma \in W \; t = first(\sigma)\} \tag{2}$$

$$T_I = \{t \in T \mid \exists \sigma \in W \; t = last(\sigma)\} \tag{3}$$

$$X_W = \{(A, B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B} a \rightarrow b \wedge \forall_{a_1, a_2 \in A} a_1 \# a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \# b_2\} \tag{4}$$

$$Y_W = \{(A, B) \in X_W \mid \forall_{(A', B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A, B) = (A', B')\} \tag{5}$$

$$P_W = \{p(A, B) \mid (A, B) \in Y_W\} \cup \{i_W, o_W\} \tag{6}$$

$$F_W = \{(a, p(A, B)) \mid (A, B) \in Y_W \wedge a \in A\} \cup \{(P(A, B), b) \mid (A, B) \in Y_W \wedge b \in B\} \cup \{(i_W, t) \mid t \in T_I\} \cup \{(t, o_W) \mid t \in T_O\} \tag{7}$$

$$\alpha(W) = (P_W, T_W, F_W) \tag{8}$$

Wichtig ist die Unterscheidung zwischen Places und Transitionen. Es darf nur einen Input- und Output-Place geben, aber mehrere Transitionen, die Tokens aus diesen Places

konsumieren bzw. neue Tokens erzeugen. Außerdem ist zu beachten, dass das Verfahren in dieser Form keine gesonderten Transitionen für AND- bzw. OR-Blöcke erzeugt. Dies kann man aber durch eine kleine Erweiterung erreichen. Dadurch wird das Modell verständlicher. Die Mengen X_w und Y_W dienen dazu, unnötige Places zu vermeiden. Diese würden sonst bei AND/OR-Splits und -Joins entstehen. Die Menge der Places entspricht P_W. Sie wird aus der Menge der Transitionen zusammen mit Input- und Outputplace bestimmt. p(A, B) sagt aus, dass zwischen den Transitionen A und B ein Place liegt. Aus X_W und Y_W wird die Menge F_W der Kanten berechnet. Nun haben wir alle drei für ein Workflow-Netz nötigen Mengen, nämlich die Mengen der Places, der Transitionen und der Kanten. Diese werden vom Algorithmus zurückgeliefert.

3 Statistisches Workflow Mining

In (Weijters und Van der Aalst, 2001) wird ein heuristischer (statistischer) Ansatz vorgestellt. Er hat aber wenig mit dem später entwickelten α-Algorithmus der gleichen Autoren (Van der Aalst et al., 2003) zu tun. Der α-Algorithmus nutzt zur Berechnung keine Tabellen mehr. Der statistische Ansatz soll in erster Linie die Inflexibilität bisheriger Ansätze beseitigen und nutzt als Modell die WF-Netze. Die Inflexibilität zeigte sich beim Umgang mit Rauschen, bei (kurzen) Schleifen und bei Nebenläufigkeit. Die Verwendung von WF-Netzen wird mit der besseren Darstellung, insbesondere von Nebenläufigkeit, begründet. Ziel ist ein Algorithmus, der (i) alle Event-Sequenzen erkennt, die im Log auftreten, (ii) dessen Modell so wenige Sequenzen wie möglich enthält (Komplexität!), (iii) nebenläufiges Verhalten erkennt und (iv) so einfach und kompakt wie möglich ist. Die zuverlässige Erkennung von Nebenläufigkeit ist ein weiteres Ziel, der Ansatz soll dies mit dem Einbau von AND/OR-Splits bzw. Joins im Modell repräsentieren.

3.1 Aufbau der Häufigkeitstabellen für die Tasks

Der Algorithmus setzt sich aus drei Teilen zusammen. Zuerst wird für jeden Task eine Tabelle erstellt, die Häufigkeitsangaben enthält. Dabei wird zwischen direkten und indirekten Vorgängern/Nachfolgern unterschieden. Ein indirekter Nachfolger B tritt zwar nach dem Task A auf, aber dazwischen dürfen beliebig viele andere Tasks liegen. Außerdem enthalten die Tabellen eine Metrik, die nach einer Formel die Stärke der kausalen Abhängigkeit zwischen zwei Tasks ausdrückt. Die Tabelle für Task A enthält also folgende Spalten:

1. Die absolute Häufigkeit aller Tasks, unabhängig von A (Notation: $\#B$),
2. Die Häufigkeit, mit der A einen direkten Vorgänger B hat (Notation: $B < A$),
3. Die Häufigkeit, mit der A direkt von einem Task B gefolgt wird (Notation: $A > B$),
4. Die Häufigkeit, mit der A einen direkten oder indirekten Vorgänger B hat, wobei aber A dazwischen nicht wieder auftritt (Notation: $B <<< A$),
5. Die Häufigkeit, mit der ein Task A direkt oder indirekt von einem Task B gefolgt wird. Auch hier darf aber A nicht dazwischen wieder auftreten (Notation: $A <<< B$),
6. Eine Metrik, welche die Stärke der kausalen Abhängigkeit zwischen Task A und einem anderen Task B anzeigt (Notation: $A \rightarrow B$).

Die Stärke der kausalen Abhängigkeit zwischen zwei Tasks nimmt mit der Anzahl der dazwischen liegenden Tasks exponentiell mit Faktor δ^n ab. Der Parameter $0 > \delta \geq 1$ regelt also die Stärke der kausalen Abhängigkeit bei zunehmender Entfernung zweier Tasks. In den späteren Experimenten wurde $\delta = 0{,}8$ gesetzt. Bei einem Wert von $\delta = 1$ ergibt sich keine Entfernungsabnahme der Metrik, bei $\delta = 0$ funktioniert die Metrik nicht mehr. Nach Berechnung der Tabellen für jeden Task wird aus den Tabellen ein Graph gewonnen. Hierzu wird zunächst folgende Formel eingesetzt.

$$IF((A \rightarrow B \geq N)\ AND\ (A > B \geq \sigma)\ AND\ (B < A \leq \sigma))\ THEN < A, B > \in T \qquad (9)$$

N ist ein Parameter und widerspiegelt die erwartete Stärke des Rauschens. Standardmäßig wählt man $N = 0{,}05$. Die erste Bedingung sagt aus, dass die kausale Abhängigkeit größer als das Rauschen sein sollte. In der zweiten und dritten Bedingung kommt ein Häufigkeits-Schwellwert σ vor. Er sorgt dafür, dass das Rauschen nicht den Algorithmus beeinflussen kann. σ entspricht der Häufigkeit, die eine Sequenz in allen Traces mindestens haben sollte, damit sie zuverlässig genug ist.

$$\sigma = 1 + \mathrm{Round}(N \cdot (|L|/|T|)) \tag{10}$$

σ wird automatisch aus N berechnet, um die Anzahl der Parameter zu begrenzen. $|L|$ ist die Anzahl der Traces, $|T|$ die Anzahl der unterschiedlichen Tasks. Die beiden Bedingungen sagen also aus, dass B häufig einen Vorgänger A hat, aber umgekehrt A nicht sehr häufig nach B kommt. Treffen alle drei Bedingungen zu, so wird eine Kante im Abhängigkeitsgraphen von Task A zu Task B gezeichnet.

3.2 Erkennung von Rekursionen und Schleifen

In dieser Form kann der Algorithmus jedoch keine Rekursionen und (kurze) Schleifen erkennen. Eine Rekursion äußert sich darin, dass ein Task A sowohl Ursache als auch Folge von sich selbst ist. Rekursionen kann man daran erkennen, dass die Häufigkeit von $A<A$ und $A>A$ relativ groß und gleich ist. Das gleiche gilt für $A <<< A$ und $A >>> A$. Schleifen wiederum können daran erkannt werden, dass ein Task A sowohl Ursache als auch Folge eines anderen Tasks B ist. Dies äußert sich dann in der Tabelle darin, dass die Häufigkeiten $A<B$ und $B>A$ groß und etwa gleich sind. Gleiches gilt auch für $A <<< B$ und $B >>> A$.

Aus diesen beiden Erkenntnissen werden daher zwei weitere Bedingungen entwickelt. Dadurch kann der Algorithmus auch Schleifen und Rekursionen erkennen.

$$\begin{aligned} &IF\,((A \rightarrow A \approx 0) \wedge (A < A + A > A \;>\; 0{,}5 \cdot \#A) \wedge (A < A - A > A \approx 0)) \\ &THEN < A, A > \in T \end{aligned} \tag{11}$$

$$\begin{aligned} &IF\,((A \rightarrow A \approx 0) \wedge (A > B \geq \sigma) \wedge (B < A \approx A > B) \\ &\wedge (A >>> B \geqslant 0{,}4 \# A) \wedge (B <<< A \approx A >>> B))\,THEN < A, B > \in T \end{aligned} \tag{12}$$

Damit das Rauschen keine Probleme bereitet, wird statt des Gleichheitszeichens ein Approximationszeichen verwendet. Nun muss nur noch aus dem entstandenen Graphen ein vollständiges WF-Netz erzeugt werden.

Es fehlen dazu nur noch die Typen der Nebenläufigkeit, also AND/OR-Split und -Join. Die Festlegung geschieht mit einfachen heuristischen Überlegungen. Will man den Typ eines Übergangs der Form A, B AND/OR C bestimmen, so schaut man in der Tabelle nach den Werten von $C<B$ und $B>C$. Wir erwarten bei einem AND-Split bei beiden einen positiven Wert, da sowohl die Reihenfolge B, C als auch die Reihenfolge C, B auftreten kann. Bei einem OR-Split können diese beiden Muster aber nicht auftreten, sondern entweder nur B oder nur C. In dem Aufsatz wird der Pseudocode für einen Algorithmus angegeben, der diese Aufgabe erledigt.

3.3 Auswertung

Zum Test wurden sechs unterschiedlich komplizierte WF-Netze (13 bis 16 Tasks) benutzt. Im Vergleich mit aktuellen Verfahren ist diese Zahl gering. Viele neuere Ansätze werden mit bis zu 100 Tasks getestet. Alle Modelle enthalten nebenläufige Prozesse und Schleifen. Aus diesen Modellen wurden dann zufällig drei Logs mit jeweils 1000 Sequenzen erzeugt: (i) ohne Rauschen, (ii) mit 5% Rauschen und (iii) mit 10% Rauschen. Dazu wurde erst die gewünschte Anzahl aus den zuvor erzeugten Traces ausgewählt, und dann wurde das Rauschen hinzugefügt. Das geschah durch eine der folgenden Aktionen:

- Entfernen des Kopfs eines Traces,
- Entfernen des Endes eines Traces,
- Entfernen eines Elements aus dem Körper,
- Vertauschen zweier Elemente.

Bei den Löschoperationen wurde mindestens ein Element und maximal ein Drittel der Sequenz entfernt. Der Algorithmus erzeugte aus den fehlerfreien Logs perfekte Modelle, die auch mit dem ursprünglichen Modell identisch waren. Das gleiche gilt für die Logs mit 5% Rauschen. Bei den Logs mit 10% Rauschen war eines von sechs Modellen fehlerhaft. Dies lag an dem zu niedrigen Schwellwert $\sigma = 5$ in Regel (1). Wird der Rauschfaktor N auf 0,10 angehoben, so erhöht sich σ automatisch auf 9, und das erzeugte Modell ist fehlerfrei.

Der Algorithmus scheint also gut mit Rauschen umgehen zu können. Es ist lediglich eine Abschätzung notwendig, wie stark die Logs verrauscht sind. Fraglich ist allerdings, wie es bei zufällig verrauschten Logs aussieht. Diese Frage wird in dem Artikel von Van der Aalst und Weijters nicht beantwortet. Eine Abschätzung des Rauschfaktors ist zwar möglich, aber Fehlerfreiheit kann nur bei einem genau richtigen Rauschfaktor garantiert werden. Beim zweiten Parameter, δ, kann in der Regel der Standardwert von 0,8 verwendet werden. Selbst wenn sieben Tasks dazwischen liegen, wird die kausale Abhängigkeit noch mit 0,2 gewichtet. Kleinere Werte als 0,8 machen wenig Sinn, größere verschieben die Grenze geringfügig nach oben. Da man aber in der Regel immer möglichst viele Abhängigkeiten erhalten möchte, kann man den Wert entweder auf 0,8 lassen oder auf 0,9 setzen. Ein genaueres Tuning ist sinnlos, da man dann genau wissen muss, wie weit die Abhängigkeiten voneinander entfernt sind.

4 Abductive Workflow Mining

In (Buffett und Hamilton, 2008) wird ein neuer Ansatz vorgestellt, der das zu erreichen versucht, was bisherige Ansätze nicht geschafft haben. Der α-Algorithmus (Van der Aalst et al., 2003) und seine Weiterentwicklung zum α++-Algorithmus[2] (Wen et al., 2006) erzeugen zwar einfache WF-Netze, die auch gut zu visualisieren sind, aber sie opfern dafür Genauigkeit. Auf der anderen Seite die Ansätze Petrify (Van der Aalst et al., 2006) und Region Miner (Van Dongen et al., 2006), die zwar schnell und präzise sind, aber unter Umständen sehr große und komplizierte Modelle erzeugen. Das Abductive Workflow Mining soll beides erreichen: Schnelligkeit, Präzision und gleichzeitig eine Verringerung unnötiger Zustände im Modell. Hier wird also ein Verfahren vorgestellt, das ein Modell auf diejenigen Zustände und Übergänge reduziert, die von Interesse sind. Im Folgenden wird zuerst der Begriff *Abduktion* und anschließend das verwendete Verfahren eingeführt. Zuletzt wird im Vergleich mit dem α-Algorithmus die Leistungsfähigkeit des Verfahrens ermittelt.

4.1 Was ist Abduktion?

Die Abduktion ist eine Variante des logischen Schließens. Es wird jedoch – im Gegensatz zur Induktion oder Deduktion – der *Grund* für eine bestimmte Beobachtung gesucht. Wir suchen also nach einer Hypothese, die – wenn sie wahr ist – notwendigerweise eine bestimmte Beobachtung erklärt. Das ist besonders interessant, wenn man eine Beobachtung macht und wissen möchte, was dieses Ereignis hervorruft. Gegeben eine Beobachtung B und eine Menge von Regeln R. Wir versuchen, Fakten zu finden, die, hinzugefügt zu R, notwendigerweise B implizieren. Dieses Verfahren wird in der Fehlerdiagnose bei komplexen Systemen eingesetzt. Die Beobachtung ist hier eine Fehlfunktion und die Menge R repräsentiert die Betriebs-Eigenschaften des Systems. Das abduktive Schließen wird dann benutzt, um zu bestimmen, welche Art von Aktivität diesen Fehler notwendigerweise verursacht haben kann.

Im Workflow Mining wird dieses Verfahren ganz ähnlich eingesetzt. Das System ist hier der Prozess bzw. sein Modell, und die Beobachtungen entsprechen potenziell gefährlichen Tasks. Diese sollten nicht einfach frei ausgeführt werden, sondern es muss dafür einen triftigen Grund geben. Das abduktive Schließen kann dann dazu eingesetzt werden, das kritische Verhalten zu finden. Der abduktive Workflow enthält nur Transitionen und Zustände, die Ursache für ein bestimmtes Ereignis sind. Dadurch kann das zu verarbeitende Modell erheblich verkleinert werden.

4.2 Einsatz der Abduktion im Workflow Mining

Gegeben eine Menge von Tasks T, eine Menge von Traces C [3] und ein Modell W. Eine Teilmenge W' von W ist die Beobachtung. Meist wird es sich dabei nur um einen einzelnen Task handeln, doch theoretisch kann es auch jedes beliebige (Teil-)Modell sein. Der abduktive Workflow W_a ist nun derjenige Teil von W, der – sofern ausgeführt – W' impliziert. Das heißt: jede Aktivität, die in W_a beobachtet wird, *muss* als Folge eine Wirkung in W' haben.

---

[2] siehe Abschnitt 5

[3] die Abkürzung C rührt von der etwas abweichenden Nomenklatur her und stammt vom Begriff „Case“

Ein Trace $c \in C$ wird *konsistent* zu einem beliebigen Workflow W genannt, wenn es einen gültigen Pfad durch W gibt, der ein (nicht leerer) Teilstring von c ist. Darüber hinaus ist jede Sequenz t von Tasks konsistent zu W, wenn es einen (nicht leeren) Pfad durch W gibt, der t entspricht. Hierbei ist zu beachten, dass W nicht notwendigerweise auf C basieren muss – es kann auf einer beliebigen Grundmenge von Traces basieren. Die Menge C unterteilen wir nun in die Menge der konsistenten Cases C^+ und die Menge der nicht konsistenten Traces C^-. Nun definieren wir eine Sequenz von Tasks t_a als abduktiven Pfad, wenn folgendes gilt:

- t_a ist Teilstring eines beliebigen $c^+ \in C^+$,
- für jedes $c^- \in C^-$ gilt (1) t_a ist nicht Teilstring eines $c^- \in C^-$ (2) Wenn ein Pfad $t_{W'}$ durch W' eine Teilmenge von t_a ist, dann gibt es keinen String S, so dass durch die Ersetzung von $t_{W'}$ mit S t_a zu einem Teilstring von c^- wird.

Um das zu verstehen nun ein einfaches Beispiel mit zwei Traces *ABDE* und *ACDF*. W' bestehe einfach nur aus B. Dadurch gehört *ABDE* zu den positiven und *ACDF* zu den negativen Cases. *ABD* kann nicht als abduktiver Pfad betrachtet werden. Es ist zwar Teilmenge von *ABDE* und auch keine Teilmenge von ACDF, aber $t_{W'} = B$ kann durch $S = C$ ersetzt werden, so dass daraus *ACD* entsteht. *ABDE*, *BDE*, *DE* und sogar *E* kommen jedoch als abduktive Pfade in Frage.

Ein abduktiver Workflow besteht nun ausschließlich aus den abduktiven Pfaden in C für W'. Um die Berechnung zu vereinfachen, wird folgende Annahme aufgestellt. Wenn wir nach Substrings von C^+ suchen, betrachten wir nur solche, die einen Pfad durch W' enthalten. Das ist aus zwei Gründen eine attraktive Annahme. Erstens haben Ereignisse in der Nähe der in Frage stehenden Aktivität mit hoher Wahrscheinlichkeit eine kausale Beziehung zu dieser Aktivität und zweitens ist es relativ einfach, diese Substrings zu identifizieren.

Das Verfahren arbeitet wie folgt. (1) Aus einem Log wird ein Modell erstellt. (2) W' wird festgelegt, in der Regel ein einzelner Task. (3) Es werden Pfade a_i und a_o gesucht, so dass ein Pfad durch W' direkt auf a_i folgt und direkt an a_o anschließt. (4) Es werden weitere Pfade gesucht, die a_i und a_o enthalten, aber nicht W'. Wird so ein Pfad gefunden, so werden a_i und a_o neu belegt oder es wird eine längere Sequenz vorhergehender oder folgender Aktivitäten ausgewählt. Wenn nicht, dann hat man einen abduktiven Workflow in C für W' gefunden. Das Ganze wird immer weiter wiederholt, bis keine weiteren abduktiven Pfade mehr gefunden werden.

4.3 Auswertung

Die Autoren haben ihr Verfahren mit fünf beim Programm ProM (siehe Abschnitt 7.2) mitgelieferten Beispiel-Logs getestet. Dabei haben sie als Mining-Algorithmus sowohl ein nicht weiter definiertes, eigenes Verfahren, als auch den α-Algorithmus verwendet. Aus dem erstellten Modell wurde dann ein einzelner Task ausgewählt, zu dem der abduktive Workflow gesucht werden soll. Die durchschnittliche Zahl von Transitionen und Kanten ist für beide Mining-Algorithmen in Tabelle 1 wiedergegeben. Man sieht klar, dass der Algorithmus der Autoren nicht so leistungsfähig ist wie der α-Algorithmus. Er enthält für das gleiche Log ja mehr als dreizehn mal so viele Transitionen. Man sollte sich daher nicht davon beeindrucken lassen, sondern vor allem dem Ergebnis mit dem α-Algorithmus Beachtung schenken. Der abduktive Workflow enthält nur knapp halb so viele Transitionen und Kanten wie der ursprüngliche Workflow. Man sollte aber beden-

ken, dass er auch nicht die gleichen Informationen enthält, sondern eben nur diejenigen Transitionen, die einen Einfluss auf den gewählten Teil-Workflow bzw. Task haben.

	Algorithmus der Autoren		**α-Algorithmus**	
	Orginal	abduktiver WF	Orginal	abduktiver WF
Transitionen	156,2	37,2	12,0	5,6
Kanten	318,6	77,0	33,2	15,6

Tabelle 1: *Durchschnittliche Größe des ursprünglichen und des abduktiven Workflows für beide Mining-Verfahren*

Nichtsdestotrotz ist der vorgestellte Ansatz sinnvoll. In komplexen Modellen ist es eine Herausforderung, alle Ursachen für einen gewählten Task oder einen Teil des Modells zu erfassen. Das spielt insbesondere bei der Analyse von Problemen und somit bei der Optimierung eine große Rolle. Durch das abduktive Workflow-Mining wird diese Arbeit automatisiert. Es sind jedoch noch viel umfangreichere Tests vonnöten, die in dem hier vorgestellten Aufsatz leider zu kurz kommen. Es muss klar sein, welcher Task bzw. welches Teilmodell als Ziel des Algorithmus gewählt wurde. Denn von dessen Position hängt natürlich die Größe des resultierenden abduktiven Workflows ab.

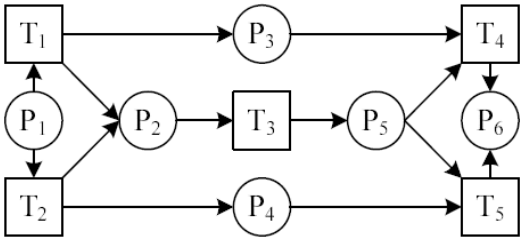


Abbildung 7: *Ein WF-Netz mit expliziten und impliziten Abhängigkeiten*

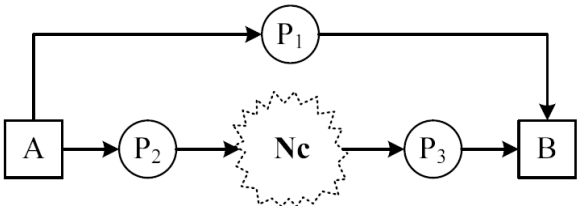


Abbildung 8: *Ein Modell mit impliziter Abhängigkeit*

5 Erkennung von (impliziten) Abhängigkeiten

In Prozessen gibt es viele kausale Abhängigkeiten. Häufig muss oder darf ein Task erst ausgeführt werden, wenn zuvor ein anderer Task ausgeführt wurde. Man unterscheidet zwischen zwei Arten von kausalen Abhängigkeiten: *explizite* und *implizite*. Explizite Abhängigkeiten widerspiegeln direkte kausale Abhängigkeiten zwischen Tasks. Sie können von allen bisherigen Algorithmen erkannt werden. Anders sieht es bei impliziten Abhängigkeiten aus, die indirekte kausale Abhängigkeiten widerspiegeln. In Abbildung 7 wird der Unterschied zwischen expliziten und impliziten Abhängigkeiten deutlich. Place P_2 und seine umgebenden Kanten repräsentieren eine explizite Abhängigkeit zwischen T_1 und T_3 bzw. zwischen T_2 und T_3. P_3 dagegen repräsentiert eine implizite Abhängigkeit zwischen T_1 und T_4. Das Log, das von dem in Abbildung 7 gezeigten Modell erzeugt wurde, kann nur zwei verschiedene Traces enthalten, $T_2T_3T_5$ und $T_1T_3T_4$. Würde man den α-Algorithmus auf dieses Log anwenden, so würden P_3 und P_4 und die dazugehörigen Kanten einfach fehlen. Dieses Modell wäre dann nicht verhaltensäquivalent mit dem ursprünglichen Modell, denn es würde auch Traces $T_1T_3T_5$ und $T_2T_3T_4$ erzeugen. Die Arbeit von (Wen et al., 2006) widmet sich dieser Problematik und entwickelt den bekannten α-Algorithmus weiter (zum α++-Algorithmus). Die einzige Einschränkung ist, dass das WF-Netz keine Schleifen der Länge eins enthalten darf, also Tasks, die immer wieder sich selbst aufrufen.

Die Autoren betrachten zuerst die einfachste Form eines Netzwerks mit impliziten Abhängigkeiten, nämlich Abbildung 8. Wenn es eine implizite Abhängigkeit zwischen A und B gibt, dann kann es keinen Pfad geben, so dass B direkt auf A folgt. Im Gegenteil muss mindestens ein Task zwischen den beiden liegen. Diese Rolle übernimmt das Netzwerk N_c. Es enthält mindestens einen Task. Alle komplizierteren Fälle lassen sich von diesem einfachen Fall ableiten. Gibt es keine Tasks, die mit P_1, P_2 oder P_3 verbunden sind, so wird P_1 zu einem impliziten Place, der keinerlei Einfluss auf das Verhalten des Modells hat. Jeder Mining-Algorithmus sollte es daher vermeiden, solche impliziten Places zu erzeugen. In Abbildung 9 sind alle sieben *zuverlässigen* WF-Netze dargestellt, die sich auf den Eingangs- und Ausgangs-Task von P_1, die Eingangs-Tasks von P_2 und die Ausgangs-Tasks von P_3 beziehen. Im Folgenden werden diese sieben WF-Netze in drei Theoremen behandelt. Für WF-Netze (a) wird P_1 nicht im Modell enthalten sein (Theorem 3), für (b) und (g) könnte P_1 durch zwei oder mehr Places ersetzt werden (Theorem 1). Für (c) und (e) wird die Kante (P_1, B) ausgelassen und für (d) und (f) die Kante (A, P_1) (Theorem 2).

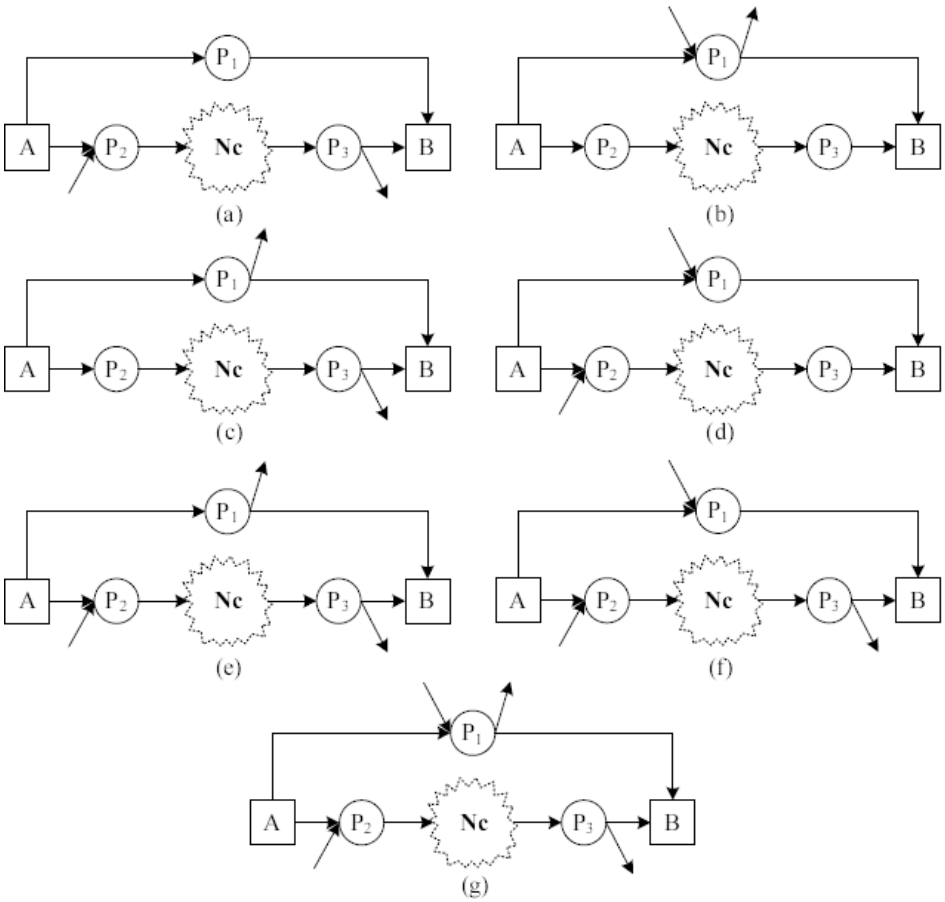


Abbildung 9: *Zuverlässige WF-Netze mit impliziter Abhängigkeit*

5.1 Funktionsweise

Der α++-Algorithmus baut auf dem α+-Algorithmus (de Medeiros et al., 2004) auf und erweitert dessen Ordnungsrelationen. Hier seien die verwendeten Ordnungsrelationen in umgangssprachlicher Form wiedergegeben.

- $a \triangle b$ genau dann, wenn es je eine Sequenz *aba* gibt,
- $a > b$ genau dann, wenn *a* direkt vor *b* auftritt,
- $a \rightarrow b$ genau dann, wenn *a* direkt vor *b* auftritt, aber umgekehrt *b* nicht direkt vor *a,*
- $a \# b$ genau dann, wenn weder *a* vor *b* noch *b* vor *a* auftritt,
- $a \| b$ genau dann, wenn sowohl *a* vor *b* als auch *b* vor *a* auftritt, aber nicht in der Form $a \triangle b$ oder $b \triangle a$. Das impliziert Parallelität,
- $a \triangleleft b$ genau dann, wenn zwischen *a* und *b* ein XOR-Split liegt,
- $a \triangleright b$ genau dann, wenn zwischen *a* und *b* ein XOR-Join liegt,
- $a \gg b$ genau dann, wenn *b* nicht direkt auf *a* folgt, sondern andere Tasks dazwischen liegen,
- $a \succ b$ genau dann, wenn $a \rightarrow b$ oder $a \gg b$,
- $a \mapsto b$ genau dann, wenn *b* implizit von *a* abhängt oder es eine implizite Abhängigkeit von *b* nach *a* gibt.

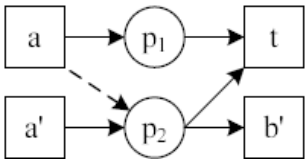


Abbildung 10: *Beweis von Theorem 1*

Die Definitionen von $\triangle$, > und # wurden vom α bzw. α+-Algorithmus direkt übernommen, $\rightarrow$ und $\|$ wurden leicht verändert. Die restlichen fünf Relationen sind komplett neu. Nun können wir aus dem Log $\{T_1T_3T_4, T_2T_3T_5\}$ welches dem Modell in Abbildung 7 entspricht, die folgenden Relationen herleiten.

$$T_1 \triangleright T_2, T_4 \triangleleft T_5, T_1 \gg T_5, T_2 \gg T_4, T_1 \mapsto T_5, T_2 \mapsto T_4 \tag{13}$$

Es liegen also zwei implizite Abhängigkeiten vor. In diesem Beispiel mag die Erkennung einfach sein, im Allgemeinen ist es aber aufwändig, aus den erkannten impliziten Abhängigkeiten die richtigen Konsequenzen zu ziehen, damit das Modell keine impliziten Places enthält und keine unerwünschten Sequenzen erzeugen kann.

Die Eingangsmenge $\bullet T_S$ und Ausgangsmenge $T_S\bullet$ einer Menge von Tasks T_S besteht aus denjenigen Tasks t für die es ein $t' \in T_S$ gibt, so dass gilt: $t \rightarrow t'$ (Eingangsmenge) bzw. $t' \rightarrow t$ (Ausgangsmenge). Wenn wir das oben angegebene Log für das Modell in Abbildung 7 betrachten, ergeben sich beispielsweise folgende Mengen:

$$\bullet\{T_4\}=\{T_3\}, \bullet\{T_5\}=\{T_3\}, \{T_1, T_2\}\bullet=\{T_3\} \tag{14}$$

Im Folgenden werden nun die drei Theoreme zusammengefasst, die von den Autoren zur Erkennung und Anpassung des Modells vorgestellt wurden. Dabei werden die Mengen X_w und Y_W des α-Algorithmus (siehe Seite 16) neu definiert, so dass eine Erkennung von impliziten Abhängigkeiten des jeweiligen Typs möglich wird. Dadurch werden implizite Places erkannt und – je nachdem, welcher Fall vorliegt – Places zusammengelegt oder komplett ausgelassen. Sind diese beiden Mengen berechnet, wird der α-Algorithmus ganz normal fortgesetzt.

Theorem 1. In den WF-Netzen in Abbildung 9 der Form (b) oder (g) haben wir jeweils Tasks, die in P_1 hinein zeigen und aus P_1 Tokens erhalten. Wie bereits gesagt, werden solche Places durch zwei oder mehr Places ersetzt. Angenommen, wir haben für jeden Task $t \in T_W$ zwei weitere Tasks $t_1, t_2 \in T_W$ für die gilt $t_1 \rightarrow t$, $t_2 \rightarrow t$, $t_1 \# t_2$ und $\{t_1\}\bullet \neq \{t_2\}\bullet$. Dann berechnen wir

$$X_W=\{(A,B) \mid A,B \subseteq T_W \wedge t \in B \wedge \forall_{a\in A} \forall_{b\in B} a \rightarrow b \wedge \forall_{a_1,a_2\in A} a_1 \# a_2 \wedge \forall_{b_1,b_2\in B} b_1 \# b_2\} \tag{15}$$

$$Y_W=\{(A,B) \in X_W \mid \forall_{(A',B')\in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B)=(A',B')\} \tag{16}$$

Man beachte, dass der einzige Unterschied zum bisherigen α-Algorithmus in der Bedingung $t \in B$ in Formel (15) liegt. Jetzt gilt für jedes (A_1, B_1), $(A_2, B_2) \in Y_W$ und für jeden Task $a \in A_1$ und $a \notin A_2$ die Folgerung

$$\nexists_{a'\in A_2} a' \parallel a \vee a' \succ a \implies \forall_{b'\in B_2-\{a\}\bullet} a \mapsto b' \tag{17}$$

Zum Beweis nutzen wir Abbildung 10. Es gilt offensichtlich $a \rightarrow t$ .Task t hat zwei Eingangs-Places, p_1 und p_2. Um aktiviert zu werden, muss sich in beiden Places ein Token befinden. Wenn a' von a direkt oder indirekt gefolgt werden kann, so muss a' bereits ausgeführt worden sein. Sind a und a' parallel, werden beide gleichzeitig ausgeführt. Wird a aber vor a' ausgeführt, so kann t nicht direkt auf a folgen und wir haben einen Widerspruch. In diesem Fall muss es also eine Verbindung zwischen a und p_2 geben (gestrichelte Linie). Theorem 1 sagt aus, dass, wenn es einen Place gibt, der zwei aufeinander folgende Tasks verbindet und der letztere der Beiden mehr als einen Eingang hat, dann kann es immer passieren, dass der spätere Task direkt nach dem ersten ausgeführt wird.

Theorem 2. Bei bei den WF-Netzen in Abbildung 9(c) bis (f) wird eine der Kanten ausgelassen. Hat p_1 zwei Eingänge, so wird die Kante von A nach p_1 ausgelassen. Hat p_1 dagegen zwei Ausgänge, so wird die Kante von p_1 nach B ausgelassen. Es wird also unterschieden zwischen einem Split und einem Join. Wir betrachten also die folgenden zwei Fälle
(1) Für den Split. Für jedes $t \in T_W$, für das es zwei Tasks t_1 und t_2 gibt, so dass $t \rightarrow t_1$, $t \rightarrow t_2$ und $t_1 \| t_2$, berechnen wir

$$X_W = \{ X \subseteq T_W | \forall_{e \in X} t \rightarrow x \wedge \forall_{x_1, x_2 \in X} x_1 \# x_2 \} \tag{18}$$

$$Y_W = \{ X \in X_W | \forall_{X' \in X_W} X \subseteq X' \Rightarrow X = X' \} \tag{19}$$

Dann gilt für jede Menge von Tasks $Y \in Y_W$

$$\begin{array}{c} \forall_{a,b,\in T_W} a \lhd b \wedge \exists_{y \in Y}) (y \parallel b \vee y \succ b) \wedge \\ \nexists_{y \in Y}) (y \parallel a \vee y \succ a) \wedge t \gg a \implies t \mapsto a \end{array} \tag{20}$$

(2) Für den Join. Für jedes $t \in T_W$, für das es zwei Tasks t_1 und t_2 gibt, so dass $t_1 \rightarrow t$, $t_2 \rightarrow t$ und $t_1 \| t_2$, berechnen wir

$$X_W = \{ X \subseteq T_W | \forall_{e \in X} x \rightarrow t \wedge \forall_{x_1, x_2 \in X} x_1 \# x_2 \} \tag{21}$$

Die Menge Y_W ist für beide Fälle gleich. Die Folgerungen unterscheiden sich nur in der Ordnung der Elemente.

$$\begin{array}{c} \forall_{a,b,\in T_W} a \rhd b \wedge \exists_{y \in Y}) (b \parallel y \vee b \succ y) \wedge \\ \nexists_{y \in Y}) (a \parallel y \vee a \succ y) \wedge a \gg t \implies a \mapsto t \end{array} \tag{22}$$

Zum Beweis nutzen wir Abbildung 11. Nachdem t ausgeführt wurde, liegen zwei Tokens in p_1 und p_2. Wegen $t \gg a$ muss es möglich sein, dass ein Token in p_3 liegt und a aktiviert ist. Wird in diesem Moment y_2 oder y_3 ausgeführt, so wird a deaktiviert (weil nur ein aktivierter Task zugleich feuern darf). In diesem Fall wird letztendlich b ausge-

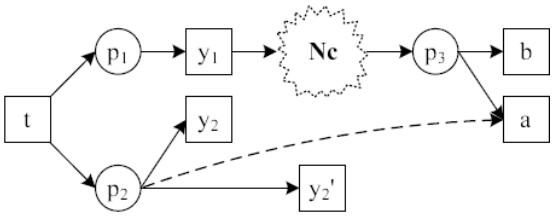


***Abbildung 11**: Beweis von Theorem 2*

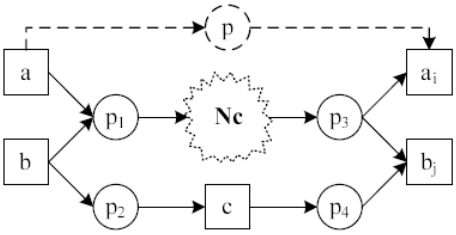


***Abbildung 12:** Beweis von Theorem 3*

führt. Auf der anderen Seite werden y_2 und y_3 deaktiviert, wenn a ausgeführt wird. Dann bleibt aber ein Token in p_2 zurück. Das ist ein Widerspruch. Es muss also eine Verbindung von p_2 nach a geben (gestrichelte Linie). Daraus folgt, dass es eine implizite Abhängigkeit zwischen t und a geben muss.

Theorem 2 zeigt, dass ein Task, der ein Token aus einem von mehreren parallelen Zweigen verbraucht, zusammen mit seinen parallelen Tasks auch Tokens aus anderen Zweigen konsumieren *muss*.

Theorem 3. Nun kommen wir zum schwierigsten Fall, nämlich WF-Netze wie das in Abbildung 9(a) dargestellte. In diesem Fall soll der Place p_1 nicht im erzeugten Modell enthalten sein. Für zwei Tasks $a, b \in T_W$ und $a \triangleright b$ berechnen wir

$$\begin{aligned} X_W = \{(A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge (\forall_{a_i \in A} a \gg a_i \wedge b \not\gg a_i) \wedge \\ (\forall_{b_j \in B} a \not\gg b_j \wedge b \gg b_j) \wedge \forall_{a_i \in A} \exists_{b_j \in B} b_j \lhd a_i \wedge \forall_{b_j \in B} \exists_{a_i \in A} a_i \lhd b_j \wedge \\ \forall_{a_i, a_j \in A} a_i \parallel a_j \wedge \forall_{b_i, b_j \in B} b_i \parallel b_j\} \end{aligned} \tag{23}$$

$$Y_W = \{(A,B) \in X_W \mid \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B')\} \tag{24}$$

Für jedes $(A, B) \in Y_W$ berechnen wir

$$A' = \{t \in T_W \mid t \notin A \wedge \exists_{b_j \in B} b_j \lhd t \wedge \exists_{a_i \in A} a_i \succ t\} \tag{25}$$

$$B' = \{t \in T_W \mid t \notin B \wedge \exists_{a_i \in A} a_i \lhd t \wedge \exists_{b_j \in B} bj \succ t\} \tag{26}$$

Dann gilt:

$$\forall_{a_i \in A} \bullet \{a_i\} \subseteq \bullet B \cup \bullet B' \implies a \mapsto a_i \tag{27}$$

$$\forall_{b_j \in B} \bullet \{b_j\} \subseteq \bullet A \cup \bullet A' \implies b \mapsto b_j \tag{28}$$

Zum Beweis betrachte man das WF-Netz in Abbildung 12. Wird a ausgeführt, so befindet sich ein Token in p_1. Nach einer gewissen Zeit befindet sich ein Token in p_3. Dadurch ist a_i aktiviert ($a \gg a_i$), b_j aber immer noch deaktiviert. Wird jedoch b ausgeführt, so werden nach einer bestimmten Zeit sowohl a_i als auch b_j aktiviert. Weil aber b kein indirekter Nachfolger von a_i ist, liegt darin ein Widerspruch. Task a_i kann nur aktiviert werden, wenn a ausgeführt wurde. Es gibt also eine implizite Abhängigkeit zwischen a

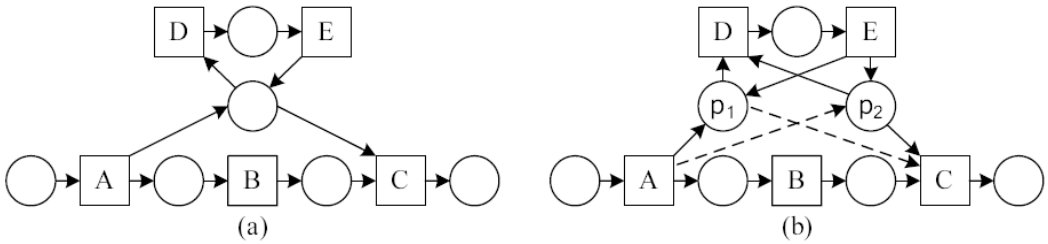


Abbildung 13: *Ein Beispiel für die Anwendung von Theorem 1*

und a_i. Wegen $\bullet\{a_i\} \subseteq \bullet\{b_j\}$ muss der gestrichelt gezeichnete Teil des Graphen eingefügt werden. Es gibt also eine implizite Abhängigkeit zwischen *a* und a_i.

Theorem 3 sichert zu, dass wenn zwei exklusive Tasks, d.h. Tasks, die in einem XOR liegen, zu unterschiedlichen Teilen paralleler Zweige führen und diese beiden Teile mit ihren Tasks bestimmte Bedingungen erfüllen, so ist das erzeugte WF-Netz immer noch zuverlässig.

Damit haben wir nun für jede Form impliziter Abhängigkeit ein Werkzeug, um sie zu erkennen und die Erzeugung impliziter Places zu verhindern.

5.2 Auswertung

Die Autoren zeigen in ihrem Aufsatz an mehreren Beispielen, dass die drei Theoreme ausreichen, um implizite Abhängigkeiten in Workflow Logs zu finden und Modelle zu erzeugen, die keine impliziten Places enthalten. Die Funktionsweise der drei Theoreme wurde anhand der drei oben vorgestellten WF-Netze bewiesen. Um ein anfängliches Modell zu erzeugen wird der α+-Algorithmus genutzt. Dann zeigen Wen et al., welche Kanten und Places durch die Theoreme als implizite Abhängigkeiten erkannt wurden und wie der Algorithmus darauf reagiert. Zum Verständnis sei hier exemplarisch die Anwendung von Theorem 1 geschildert. Das Log in Abbildung 13(a) zeigt das ursprüngliche WF-Netz, aus dem das Log erzeugt wurde. Das Modell kann die Traces {*ABC*, *ADEC*, *AC*} erzeugen. Durch reine Anwendung des α-Algorithmus erhält man das Modell in Abbildung 13(b), ausgenommen der gestrichelten Kanten. Dieses Modell kann aber keinen Trace {*AC*} erzeugen. Zwischen *A* und *C* liegt eine implizite Abhängigkeit, die durch Theorem 1 erkannt wird. Dadurch können die beiden Places p_1 und p_2 vereinigt werden. Allerdings ist wird an anderer Stelle erklärt, dass es keinen direkten Übergang zwischen zwei Tasks geben darf, zwischen denen eine implizite Abhängigkeit besteht. In diesem Fall ist aber ein Übergang zwischen *A* und *C* möglich. Die Autoren widersprechen sich also selbst. Es ist nicht klar, wieso das überhaupt eine implizite Abhängigkeit ist, denn es gibt eine direkte Abhängigkeit zwischen den beiden Tasks. Demnach sollte diese Abhängigkeit eigentlich vom normalen α-Algorithmus erkannt werden.

Die Erkennung von impliziten Abhängigkeiten ist allerdings relativ aufwändig und hat daher ein schlechtes Kosten-Nutzen-Verhältnis. Die verwendeten zehn Relationen wirken sich mit Sicherheit negativ auf die Laufzeit aus, wenn man bedenkt, dass sie für jedes einzelne Paar von Tasks berechnet werden müssen und Prozesse mit dutzenden Tasks der Realität entsprechen. Der Algorithmus wurde von Wen et al. allerdings nur unzureichend getestet – so liegen auch keine Erkenntnisse zur Laufzeit vor. Ein Modell

mit impliziten Abhängigkeiten ist lediglich etwas größer und komplexer als das entsprechende Modell ohne implizite Abhängigkeiten. Allerdings kann es vorkommen, dass eine andere Menge von Traces akzeptiert wird, wenn die implizite Abhängigkeit nicht erkannt wurde.

Sicher sind implizite Places bei der Ausführung des Modells unerwünscht, aber wenn es lediglich um das Verständnis des Prozesses geht sind sie wenig hinderlich, da sie das Verhalten des Prozesses nicht verändern. Die manuelle Erkennung impliziter Abhängigkeiten ist nur bei entsprechend kleinen Modellen möglich. Bei Prozessen mit sehr vielen Tasks, wie sie in der praktischen Anwendung häufig auftreten, ist eine manuelle Erkennung jedoch nicht möglich. Aufgrund der fehlenden Aussagen zur Laufzeit kann man nur abschätzen, dass der Algorithmus von Wen et al. wohl im Average Case länger brauchen wird als der ursprüngliche α-Algorithmus.

6 Erkennung von Rauschen und Anomalien

In praktisch jedem Workflow-Log aus einem realen Prozess tritt Rauschen auf, d.h. man kann sich nicht zu 100% auf die im Log enthaltenen Daten verlassen. Außerdem variieren die einzelnen Ausführungen eines Prozesses oftmals leicht, z.B. durch unterschiedliche Mitarbeiter. Solche Varianten sind – je nach gewünschter Information – nicht von Interesse. Diese beiden Faktoren wurden bisher meist nur durch einen Schwellwert über die Häufigkeit eines Traces herausgefiltert. Es ist aber falsch, dass seltene Ereignisse und Sequenzen automatisch Rauschen darstellen müssen. Es könnte sich dabei zum Beispiel auch um einen Angriff, eine gefährliche Fehlfunktion oder den Ausbruch einer Krankheit handeln, die erkannt werden sollten. Deshalb sprechen wir nun von Anomalien. In diesem Zusammenhang wird in Abschnitt 6.1 ein Ansatz vorgestellt um solche Anomalien vor bzw. während der Erzeugung eines Modells zu erkennen. Dazu werden verschiedene Heuristiken vorgestellt, um anhand der Traces Anomalien zu erkennen, sodass sie aus dem Log herausgefiltert werden können. Abschnitt 6.2 widmet sich der Erkennung von Anomalien anhand des Modells, das heißt das Modell selbst ist fehlerfrei und enthält keine Anomalien. Es sollen dagegen anhand des Modells in Echtzeit Fehler bzw. Angriffe in sicherheitskritischen Anwendungen gefunden werden.

6.1 Erkennung vor der Modellerstellung

In (Bezerra und Wainer, 2008a) werden drei verschiedene Heuristiken zur Erkennung von Anomalien vorgestellt. Die Erkennung geschieht nicht einfach nur doch einen Schwellwert wie bei anderen Arbeiten, sondern es wird eine Metrik dafür entwickelt, wie stark ein Trace das bestehende Modell verändert. Das ist möglich durch den Einsatz des inkrementellen Mining-Algorithmus von Wainer et al. (2005). Dabei wird das Modell zuerst nur für einen einzigen Trace erstellt und dann Trace für Trace verfeinert. Ein Trace, der eine Anomalie oder Rauschen enthält wird in der Regel keine Instanz des Modells sein und somit muss das Modell verändert werden, um auch diesen Trace zu akzeptieren. Ein Modell ist ein gerichteter Graph, besteht also aus Ecken (Vertices) und Kanten (Edges). Die Knoten können dabei sowohl Tasks als auch AND/OR Konstrukte (Split/Join) sein. Wird das Modell geändert, so kommt eine bestimmte Anzahl neuer Ecken hinzu. Die Kosten der Änderung (*Inclusion Cost*) können daher wie folgt dargestellt werden.

$$IC(M_O, M_N) = |V(M_N)| - |V(M_O)| \tag{29}$$

M_O ist dabei das bisherige Modell und M_N das Neue. Der erste Algorithmus (Sampling) beruht auf einem Satz von Traces, der sich bei jedem Test ändert. Für jeden zu prüfenden Trace wird immer ein neues, positives Testmuster S mit 50% der einzigartigen[4] Traces im Log gebildet. Dabei ist zu beachten, dass der aktuell zu prüfende Trace natürlich aus diesem Muster herausgenommen werden muss. Dann wird aus diesem Satz ein Modell erzeugt und anschließend geprüft, ob der zu prüfende Trace dieses Modell erfüllt. Erfüllt er es nicht, so zählt er als Anomalie. S darf dabei weder zu groß noch zu

[4] „einzigartig“ bedeutet, dass mehrfach auftretende, gleiche Traces ausgenommen sind.

klein sein. Ist *S* zu groß, so ist die Wahrscheinlichkeit groß, dass das Modell von einer Anomalie kompromittiert wird. Auf der anderen Seite führt eine zu kleine Menge *S* dazu, dass ein ungenaues Modell erzeugt wird, weil zu wenige "normale" Traces enthalten sind. Der Anteil von 50% stellt einen guten Kompromiss dar.

Der zweite Ansatz macht von den oben definierten Einschluss-Kosten Gebrauch. Er wird iterativ genannt, weil er solange einen Trace aus einer Menge auswählt, bis alle Traces geprüft wurden. Die Menge der Traces wird aufgeteilt in häufige (*F*) und seltene Traces (*C*). *F* enthält alle Traces, die mehr als 10% aller Traces ausmachen. Die Traces in *C* sind möglicherweise Anomalien. Anschließend gibt die Funkion *select*(*C*, *F*) den Trace mit den größten Einschlusskosten aus *C* oder Null zurück. Sind diese Kosten größer als 2, so zählt dieser Trace als Anomalie und wird aus *C* entfernt. Die Funktion *select*(*C*, *F*) wird so lange aufgerufen bis sie Null zurück gibt oder ein Trace gefunden wurde, dessen Kosten kleiner gleich 2 sind. Die Menge *F* ist nur deshalb Parameter der Funktion, weil aus ihr das „normale“ Modell berechnet wird, anhand dessen man die Einschluss-Kosten bestimmt. Der Schwellwert wird auf 2 festgelegt, weil eben auch „normale“ Traces gewisse Kosten verursachen können. Dieser Wert wurde durch Versuche festgelegt und stellt die 3. Quartile aller Einschlusskosten von 150 garantiert rauschfreien Logs dar.

Der dritte Ansatz funktioniert im Prinzip genauso, wählt aber alle Anomalien in einem einzigen Schritt aus. Er läuft schneller als der iterative Ansatz, weil alle Anomalien in einer einzigen Schleife gefunden werden und keine aufwändige Funktion zur Auswahl nötig ist. Aufgrund des variablen Parameters wird er Threshold-Algorithmus genannt.

Die Algorithmen sollten möglichst viele Anomalien erkennen (*True Positive*) aber gleichzeitig möglichst keine „normalen“ Einträge als Anomalien klassifizieren (*False Positive*). Die Autoren erzeugten ein Log aus einem vorliegenden Modell und verändern dieses Log dann. Es wurden z.B. AND-Blöcke in OR-Blöcke umgewandelt oder einfach Tasks vertauscht. Dabei gibt es einfache und doppelte Anomalien. Bei doppelten Anomalien handelt es sich um einfache Anomalien, die an zwei Stellen auftreten. Bei der Auswertung wurden dann Logs mit einer, zwei oder drei einfachen oder doppelten Anomalien eingesetzt.

	Sampling	***Threshold***	***Iterativ***
Average Case	4,896 s	9,016 s	16,275 s
Worst Case	466,422 s	670,625 s	1145,16 s

Tabelle 2: *Laufzeiten der drei Ansätze*

	Sampling			***Iterativ***			***Threshold***		
	ACC	***TPR***	***FPR***	***ACC***	***TPR***	***FPR***	***ACC***	***TPR***	***FPR***
Einfach	90,56%	98,85%	11,50%	80,15%	93,08%	23,13%	84,66%	88,72%	16,37%
Doppelt	81,34%	49,23%	10,53%	80,15%	93,08%	23,13%	84,66%	88,72%	16,37%
Alle	85,96%	74,04%	11,01%	80,15%	93,08%	23,13%	84,66%	88,72%	16,37%

Tabelle 3: *Vergleich der Genauigkeit und Fehlerrate*

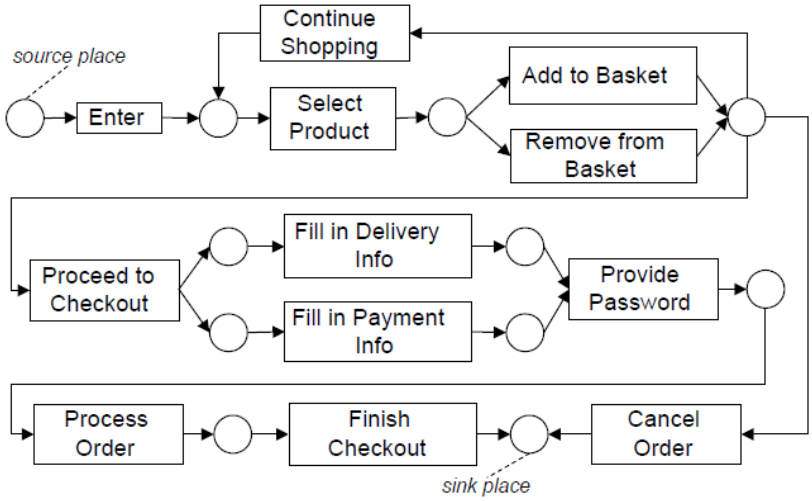


Abbildung 14: *Erzeugtes Modell des Shoppingsystems*

In Tabelle 2 sind die Laufzeiten der Algorithmen eingetragen, in Tabelle 3 die Genauigkeit und Fehlerraten. Dabei steht ACC für den Anteil der Traces, die korrekt klassifiziert wurden, d.h. die Genauigkeit (Accuracy), TPR für *True Positive Rate* und FPR für *False Positive Rate*. Wie man sieht ist von den drei Ansätzen der Erste (Sampling) in vielen Fällen zu bevorzugen. Er ist schnell und hat eine gute Leistung. Probleme hat er nur, bei doppelten Anomalien. In diesem Fall ist der Threshold-Ansatz besser geeignet, da er genauer und schneller als der iterative Ansatz ist. Der iterative Ansatz zeigt in allen Fällen eine schlechte Leistung und ist der langsamste. Er kann daher vernachlässigt werden (Bezerra und Wainer, 2008b).

6.2 Konformitätsprüfung anhand eines Modells

Eine andere Perspektive wird in (Van der Aalst und de Medeiros, 2004) vorgestellt. Hat man bereits ein fertiges, funktionsfähiges Modell, z.B. von einem Shoppingsystem, so möchte man unerwünschtes Verhalten feststellen. Das wäre zum Beispiel der Fall, wenn jemand versucht die Bestellung abzuschicken ohne ein Passwort einzugeben. Um das Modell zu erzeugen wird der α-Algorithmus benutzt. Er erhält als Eingabe eine Menge W_{OK} von geprüften Traces, die das „normale“ Verhalten vollständig wiedergeben. Aus diesen Traces erzeugt das Algorithmus im Beispiel das Modell in Abbildung 14.

Nun kann man alle neuen Traces, d.h. Sessionabläufe von Benutzern, prüfen, indem man das sogenannte „*Token Game*“ spielt. Unerwünschte Traces entsprechen nicht einer möglichen Schaltsequenz (wiederholte Anwendung der *Firing Rule*). Das heißt, die Tokens bleiben irgendwo im Netz „hängen“. Ein Beispiel für einen solchen, unerwünschten Trace wäre {*Enter*, *Select Product*, *Remove from Basket*, *Proceed to Checkout*, *Fill in Payment Info* , *Fill In Delivery Info*, *Process Order*, *Finish Checkout*}. Problematisch ist, dass der Task *Provide Password fehlt.* Das Verfahren erkennt fehlerhafte Traces daran, dass sie sich nicht vom Modell erzeugen lassen. Mit dieser Methode lässt sich auch feststellen, wo der Trace vom normalen Verhalten abkommt. Es ist auch eine Echtzeit-Überwachung von Traces möglich.

Es fehlt lediglich eine Möglichkeit, beliebig lange Schleifen im Modell zu verhindern. Man möchte ja zum Beispiel verhindern, dass ein Benutzer beliebig viele Anmel-

deversuche unternehmen kann. Diese Bedingung kann nicht vom α-Algorithmus erkannt werden, da sie nicht in den Traces enthalten ist. Sie muss nachträglich im Modell festgelegt werden. Das bedeutet es muss ein Zähler eingebaut werden, der die Anzahl der misslungenen Versuche aufzeichnet.

Natürlich kann das „Token Game“ auch ohne vorherige Anwendung des α-Algorithmus auf einem fertigen Modell angewendet werden. Der α-Algorithmus ist aber trotzdem sinnvoll, da er das regelmäßige Aktualisieren des Modells ohne großen Arbeitsaufwand erlaubt. Dadurch bleibt das System flexibel. Und es können relativ schnell neue Verhaltensweisen in das Modell eingepflegt werden.

Manchmal ist jedoch nur ein Teil des Modells sicherheitskritisch. Im obigen Beispiel wäre das z.B., dass der Task *Provide Password* den Task *Process Order* auslöst (kausale Abhängigkeit). Diese Abhängigkeit kann zusammen mit dem oben vorgestellten Ansatz überprüft werden. Man nutzt jetzt aber kein vollständiges Modell mehr, sondern ein Teilmodell, das nur die kausale Abhängigkeit *ProvidePassword → Process Order* enthält. So lässt sich jeder unerwünschte Trace identifizieren. Man benötigt dazu nicht mehr ein vollständiges Log, sondern es reicht, dass die Tasks vorkommen, die in diesem Teilmuster auftreten. Bei der Sicherheitsüberprüfung vieler Sessions in Echtzeit dürfte der Geschwindigkeitsvorteil nicht unerheblich sein.

6.3 Auswertung

Durch die beiden hier vorgestellten Methoden kann man einerseits Anomalien in einem Log herausfiltern, damit sie nicht das Modell unnötig verkomplizieren bzw. verfälschen. Der Algorithmus arbeitet zwar nicht zu 100% genau, aber der Großteil der Anomalien wird herausgefiltert. Natürlich ist damit kein sicherheitskritisches Modell realisierbar, aber im Zweifelsfall kann das Modell immer noch von einem Menschen auf fehlende Zustände (durch *false positives*) und übermäßige Komplexität (durch Anomalien) geprüft werden. In jedem Fall wird einem menschlichen Kontrolleur sehr viel Arbeit abgenommen, denn Rauschen ist für einen Menschen aufgrund seiner zufälligen Natur schwer erkennbar.

Andererseits kann man mithilfe eines bestehenden fehlerfreien Modells unerwünschtes Verhalten wie Angriffe und Sicherheitsrisiken in Echtzeit erkennen und damit eine Kontrollstruktur für alle möglichen sicherheitsrelavanten Systeme aufbauen. Dieses Verfahren arbeitet perfekt, sofern das zugrunde liegende Modell keine Angriffsmöglichkeiten enthält. Alle Anomalien, die nicht im Modell enthalten sind, werden erkannt. Indem man sich auf die sicherheitskritischen Abhängigkeiten des Modells beschränkt kann man eine große Anzahl Anfragen sehr schnell überprüfen.

Zusammen bieten diese beiden Ansätze ein mächtiges Werkzeug um Systeme zu sichern. Dazu wird zuerst − zum Beispiel durch überwachte Testläufe − ein Log erzeugt, auf welches dann die Anomalieerkennung angesetzt wird. Da es sich um einen überwachten Test handelt, sind wenige Anomalien zu erwarten. Anschließend wird auf dem Anomalien-bereinigten Log der α-Algorithmus angewendet um ein Modell zu erzeugen. Nun kann das System in den laufenden Betrieb übergehen. Angriffe werden zuverlässig erkannt. Änderungen am Modell, z.B. neue Funktionen lassen sich einfach einbinden, indem man einen Trace generiert, der diese neue Funktion enthält und anschließend den α-Algorithmus auf diesen Trace und die bisherigen Traces anwendet. Auf den Begriff der Sicherheit im Kontext des Workflow Mining wird in Abschnitt 8.3 eingegangen.

7 Disjunktives Workflow-Mining

Ein vollständiges Modell muss nicht zwingend mit dem durch das Log beschriebenen Prozess übereinstimmen. Es kann nämlich Muster geben, die zwar nicht im Log auftreten, aber dennoch vom Modell akzeptiert werden. Deswegen sollte man nicht nur auf die Vollständigkeit eines Modells achten, sondern auch auf die Zuverlässigkeit oder Minimalität. Die Zuverlässigkeit wird bestimmt durch den Anteil der möglichen Traces, die im Log enthalten sind und vom Modell akzeptiert werden. Je mehr „fremde“ Traces das Modell akzeptiert, desto unzuverlässiger ist es. Man beachte, dass der hier verwendete Begriff der Zuverlässigkeit nichts mit dem gleichen Begriff im Zusammenhang mit WF-Netzen zu tun hat. Bei den WF-Netzen bezeichnet er nur eine Eigenschaft des Modells, im hier verwendeten Zusammenhang dagegen eine Eigenschaft des Modells in Bezug auf das Log.

1	*acbgfh*	5	*abfcgikl*	9	*abefcgil*	13	*abcfdgikl*
2	*abfcgh*	6	*acbfgijl*	10	*acgbefijl*	14	*acdbfgikl*
3	*acgbfh*	7	*acbgfijkl*	11	*abcedfgil*	15	*abcdgfikl*
4	*abcgfil*	8	*abcegfijl*	12	*acdbefgil*	16	*acbfdgil*

Tabelle 4: *Log-Traces für den Prozess OrderManagement*

Bei komplexen Systemen mit dutzenden Tasks und vielschichtigen Verhaltensweisen ist die Analyse häufig sehr aufwändig und kostenintensiv. Eine Möglichkeit, das Verständnis über den Prozess zu verbessern ist, das Auftreten verschiedener Varianten des Prozesses zu beobachten. Dann kann man jede Variante unabhängig von den anderen in ein Modell einfließen lassen. Natürlich sollten bei dieser Methode nur die wirklich relevanten Unterschiede verwendet werden, sonst wird das Modell zu speziell. Nun kann man aus den erhaltenen Varianten schrittweise ein Modell des gesamten Prozesses aufbauen, indem man sie miteinander kombiniert.

Während der Begriff der Zuverlässigkeit und seine Bedeutung für das Workflow Mining schon relativ bekannt ist, gibt es bisher kaum Ansätze, die die Erkennung von Varianten des Prozesses unterstützen. In (Greco et al, 2006) werden diese beiden Probleme angegangen und ein neuer Algorithmus vorgestellt, der mit sehr großen und komplexen Modellen umgehen kann. Es wird dazu eine klare und modulare Repräsentation des Prozesses erzeugt, wobei die irrelevanten Varianten mit Hilfe einer Gruppierungs-Methode für die Traces ausgesondert werden. Jede Variante wird durch ein eigenes Workflow-Schema *WS* (siehe Abschnitt 2.2.2) repräsentiert. Aus diesen einzelnen Schemata wird dann das disjunktive Workflow Schema $WS^{\vee}$ für den gesamten Prozess erzeugt. Ein disjunktives Schema ist zwar ausdrucksstärker, aber die Erzeugung eines solchen Schemas bringt auch Schwierigkeiten mit sich. So stellte es sich als praktisch unmöglich heraus, ein konformes disjunktives Schema zu berechnen und gleichzeitig zu prüfen ob dieses Schema zuverlässig genug ist. Deswegen wurde in dem Paper von Greco et al. eine Greedy-Strategie gewählt. Diese basiert auf der hierarchischen, iterativen Verfeinerung des disjunktiven Schemas. In jedem Schritt wird ein Schema $WS^{j} \in WS^{\vee}$ ausgewählt und die Traces, die von diesem Schema unterstützt werden, werden in Gruppen (Cluster) aufgeteilt. Der Algorithmus ist dabei so raffiniert, dass die Zuverlässigkeit bei jedem Schritt nur zunehmen kann.

Um die Traces effizient in Gruppen aufzuteilen, wird eine „flache“ relationale Repräsentation verwendet. Die Traces werden auf eine Menge aussagekräftiger Merkmale (*Features*) abgebildet. Dies hat den Sinn, dass diese Merkmale diejenigen Traces identifizieren, die nicht richtig von dem aktuell verfeinerten Modell repräsentiert werden. Als Beispiel wird in diesem Abschnitt der Prozess *OrderManagement* eingesetzt. Die Beschreibung der Zustände und ein anfängliches Schema findet sich in Abbildung 6. In Tabelle 4 finden sich Log-Traces dazu, anhand derer die Funktionsweise des Algorithmus erklärt werden soll.

7.1 Grundlagen für das Clustering-Verfahren

In dem Artikel wird ein eigener Mining-Algorithmus entwickelt. Die Autoren proklamieren aber, dass das Verfahren modular ist und sich im Prinzip mit jedem Algorithmus nutzen lässt. Das gilt auch für das verwendete Modell, aber natürlich sind in beiden Fällen Anpassungen nötig.

Zuerst wird aus dem Log ein anfängliches Modell erstellt, das dann immer weiter zu einer Anzahl von charakteristische Modellen verfeinert wird, die jeweils eine Klasse von Traces mit einer bestimmten Verhaltensweise modellieren. Im Folgenden wird die Zuverlässigkeit eines disjunktiven Workflow-Schemas $WS^{\vee}$ für ein Log L_P des Prozesses P definiert.

$$Zuverlässigkeit(WS^{\vee}, L_P) = \frac{\{s \mid s \in L_P \wedge s \models WS^{\vee}\}}{\{s \mid s \models WS^{\vee}\}} \tag{30}$$

$s \models WS^{\vee}$ bedeutet, dass s zu dem Schema konform ist (siehe Grundlagen 2.2.2). Formel (30) stellt also den Anteil der Traces im Schema dar, die auch im Log enthalten sind. Ähnlich wird die Vollständigkeit als der Anteil der Traces im Log definiert, die zu $WS^{\vee}$ konform sind:

$$Vollständigkeit(WS^{\vee}, L_P) = \frac{\{s \mid s \in L_P \wedge s \models WS^{\vee}\}}{\{s \mid s \in L_P\}} \tag{31}$$

Je größer der Anteil, desto zuverlässiger bzw. vollständiger ist das Modell. Das Log in Abbildung 6 hat beispielsweise eine Zuverlässigkeit von $\frac{16}{276}$ = 5,797% und eine Vollständigkeit von $\frac{16}{16}$ =100%. Die Zuverlässigkeit muss also dringend verbessert werden, während die Vollständigkeit bereits optimal ist. Das erreicht man, indem man mehrere, spezifischere Modelle einsetzt. Die beiden Modelle in Abbildung 15 bilden zusammen ein disjunktives WF-Schema, welches eine Zuverlässigkeit von 11,34%, aber eine Vollständigkeit von nur 68,75% hat, weil es bestimmte Traces im Log nicht akzeptiert (nämlich Nr. 8-12). Der vorgestellte Algorithmus macht eine Abnahme der Zuverlässigkeit unmöglich. Es werden immer mehr spezifische Modelle so kombiniert, dass die Zuverlässigkeit zunimmt und die Vollständigkeit möglichst hoch ist.

Im Folgenden werden die Begriffe α-zuverlässig und β-vollständig für α, $\beta \in [0, ...1]$ verwendet, wenn die Zuverlässigkeit größer gleich α bzw. die Vollständigkeit größer gleich β ist. Wünschenswert sind natürlich Werte möglichst nahe 1. Die Entscheidung, ob ein Schema 1-zuverlässig ist, ist *NP*-vollständig. Das *Maximum Process Discovery*-Problem MPD(L_P, m) erhält als Eingabe ein Log L_P des Prozesses P und eine natürliche

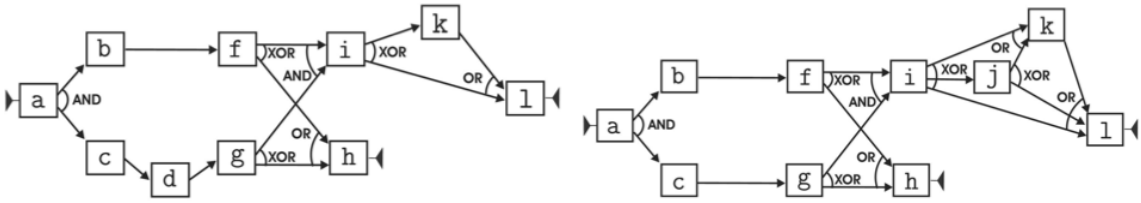


Abbildung 15: *Zwei Schemata, die zusammen das disjunktive Workflow Schema $WS^{\vee}$ ergeben*

Zahl *m*. Ziel ist es, ein 1-vollständiges disjunktives Workflow-Schema $WS^{\vee}$ zu finden, so dass $|WS^{\vee}| \leq m$ gilt und die Zuverlässigkeit das Maximum über alle 1-vollständigen Schemata ist. Da selbst dieses Problem noch recht aufwändig ist, wird eine Heuristik eingesetzt. Diese erzeugt ein Modell und verfeinert es schrittweise. Dabei wird aber immer eine Steigerung der Zuverlässigkeit erreicht. Dies erlaubt eine monotone Suche im Lösungsraum ohne jedes mal die Zuverlässigkeit überprüfen zu müssen.

7.2 Die Algorithmen MineWorkflow und ProcessDiscovery

In (Greco et al., 2006) wird nur eine Lösung für den Fall *m* = 1 vorgestellt. Gegeben sei ein Log L_P für den Prozess *P* über die Menge *A* von Tasks. Jeder Trace enthalte eine Anfangsaktivität a_0. Der *Abhängigkeitsgraph* enthält als Knoten die Tasks und für die Menge der Kanten gilt folgende Bedingung.

$$E = ((a,b) | \exists e \in L_P, i \in \{1,\ldots, Länge(s)\} \wedge a = s[i] \wedge b = s[i+1]) \tag{32}$$

Zwei Aktivitäten *a* und *b* laufen parallel ab, wenn sie in einem Zyklus auftreten. Sind sie nicht parallel und folgen aufeinander, so schreibt man $a \rightarrow b$ und sagt *a* ist der Vorgänger von *b*[5]. Im Folgenden werden schrittweise verschiedene Methoden vorgestellt. In der fertigen Implementierung werden diese alle aus der Methode *ProcessDiscovery* heraus aufgerufen. Hier eine Übersicht, damit die Orientierung leichter fällt.

- *ProcessDiscovery* (liefert ein fertiges disjunktives WF-Schema zurück)
  - *MineWorkflow* (liefert ein einzelnes Schema zurück)
  - *RefineWorkflow* (Verfeinert das Schema mit dem aufgerufen wurde)
    - *FindFeatures* (Sucht Kennzeichen und liefert eine Anzahl zurück)
    - *Project* (Bildet die Kennzeichen auf einen Vektor ab)

Der Algorithmus *MineWorkflow* konstruiert zuerst den Abhängigkeitsgraph und entfernt dann die Zyklen, da diese in Wirklichkeit parallelen Aktivitäten entsprechen. Um dabei nicht den Graph zu beschädigen, werden andere Kanten eingefügt um die Konnektivität zu erhalten. Wird eine Kante (*a*, *b*) entfernt, so werden Kanten eingefügt, die *a* und *b* mit einer vorhergehenden und nachfolgenden Aktivität verbinden. Dies wird für jeden Trace in L_P durchgeführt. So bleibt die Vollständigkeit erhalten. Anschließend werden noch die AND- OR- und XOR-Forks und -Joins festgelegt. Diese können leicht anhand der Vorgänger und Nachfolger ermittelt werden, wie das auch in anderen Algorithmen getan wird. Ihre Anzahl sollte aber so gering wie möglich bleiben, da sie die Zuverlässigkeit verschlechtern.

[5] Diese Notation ist die gleiche wie beim α-Algorithmus.

Um mit Rauschen umzugehen, kann die Konstruktion des Abhängigkeitsgraphen leicht verändert werden. Dazu wird ein Schwellwert ρ verwendet, sodass eine Kante nur dann in dem Graphen auftritt, wenn sie mindestens $\rho \times |L_P|$ auftritt.

Der Algorithmus *MineWorkflow* löst das MPD(L_P, 1)-Problem iterativ und inkrementell in linearer Zeit. Das resultierende Schema ist 1-vollständig. Nun wird ein Greedy-Algorithmus vorgestellt, der das Schema $WS^{\vee}$ durch eine hierarchische Gruppierung (Clustering), schrittweise in k Gruppen von Traces aufteilt, die durch – möglicherweise unterschiedliche – Modelle repräsentiert werden. Ziel ist es dabei die Zuverlässigkeit laufend zu erhöhen, die maximale Vollständigkeit aber zu bewahren.

Der Algorithmus *ProcessDiscovery* ruft zuerst den Algorithmus *MineWorkflow* auf und erzeugt so das anfängliche Modell WS_0^1. Anfangs besteht das disjunktive Workflow Schema $WS^{\vee}$ nur aus WS_0^1. Dieses Schema beinhaltet alle Traces in L_P und ist der Startpunkt für die folgenden Berechnungen. Ein Schema WS_i^j bezeichnet die i-te Verfeinerung seit dem Beginn der Berechnung. Der Index j dient dazu, Schemata auf der gleichen Vereinigungsebene zu unterscheiden. Jedes Schema ist zudem mit einer Menge von Traces $L(W_i^j)$ ausgestattet, welche von diesem Schema repräsentiert werden. In jedem Schritt wählt *ProcessDiscovery* ein Schema WS_i^j aus $W^{\vee}$ aus. Dabei sollte das Schema mit der geringsten Zuverlässigkeit genommen werden. Da dies aber sehr aufwändig ist, nimmt man eine Approximation zu Hilfe. Es wird anhand der OR-Forks entschieden, wie zuverlässig ein Schema ist. Diese führen zu Nichtdeterminismus und Traces, die nicht im Log auftreten, und verringern damit die Zuverlässigkeit.

Bei der Verfeinerung in der Methode *RefineWorkflow* werden die Traces, die zum jeweiligen Schema WS_i^j gehören, in Cluster aufgeteilt. Das geschieht so, dass die Zuverlässigkeit immer zunimmt. Dies wird Prinzip folgendermaßen erreicht: Die Methode *FindFeatures* wird innerhalb von *RefineWorkflow* aufgerufen und sucht relevante Kennzeichen (siehe unten), von denen angenommen wird, dass sie die Traces im Cluster charakterisieren. Diese Kennzeichen werden dann von der Funktion *Project* auf einen Vektor über dem Raum aller solcher Kennzeichen abgebildet. Die Komponenten des Vektors entsprechen dann den Kennzeichen. Das dient dazu, dass dann ein bereits bekannter effizienter Algorithmus zur Gruppenbildung (unabhängig vom Workflow Mining) eingesetzt werden kann, der als Eingabe einen Vektor erfordert. Um die Traces zu klassifizieren, werden häufige Teilsequenzen der Traces untersucht.

Der Algorithmus *ProcessDiscovery* hat eine Reihe von Parametern, die hier kurz vorgestellt werden:

- MPD(L_P, 1),
- k, die Anzahl neuer Schemata, die bei jeder Verfeinerung hinzugefügt werden,
- σ und γ sind zwei Schwellwert-Parameter für die Berechnung der *Features,*
- l ist die maximale Länge der extrahierten Features,
- *maxF* ist die obere Schranke für die Anzahl an Features, die von der Methode *FindFeatures* zurückgegeben werden sollen.

Grundlegend ist die Wahl des Parameters k. Seine Werte können von 2 bis unendlich reichen. Bei einem Wert von $k = 2$ benötigt der Algorithmus mehrere Schritte für die Berechnung. Bei einem unbegrenzten Wert wird das Ergebnis sofort zurückgegeben. Die schrittweisen Verfeinerungen haben aber viele Vorteile, sodass in der Praxis ein Wert $2 \leqslant k \leqslant 4$ eingesetzt wird. Die Parameter σ, γ, l und *maxF* werden zusammen mit i und

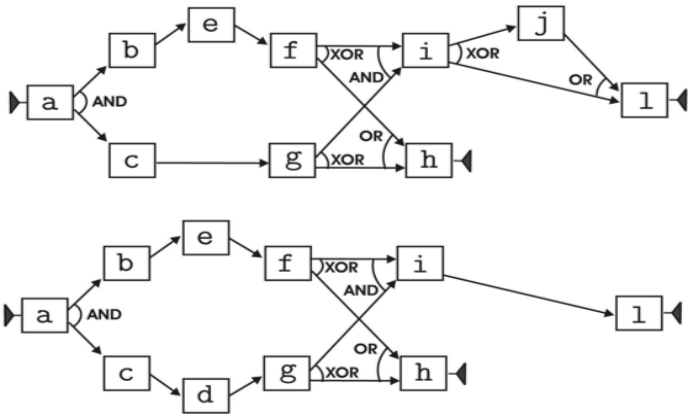


Abbildung 16: *Ergebnisse für ProcessDiscovery angewendet auf den Prozess OrderManagement*

j zuerst an die Funktion *RefineWorkflow* und später zusammen mit dem Schema und den dazugehörigen Traces an die Funktion *FindFeatures* weitergegeben. Ein Schema mit einer geringen Zuverlässigkeit vermischt unterschiedliche Ausführungsszenarien miteinander. Um die Zuverlässigkeit zu erhöhen müssen die (häufigen) Verhaltensweisen ausgesondert werden, die nicht richtig modelliert werden. Dazu nun ein Beispiel. Es kann bestimmte Einschränkungen im Ablauf des Einkaufs geben. So wird zum Beispiel neuen Kunden kein Treuebonus gewährt (Task *k* in Abbildung 6). Zwar treten die (Teil-)Sequenzen *abefi* und *ik* häufig im Log auf, aber ihre Kombination, *abefik,* wird zwar vom Modell angeboten, tritt aber nicht im Log auf. Das ist ein Grund für die Unzuverlässigkeit des Modells in Abbildung 6. Genau solche Kennzeichen sollen also ausgesondert werden.

7.3 Die Funktionen FindFeatures und Project

Die Funktion *FindFeatures* berechnet die Menge der minimalen Kennzeichen. Dazu wird in allen Schemata mit dem gleichen Vereinfachungs-Grad im Raum der möglichen Kennzeichen gesucht. Die beiden Schwellwerte σ und γ stellen Grenzen für die Häufigkeit der Kennzeichen dar. Ein *Kennzeichen* mit Schwellwerten (σ, γ) in einem Log L_P ist eine Sequenz ϕ der Form $a_1...a_h a$ für den die folgenden Bedingungen gelten. $a_1...a_h$ und $a_h a$ sind σ-häufig in L_P und $a_1...a_h a$ ist nicht γ-häufig in L_P. Je niedriger γ ist, desto unerwarteter ist dieses Kennzeichen. Für $\gamma = 0$ würde der Algorithmus demnach zwei Szenarien mischen, die völlig unabhängig voneinander sind. Eine Sequenz $a_1 a_2 ... a_h$ nennt man σ-häufig, wenn für jedes Paar $(a_i, a_{i+1}), 1 \leqslant i \leqslant h-1$ aufeinander folgender Aktivitäten einen Pfad von a_i nach a_{i+1} im Abhängigkeitsgraphen gibt und folgende Bedingung erfüllt ist:

$$\frac{\left|\{s \in L_P \mid a_1 = s[i_1], a_h = s[i_h] \wedge i_1 < ... < i_h\}\right|}{\left|L_P\right|} > \sigma \tag{33}$$

Die Methode *FindFeatures* benötigt maximal l – 2 Wiederholungen. Am Ende existiert eine Menge der minimalen Kennzeichen der Länge l (obere Schranke). Nachdem der Algorithmus die Features berechnet hat werden die *maxF* unerwartesten davon zurückgegeben. Die Methode *Project* bildet jedes gefundene Merkmal auf einen Punkt in

einem passenden Vektorraum ab. Dieser Vektorraum hat so viele Dimensionen, wie Kennzeichen betrachtet werden. Jeder Trace *s* wird wie folgt auf den Punkt $\vec{s}$ abgebildet. Sei $\phi = [a_1, \ldots, a_h]$ ein Merkmal, das von *FindFeatures* berechnet wurde. Dann ist der Wert der Komponenten von $\vec{s}$, die zu ϕ gehört gleich 0, wenn $a \in tasks(s)$ oder

$$\frac{\sum_i \left|\{s[i]\} \cap \{a_i, \ldots, a_h\}\right| \times length(s)^{length(s)-i}}{\sum_i length(s)^{length(s)-i}} \tag{34}$$

sonst. *Project* versucht also, die Traces abzubilden, indem die Funktion sie passend zum Vorkommen der Merkmale aufteilt. Der Wert wird bestimmt, indem das Vorkommen solcher Knoten lexikographisch gewichtet wird.

Nun kann die Methode *RefineWorkflow* das von *ProcessDiscovery* übergebene Schema anhand der zurückgegebenen Kennzeichen mit einem bekannten Algorithmus zur Gruppenbildung verfeinern . Das wiederholt sich so lange, bis sich die Anzahl der Schemata $|WS^{\vee}|$ nicht mehr ändert. Dann wird $WS^{\vee}$ zurückgegeben und der Algorithmus ist fertig.

7.4 Auswertung

Der von den Autoren vorgestellte Algorithmus wurde in dem Programm ProM implementiert und konnte somit umfangreichen Tests unterzogen werden. Die Ergebnisse sollen in diesem Abschnitt zusammengefasst werden. Zuerst wurde der Algorithmus mit dem eingangs vorgestellten Prozess *OrderManagement* (Abbildung 6) getestet. Es wurde ein Log generiert, wobei aber auch darauf geachtet wurde, dass *k* und *e* sich gegenseitig ausschließen. Das gleiche gilt für *j* und *d*. So ergeben sich unterschiedliche Anwendungsfälle, die in dem Schema in Abbildung 6 zusammengefasst werden. Durch die Ausführung von *ProcessDiscovery* mit den Parametern *maxF* = 5, $k = 4$, $\sigma = 0{,}05$, $\gamma = 0{,}01$ und $l = 5$ werden die Schemata in den Abbildungen 15 und 16 erzeugt. Jedes dieser Schemata entspricht genau einem Anwendungsfall, der von Typ des Kunden (Neukunde oder bereits registrierter Kunde) und vom Lagerbestand abhängt. Das resultierende Modell ist 1-zuverlässig und 1-vollständig.

Weitere Tests mit von einem Generator zufällig erzeugten Logs ergaben, dass der Algorithmus mit der Größe von *k* linear skaliert. Für den Fall $k = 1$ erhält man eine Zuverlässigkeit von unter 50% – der Algorithmus entspricht dann jedem normalen Mining-Algorithmus ohne Clustering. Bei Werten von $k \geq 4$ erreichen die Modelle 90-100% Zuverlässigkeit. Auch die benötigte Zeit skalierte linear mit *k*. Für 10000 Traces benötigte das Testsystem[6] mit $k=2$ ca. 60s, mit $k=8$ dagegen 160s. Das gleiche gilt auch für die Zeit, die für das Mining benötigt wurde bei konstantem *k* und Logs mit 1000 bis 10000 Traces.

Der Ansatz von Greco et al. ist fraglos sehr interessant und nützlich. Ein Überblick über komplizierte Prozesse fällt wesentlich leichter. Das gilt besonders dann, wenn der Prozess durch bestimmte Bedingungen beschränkt ist. Diese Bedingungen können sich zum Beispiel auf eine externe Datenbank beziehen. Das kommt in der Realität häufig vor, wenn Lagerbestände oder ähnliches Einfluss auf die Entscheidungen haben. Bishe-

[6] Pentium IV, 1600 MHz, 256 MB RAM, Windows XP Professional

rige Systeme schenkten dem entweder keine Beachtung oder drückten die Bedingungen mit Hilfe von Logik aus. Außerdem lässt sich das Prinzip auch zum Aufspüren von Anomalien benutzen. Denn bei der Auswertung fallen solche dann auf, weil sie ein eigenes Schema erzeugen.

An der Implementation des Algorithmus muss aber dringend noch gearbeitet werden. Die Anzahl von sieben Parametern zusätzlich zu den Log-Dateien ist unzumutbar für jemanden, der nicht mit dem Algorithmus vertraut ist. Wie die Autoren aber in ihrer Arbeit gezeigt haben, ist es sehr wohl möglich die Parameter auf einige realistische Werte einzuschränken. Dies könnte dann mit geringem Aufwand auch vom Algorithmus selbst anhand der Log-Dateien festgelegt werden.

8 Gegenüberstellung der Ansätze

In den vorhergehenden Abschnitten wurde die Funktionsweise der Verfahren zusammengefasst und eine erste Bewertung der Ansätze vorgenommen. Nun sollen die vorgestellten Ansätze miteinander verglichen werden. Im Folgenden werden zuerst Kriterien für die Gegenüberstellung vorgestellt und anschließend der Reihe nach betrachtet. Die einzelnen Kriterien sind hervorgehoben.

8.1 Kriterien

Die Algorithmen können verschieden große Schnittmengen von Log-Dateien verarbeiten. Daher ist es zuerst einmal wichtig zu wissen, welches Verfahren welche Logs verarbeiten kann. Die Fähigkeit *Schleifen* und *nebenläufige Prozesse* zu verarbeiten ist grundlegend. Schleifen kommen in vielen Prozessen in Form von Bedingungen nach dem Schema „solange *A* nicht erfüllt ist führe *B* aus“ vor. Nebenläufigkeit dagegen ist immer dann gefragt, wenn zwei Tasks parallel abgearbeitet werden sollen. Es ist offensichtlich, dass diese beiden Merkmale bei vielen realen Prozessen erfüllt sind. Werden Logs mit Schleifen und Nebenläufigkeit nicht unterstützt so wird bereits ein großer Teil der möglichen Anwendungsgebiete von vorne herein ausgeschlossen.

Ein weiterer Punkt ist das Rauschen. Es tritt – insbesondere bei Funkübertragungen – häufig auf und kann einen unerwünschten Einfluss auf das Modell haben. Durch Rauschen kann das Modell komplexer werden als nötig, was die Weiterverarbeitung behindert. Das gleiche gilt auch für Anomalien, die durch kleine Abweichungen beim Prozessablauf entstehen können. Diese Abweichungen sind für den Betrachter meist nicht von Interesse und sollten daher ebenfalls nicht im Modell enthalten sein. Anomalien können aber auch durch Angriffe verursacht werden. Denn ein Angriff ist nichts anderes als das Abweichen von einem festgelegten Ablauf an einem sicherheitskritischen Punkt. Dann sollten sie erst recht erkannt werden, damit die Sicherheitslücke gefunden und geschlossen werden kann. Ein Ansatz, der nicht mit *Anomalien und Rauschen* umgehen kann lässt sich nur in Umgebungen einsetzen, die nicht sicherheitskritisch sind und in denen kein Rauschen zu erwarten ist. Die kleinen Unterschiede zwischen einzelnen Ausführungen des gleichen Ablaufs machen das Modell aber dennoch komplizierter. Mit diesem Problem können Algorithmen umgehen, die Varianten erkennen können.

Die zuverlässige Erkennung *impliziter Abhängigkeiten* ist sehr aufwändig und wurde daher von nur einem hier vorgestellten Aufsatz näher verfolgt. Implizite Abhängigkeiten haben einen Einfluss auf das Verhalten des Prozesses und sollten daher in jedem Fall erkannt werden. Erkennt ein Algorithmus keine impliziten Abhängigkeiten so kann das erzeugte Modell Traces erzeugen, die ursprünglich nicht vorgesehen waren (siehe Abschnitt 5.2). Aufgrund der Komplexität des Problems haben fast alle Ansätze Logs mit solchen Abhängigkeiten ausgeschlossen. Dies ist natürlich in der praktischen Anwendung ein Hindernis, zumal man implizite Abhängigkeiten oft nicht auf einen Blick erkennen kann.

Die *Sicherheit* spielt im Zusammenhang mit Geschäftsprozessen immer eine große Rolle. Schließlich sollen Betrugsversuche, Diebstähle, Sabotage und andere Angriffe möglichst früh erkannt werden und keinesfalls in ein kritisches Stadium gelangen. Durch die in Abschnitt 6 eingeführten Methoden kann man zwar Anomalien und Rauschen vor der Modellerstellung und Abweichungen vom vorhandenen Modell während der Ausführung erkennen, aber es verbleiben weitere Probleme. So kann man äußere Be-

dingungen (*Policies*) nur schwer erfassen, denn sie sind nicht in den Logs enthalten. Policies legen zum Beispiel fest, wer mit bestimmten Daten arbeiten darf. Es muss also bei jeder Ausführung manuell geprüft werden, ob die ausführende Person überhaupt die entsprechenden Rechte hat. Solche Fragen können mithilfe eines Analyseverfahrens gelöst werden, das die Bewegung der Daten (engl. *Dataflows*) visualisieren kann. *Analysefunktionen* können aus Modellen zum Beispiel die Information gewinnen, welcher Benutzer wann mit welchen Daten gearbeitet hat oder woher eine bestimmte Information stammt. Unter diesem Punkt soll untersucht werden, welche Ansätze Funktionen mitbringen, die im Kontext der Sicherheit eingesetzt werden können

Die Usability, d.h. die *Nutzerfreundlichkeit* ist sehr wichtig, wenn der Algorithmus in der Praxis angewendet werden soll. Leider hat sich gezeigt, dass sich viele Arbeiten auf die theoretische Entwicklung der Verfahren konzentrieren und der Implementation und Erprobung wenig Aufmerksamkeit schenken. Erfordert ein Algorithmus viele Parameter, so kann er nicht von Laien genutzt werden, da gewisse Kenntnisse vonnöten sind.

Die Fähigkeit, das gegebene Modell zu *vereinfachen*, d.h. sich auf einen Teil des Modells oder eine Variante des Prozesses zu konzentrieren ist nur bei zwei der vorgestellten Ansätze gegeben. Beide Ansätze sind aber sehr viel versprechend, denn so bekommt man auch komplexe Modelle gut in den Griff. Die beiden Ansätze lösen das Problem zwar vollkommen unterschiedlich, aber der gemeinsame Nenner ist die Vereinfachung des Modells auf diejenigen Zustände, die für die aktuelle Fragestellung interessant sind.

Die *Qualität der Tests* in den hier vorgestellten Aufsätzen wurde mit Schulnoten von 1-6 bewertet. Dabei wurde der Umfang der Tests, die Realitätsnähe der verwendeten Logs und das Vorhandensein einer Implementation miteinbezogen. Manche Autoren testen ihre Algorithmen ohne wirkliche Implementation, d.h. der Algorithmus wird nur auf dem Papier erprobt. Das zeigt nicht viel mehr als die reine Funktionsfähigkeit des Verfahrens. Tests mit einer realistischen Anzahl an Tasks und Traces verbieten sich dann natürlich. Es ist also wünschenswert, dass die Verfahren implementiert und mit einer großen Anzahl von zufällig erzeugten Traces getestet werden. Dadurch sind dann auch Aussagen über die Laufzeit möglich.

Man muss beim Kriterium der Nutzerfreundlichkeit also unterscheiden zwischen dem Einsatz der Algorithmen durch möglicherweise unerfahrene Anwender und der Anwendbarkeit des Modells.

8.2 Gegenüberstellung

In Tabelle 5 finden sich die verwendeten Meta-Modelle und zugrunde liegenden Mining-Verfahren. Die Verwendung verschiedener Meta-Modelle wie WF-Netze und Workflow-Schemata ist unnötig. Die WF-Netze haben sich aber praktisch durchgesetzt. Das gleiche gilt für den α-Algorithmus zu Vergleichszwecken. Anschließend werden die oben vorgestellten Kriterien aufgeführt.

Alle Algorithmen können mit *Schleifen* umgehen. Bei dem Algorithmus von Bezerra und Wainer zur Erkennung von Anomalien findet sich zwar keine Angabe dazu, aber da das verwendete Modell lediglich ein gerichteter Graph ist, sollten Schleifen auch hier kein Problem darstellen. Der vorgestellte Algorithmus übernimmt ohnehin nur die Erkennung von Rauschen und Anomalien und nicht das eigentliche Mining. Für die *Nebenläufigkeit* gilt das leider nicht. Gerichtete Graphen kennen keine AND-Blöcke und können nicht zwei Übergänge parallel verfolgen (Bezerra und Wainer, 2008a). Alle anderen untersuchten Ansätze können aber mit nebenläufigen Prozessen umgehen. Die Erkennung von *Rauschen und Anomalien* wird von dem hier vorgestellten α-Algorithmus

nicht unterstützt. Das Log muss daher fehlerfrei sein, sonst wird ein komplizierteres Modell erzeugt als nötig. Allerdings zählt der α-Algorithmus zu den am besten untersuchten und weiterentwickelten Verfahren. Der Einsatz eines Schwellwertes für die minimale Häufigkeit eines Traces sollte zumindest grundlegende Fähigkeiten zur Erkennung von Rauschen erlauben. Aus dem gleichen Grund kann auch der α++-Algorithmus von Wen et al. Prinzipiell mit Rauschen umgehen. In dem hier vorgestellten Paper wird diese Fähigkeit aber nicht untersucht.

Die Erkennung *impliziter Abhängigkeiten* wird nur von zwei Verfahren unterstützt. In dem Aufsatz zum statistischen Workflow-Mining wird zwar nicht direkt auf implizite Abhängigkeiten eingegangen, aber da in der Tabelle auch indirekte Vorgänger bzw. Nachfolger erfasst werden, kann der Algorithmus diese zumindest ansatzweise erkennen. Die Stärke der Abhängigkeit wird durch einen Parameter geregelt. Dadurch kann man gezielt Abhängigkeiten ab einer bestimmten Entfernung voneinander ignorieren. Das Paper von Wen et al. widmet sich ausschließlich der Erkennung impliziter Abhängigkeiten mithilfe einer Erweiterung des α-Algorithmus. Die impliziten Abhängigkeiten werden hier nicht einfach nur anhand einer Wahrscheinlichkeit klassifiziert, wie im statistischen Workflow-Mining, sondern es wird speziell danach gesucht. Dieses Verfahren ist allerdings sehr von der Berechnung her sehr aufwändig, was sich bei sehr großen Prozessen sicher auch auf Laufzeit und Speicherbedarf auswirkt (Wen et al., 2006).

8.3 Sicherheit

Die Sicherheit spielt eine große Rolle bei jeder Art von Ablauf. Im folgenden soll untersucht werden, welche Hilfestellungen die vorgestellten Verfahren bei der Erkennung von Sicherheitsproblemen im Prozessablauf leisten können.

Das abduktive Workflow-Mining ist zwar nicht auf die Erkennung und Analyse von Sicherheitsproblemen spezialisiert, vermag aber dennoch eine große Hilfe zu leisten, denn es kann die Ausgabe auf diejenigen Transitionen beschränken, die mit einer gewählten Transition zusammenhängen. So kann man zum Beispiel leicht feststellen, wie ein bestimmter Mitarbeiter an sicherheitskritische Daten gelangt sein kann (Buffett und Hamilton, 2008).

Die Erkennung von Varianten kann man ebenfalls in einem sicherheitskritischen Kontext betrachten. Varianten werden oft vor der Modellerstellung als Anomalien herausgefiltert, wenn sie zu selten auftreten. Dies kann unerwünscht sein, da gewißermaßen auch Angriffe Varianten des Prozessablaufs darstellen. Diese können mit dem Clustering-Verfahren von Greco et al. dann genauer analysiert werden (Greco et al, 2006) .

Anwendung findet das Workflow-Mining bereits heute im Bereich der Data Loss Prevention (DLP). Dieses beschäftigt sich damit, wie man in einem Unternehmen verhindern kann, dass Mitarbeiter Daten auf USB-Sticks oder CDs mitnehmen bzw. ins Internet stellen. Da die administrative Abschaltung solcher Möglichkeit unproduktiv ist und strenge Restriktionen auch die Kreativität der Mitarbeiter für deren Umgehung fördern, sind subtilere Maßnahmen gefragt. Die Verletzungen der Policies werden mithilfe einer Host- oder Netzwerkbasierten Software in einer Log-Datei aufgezeichnet und können dann zu einzelnen Mitarbeitern und Rechnern zurückverfolgt werden. So erhält der Sicherheitsverantwortliche erstmal eine Übersicht, welche Daten in Umlauf sind und wer normalerweise mit welchen Daten arbeitet. Dies ist nötig, damit nicht wieder unproduktive Einschränkungen getroffen werden. Später können dann Warnungen angezeigt werden, wenn ein Mitarbeiter gegen die Policies verstößt. Im letzten Schritt werden bestimmte Zugriffe verhindert. Natürlich ist auch dieses System nicht vollkommen und

	α-Algorithmus	statistisches WF-Mining	Abduktion	α++-Algorithmus	Erkennung von Anomalien	Konformitätsprüfung	Gruppenbildung
Verwendetes Modell	WF-Netz	WF-Netz	WF-Netz	WF-Netz	Graph	WF-Netz	Schema
Verwendeter Algorithmus			α		inkrem.	α	
Fähigkeiten							
Schleifen	Ja	Ja		Ja	k.A.		Nein
Nebenläufigkeit	Ja	Ja		Ja	k.A.		Ja
Anomalien/Rauschen	Nein	Ja		k.A.	Ja		Ja
Varianten des Prozessablaufs	Nein	Ja		k.A.	Nein		Ja
impliziten Abhängigkeiten	Nein	Ja		Ja	Nein		Nein
Sicherheit							
Visualisierung von Dataflows	Nein	Nein	Ja	Nein	Nein	Nein	Nein
Darstellung ähnlicher Abläufe	Nein	Nein	Nein	Nein	Nein	Nein	Ja
Nutzerfreundlichkeit							
Anzahl der Parameter	0	2	0	0	0/1	0	7
Vereinfachung des Modells möglich?	Nein	Nein	Ja	Nein	Nein		Ja
Wie gut wurde getestet?	1	3	4	5	2	5	1
Tool-Unterstützung							
ProM	Ja	k.A.	k.A.	Ja	k.A.	k.A.	Ja

Tabelle 5: *Vergleich der Ansätze*

lässt sich beispielsweise durch Verwendung einer Linux-Live-CD austricksen, aber da der einzelne Mitarbeiter nie genau weiß, wie die Policies aussehen, kann er auch nie sicher sein, dass seine Aktivität nicht geloggt wird. Das System wirkt also auch dann abschreckend, wenn gar kein Verstoß gemeldet wurde. Versuche zeigen, dass die Anzahl der Verstöße von der Einführung der Überwachung über die Ankündigung bei den Mitarbeitern und schließlich das Anzeigen von Warnungen immer weiter zurückgehen (Ziegler, c't 3/10). Allerdings ist dieses Verfahren aus Sicht des Datenschutzes durchaus kritisch zu sehen. Ermöglicht es doch eine praktisch nahtlose Überwachung der Arbeit jedes einzelnen Angestellten.

8.4 Nutzerfreundlichkeit

Zu den Kriterien der Nutzerfreundlichkeit zählt zuerst die Anzahl der Parameter, die beim Aufruf des Algorithmus übergeben werden. Erfreulicherweise benötigen die meisten Algorithmen keine Parameter. Das statistische Workflow-Mining benötigt zwei Parameter, die geschätzte Stärke des Rauschens und einen Parameter, der die Entfernungsabnahme der Stärke von indirekten Abhängigkeiten regelt. Der zweite Parameter muss nur in speziellen Fällen angepasst werden (Weijters und Van der Aalst, 2001).

Das Verfahren von Bezerra und Wainer zur Anomalieerkennung vor der Erstellung eines Modells bietet drei unterschiedliche Methoden an, alle haben einen Wert, der vom Benutzer angepasst werden könnte, allerdings wurden die Werte von den Autoren bereits sinnvoll vorbelegt (Bezerra und Wainer, 2008a).

Einzig der Ansatz zur Gruppenbildung verlangt dem Benutzer eine sehr große Zahl von sieben Parametern ab. Man braucht schon ein sehr tiefes Verständnis des Verfahrens, um die Auswirkungen der einzelnen Parameter zu verstehen. Zwar schreiben die Autoren in ihrem Paper von den Werten, die sie im Test verwendet haben und werten die Wirkung einzelner Parameter auch sehr genau aus, aber bei so vielen Parametern verliert man dennoch leicht den Überblick und die Einstiegshürde wird damit sehr hoch angesetzt (Greco et al, 2006).

Das zweite Kriterium im Bereich der Nutzerfreundlichkeit ist die Qualität und der Umfang der Tests. Der α-Algorithmus nimmt hier jedoch eine Sonderrolle ein, da er von vielen Autoren als Grundlage oder Vergleich genutzt wird. Dadurch hat der Algorithmus genug praktische Anwendung und seine Zuverlässigkeit ist gesichert. Davon Profitiert auch der α++-Algorithmus.

Das statistische Workflow-Mining (Weijters und Van der Aalst, 2001) wurde nur mit Logs getestet, die maximal 16 unterschiedliche Tasks enthalten. Diese Anzahl ist nicht sehr realistisch für den Einsatz in einem Unternehmen. Die unterschiedlichen Typen von Rauschen sind unsinnig, denn Rauschen ist meist zufällig. Außerdem wurde darauf verzichtet, zufällig verrauschte Logs zu testen, d.h. ohne eine feste Angabe des Anteils verrauschter Traces. Das wäre insbesondere deswegen interessant, weil es ja einen Parameter gibt, der den Anteil der verrauschten Logs widerspiegelt. Im Test war der Anteil verrauschter Traces jedoch immer genau so groß wie der Parameter.

Beim abduktiven Workflow-Mining bestehen vor allem Zweifel an der Verwendung eines eigenen, nicht näher bezeichneten Mining-Algorithmus, welches offenbar wesentlich umfangreichere Modelle erzeugt, als der ebenfalls herangezogene α-Algorithmus. Das lässt die Ergebnisse übermäßig gut erscheinen. Die verwendeten Log-Dateien werden bei dem Process-Mining-Tool ProM (siehe nächster Abschnitt) mitgeliefert, was eine sehr gute Idee ist, da sie die Vergleichbarkeit der Algorithmen untereinander gewährleistet. Leider fehlt auch eine Angabe, welcher Task denn als Ziel gewählt wurde. Ist dies ein Task nahe dem Startknoten des Modells, so ist die Berechnung natürlich wesentlich einfacher als wenn der Knoten tief im Modell eingebunden ist oder gar ein Teilmodell als Ziel ausgewählt wird (Buffett und Hamilton, 2008).

Das Verfahren von Wen et al. wurde in dem Paper selbst nur rein theoretisch getestet und in den zum Test verwendeten Modellen hat sich ein Widerspruch eingeschlichen. Allerdings muss man anerkennen, dass der Algorithmus beim Tool ProM mitgeliefert wird und daher einem großen Kreis von Wissenschaftlern für Tests zur Verfügung steht. Gerade bei diesem Algorithmus wäre aber eine Untersuchung der Laufzeit im Vergleich zum α-Algorithmus sehr interessant gewesen (Wen et al., 2006).

Während der Ansatz von Bezerra und Wainer auf zweieinhalb Seiten die Testergebnisse für die drei Algorithmen präsentiert und eine tabellarische Auflistung der Ergebnisse mitliefert, gibt es in dem Verfahren von Van der Aalst und de Medeiros leider überhaupt keinen Abschnitt über die experimentellen Ergebnisse. Allerdings hängen die Ergebnisse in diesem Fall auch sehr stark von der Güte des zugrunde liegenden Modells ab und die Autoren behaupten, dass jede Abweichung von diesem Modell festgestellt werden kann. Trotzdem wäre eine praktische Erprobung wünschenswert – schon allein um möglichen Problemen auf die Spur zu kommen. Dies wäre auch im Hinblick auf die Sicherheit interessant (Bezerra und Wainer, 2008a), (Van der Aalst und de Medeiros, 2004).

Das disjunktive Workflow-Mining gehört zu den sehr gut getesteten Verfahren. Auf nahezu drei Seiten wird mit zahlreichen Diagrammen ausgewertet, welchen Einfluss die wichtigsten Parameter haben. Getestet wurde einerseits mit dem vorgestellten Prozess *OrderManagement*, der aus 12 Tasks besteht. Es wurden 5000 zufällige Traces generiert. Zusätzlich wurde noch ein weiterer Prozess vorgestellt (10 Tasks), von dem ebenfalls 5000 Traces generiert wurden. Dann wurde die Wirkung der Parameter an einem Prozess mit 40 Tasks dokumentiert. Zusätzlich wurden auch noch Log-Dateien von der ProM-Webseite genutzt. Hier wurden realistische Logs zum Test verwendet (Greco et al, 2006).

8.5 Software-Unterstützung

Es gibt unterschiedliche Tools die das Workflow Mining implementieren. Das bekannteste und flexibelste ist ProM [1]. ProM ist OpenSource und basiert auf Java. Es bietet ein Plugin-System, über das Mining-Algorithmen eingebunden werden können. Darüber hinaus werden auch Schnittstellen für Analyse, Monitoring und Umwandlung angeboten. Der Unterschied zwischen Mining und Analyse ist, dass bei der Analyse bestimmte Informationen aus dem Modell gewonnen werden und beim Mining ein Modell. Das abduktive Workflow-Mining und auch das disjunktive Workflow Mining gehören in den Bereich Analyse. Hier finden sich außerdem Plugins, die ein Bewertung der Qualität des Modells ermöglichen. Durch die Umwandlungs-Plugins kann man sehr einfach von einer Modellvariante in die andere konvertieren. Intern nutzt ProM ein XML-Format zum Verarbeiten und Speichern der Logs und Modelle. Es können aber auch viele andere Formate importiert werden. Mitgeliefert werden eine große Anzahl fertiger Logs und Modelle, welche sich gut zu Test- und Benchmarkzwecken eignen.

Das ProM-Framework ist mit seiner graphischen Oberfläche relativ intuitiv zu bedienen und bietet viele Komfortfunktionen wie das Umbenennen von Tasks oder frei konfigurierbare Filter. Diese dienen beispielsweise dazu, aus dem Log unvollständige oder aus anderen Gründen unerwünschte Daten herauszufiltern. Zusätzlich im Log enthaltene Informationen wie Timestamps oder Benutzernamen lassen sich ebenfalls auswerten, zum Beispiel in der Form von UML-Sequenzdiagrammen.

ProM ist ein sehr mächtiges Tool. Durch die vielen Plugins ist es allerdings etwas überfrachtet und könnte Einsteiger überfordern, zumal es für manche Plugins keine Dokumentation gibt. ProM ist in erster Linie ein Framework für wissenschaftliche Anwender. Sie haben dadurch die Möglichkeit, ihre Algorithmen ohne viel Arbeit an einer graphischen Oberfläche zu testen. Geschäftliche Anwender werden sich kaum mit den vielen unterschiedlichen Plugins und ihren Details auseinandersetzen wollen. Das Tutorial [1] richtet sich aber trotzdem an Manager, und beschreibt detailliert, wie man welche Information aus dem Log gewinnt.

In ProM sind im Auslieferungszustand der α-, α++ sowie ca. 40 weitere Mining-Plugins installiert. Das Mining- und Analyse-Plugin für das disjunktive Workflow-Mining kann von [5] heruntergeladen werden. Das abduktive Workflow-Mining wurde zwar mit den Beispieldatensätzen aus ProM getestet, aber leider findet sich nirgends ein Plugin. Das ebenfalls in einigem Papers erwähnte Tool EmiT scheint in ProM aufgegangen zu sein.

8.5.1 Andere Tools

Die Software *Process Mining Workbench* ist wesentlich weniger umfangreich als ProM. Sie basiert ebenfalls auf Java, ist jedoch nicht OpenSource und benötigt MS Access für die Datenbank. Bisher ist nur ein einziger Mining-Algorithmus implementiert. Leider findet man keine Angaben darüber, welcher Algorithmus dahinter steckt. Die Logs werden hier ebenfalls in einem XML-Derivat verarbeitet, das Standard-Format von ProM kann jedoch nicht geöffnet werden. Modelle lassen sich auch simulieren. Eine Dokumentation ist praktisch nicht vorhanden [6].

Außerdem gibt es noch verschiedene Lösungen, die sich an geschäftliche Anwender richten. Ein Beispiel ist YAWL (Yet Another Workflow Language). Dabei handelt es sich um ein Business Process Management (BPM)-Tool. Es bietet also (noch) kein Mining-Verfahren, sondern bietet eine Sprache an um Modelle zu beschreiben. Dabei können auch komplexe Bedingungen realisiert werden.

9 Fazit und Ausblick

In dieser Arbeit wurde ein Überblick in das Themengebiet des Workflow Mining gegeben, und einige ausschlaggebende, wissenschaftliche Verfahren aus diesem Gebiet zusammengefasst. Natürlich konnten nicht alle interessanten Verfahren vorgestellt werden, sondern es wurde exemplarisch aus verschiedenen Aufsätzen gewählt. Bei der Einführung wurden alle wesentlichen Begriffe erklärt und die verwendeten Verfahren und Metamodelle dargestellt. Es ist interessant zu sehen, wie in der Vergangenheit aufgetretene Probleme nach und nach gelöst wurden. Der Abschnitt über statistisches Workflow-Mining beschreibt ein frühes Verfahren. Die Erfahrungen daraus sind in den α-Algorithmus eingeflossen, der heute die Grundlage vieler anderer Verfahren darstellt. Die Abschnitte über die Erkennung von Anomalien und impliziten Abhängigkeiten stellen Lösungen für anfängliche Probleme zur Verfügung. Diese können in zukünftigen Arbeiten angewendet und eingebunden werden. Zwei der vorgestellten Arbeiten, nämlich das abduktive und disjunktive Workflow Mining, widmen sich der Vereinfachung des entstandenen Modells. Auch diese Ansätze sind wichtig, denn die Prozesse werden zunehmend komplizierter, und dadurch die Modelle schwerer zu verstehen.

Es fehlen noch Ansätze, die mit allen hier vorgestellten Problemen wie impliziten Abhängigkeiten, Rauschen, Anomalien und Varianten umgehen können und die gleichzeitig mächtige Analysefunktionen zur Verfügung stellen. Überhaupt wird die Analyse eine immer wichtigere Rolle einnehmen, da sie die Interessen der Manager befriedigen kann und damit Grundlage für den Einsatz in einem Unternehmen ist. Die Entwicklung leistungsfähiger Analysewerkzeuge wird also ein Schwerpunkt der zukünftigen Entwicklung sein. Eine intuitive Umgebung, die nur die notwendigen Parameter und nur ein allumfassendes Mining-Verfahren enthält, ist ebenfalls wichtig, damit auch Personen ohne tiefgehendere Kenntnisse mit der Software arbeiten können.

Workflow Mining wird zunehmend zur Analyse von Sicherheitsproblemen eingesetzt und ist dafür sehr gut geeignet. Sicherheitslücken und Angriffsversuche können durch die genaue Analyse der Datenflüsse schnell entdeckt werden, und es entsteht ein genaues Bild, wer mit welchen Daten in Berührung gekommen ist. Allerdings wird hierdurch auch die Überwachungssituation verschärft. Man muss diesen Faktor durchaus kritisch betrachten, denn die Auswertung von Kundenprofilen („Was kauft dieser Kunde in welchem Geschäft?“) lässt sich mithilfe von Workflow Mining sehr elegant bewerkstelligen. Durch Analysewerkzeuge kann man dann eventuell sogar herausfinden, wer den Kunden empfohlen hat. Dieses Szenario ist auch nicht so weit hergeholt, wenn man bedenkt, dass z.B. Kreditkarten oder Kundenkarten praktisch wie eine Log-Datei arbeiten und sich bei jedem Einkauf mit dem Firmenserver synchronisieren.

Quellen

Literatur

W.M.P. van der Aalst, A.J.M.M. Weijters und L. Maruster, Workflow Mining: Discovering Process Models from Event Logs, *IEEE Transactions on Knowledge and Data Engineering*, S. 1128-1142, 2003

J. E. Cook und A. L. Wolf, Discovering Models of Software Processes from Event-Based Data, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, S. 215-249, 1996

G. Greco, A. Guzzo, L. Pontieri und D. Saccà, Discovering Expressive Process Models by Clustering Log Traces, *IEEE Transactions on Knowledge and Data Engineering*, S. 1010 - 1027, 2006

W.M.P. van der Aalst und A.K.A. de Medeiros, Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance, *Second International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004)*, S. 69-84, 2004

A.J.M.M. Weijters und W.M.P. van der Aalst, Process Mining: Discovering Workflow Models from Event-Based Data, *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, S. 283-290, 2001

S. Buffet und B. Hamilton, Abductive Workflow Mining, *The 4th Workshop on Business Process Intelligence*, 2008

L. Wen, J. Wang und J. Sun, Detecting Implicit Dependencies Between Tasks from Event- Logs, *Frontiers of WWW Research and Development - APWeb 2006, 8th Asia-Pacific Web Conference*, 2006

W.M.P. van der Aalst, V. Rubin, B.F. van Dongen, E. Kindler und C.W. Gunther, Process Mining. A Two-step Approach Using Transition Systems and Regions, *BPM Center Report*, 2006

Van Dongen et al., 2006: B.F. van Dongen, N. Busi, G.M. Pinna und W.M.P. van der Aalst, An Iterative Algorithm for Applying the Theory of Regions in Process Mining, 2006

A.K.A. de Medeiros, B.F. van Dongen, W.M.P. van der Aalst und A.J.M.M Weijters, Process Mining for Ubiquitous Mobile Systems: An Overview and a Concrete Algorithm. , *Ubiquitous Mobile Information and Collaboration Systems (UMICS 2004)*, S. 156-170, 2004

F. Bezerra und J. Wainer, Anomaly Detection Algorithms in Business Process Logs, *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems*, 2008a

F. Bezerra und J. Wainer, Anomaly Detection Algorithms in Logs of Process Aware Systems, *Proceedings of the 2008 ACM symposium on Applied computing*, S. 951-952, 2008b

M. Ziegler, An die Kette. Werkzeuge gegen Datenklau, *c't 3/2010*, S. 138-141

Weblinks

[1] Http://www.processmining.org, Stand 15.01.2010

[2] http://de.wikipedia.org/wiki/Enterprise_Resource_Planning, Stand 21.12.2009

[3] http://de.wikipedia.org/wiki/Finite_State_Machine, Stand 22.12.2009

[4] http://de.wikipedia.org/wiki/Heuristik#Informatik, Stand 25.12.2009

[5] http://staff.icar.cnr.it/wfmining/tools.htm, Stand 15.01.2010

[6] http://www.processmining.de, Stand 26.01.2010