Integration Adaptiver Prozess-Management-Technologie und Process Mining

Diplomarbeit an der Universität Ulm

Fakultät für Informatik

Abteilung Datenbanken und Informationssysteme

vorgelegt von

Marco Waimer

Gutachter:

Dr. Stefanie Rinderle, Universität Ulm

Prof. Dr. Peter Dadam, Universität Ulm

31. August 2006

Inhaltsverzeichnis

Kurzfassung

Viele Unternehmen haben in den letzten Jahren prozessgestützte Informationssysteme (*Process-aware Information Systems, PAIS*) eingeführt, um ihre Geschäftsprozesse zu unterstützen. Diese Systeme zeichnen typischerweise die Ereignisse auf, die sich auf die Ausführungen aktueller Geschäftsprozesse beziehen (z.B. in Ausführungs-Logs). Diese Daten können sowohl für eine Prozess-Performanz-Analyse als auch für Prozessoptimierungen verwendet werden.

Process Mining bietet in diesem Kontext viel versprechende Perspektiven. Die bisher existierenden *Mining* Techniken werden im Zusammenhang mit operativen Prozessen verwendet, d.h. Information wird aus Ausführungs-Logs extrahiert (Prozesserkennung) oder Ausführungs-Logs werden mit den zugrunde liegenden Prozessmodellen verglichen (Konformitätsprüfung). Allerdings machen Ausführungs-Logs nur einen Teil der Daten aus, die während der Prozessausführung gesammelt werden. Adaptive Prozess-Management-Systeme (PMS) bieten zusätzliche Informationen über Prozessänderungen (z.B. Ad-Hoc-Änderungen einer Prozessinstanz) in Änderungs-Logs. Aus diesen Log-Daten können Informationen für mögliche Prozessoptimierungen gewonnen werden und die adaptiven PMS bieten die Werkzeuge, um durch *Process Mining* angestoßene Prozessoptimierungen nahtlos einzubringen.

In dieser Arbeit werden verschiedene *Process Mining Algorithmen* evaluiert und basierend auf den Ergebnissen ein Rahmenwerk vorgestellt, das die Technologien adaptives Prozess-Management und *Process Mining* zusammenfasst, um die Vorteile beider Ansätze nutzen zu können. Dazu wird ein Datenmodell für Änderungs-Log-Daten präsentiert und ein neuer Algorithmus für das *Mining* von Änderungs-Logs eingeführt. Der Änderungsprozess, den dieser neue Algorithmus (*Change Mining Algorithmus*) liefert, bietet eine Gesamtübersicht über alle Änderungen die (bisher) stattgefunden haben. Diese kann wiederum als Basis für alle Arten von Prozessoptimierungen dienen, z.B. könnten eine Neugestaltung des Prozesses oder bessere Kontrollmechanismen veranlasst werden.

1 Einleitung

Viele der heutigen Informationssysteme basieren auf expliziten Prozessmodellen. *Workflow-Management-Systeme* (WfMS), aber auch *Enterprise Resource Planning* (ERP), *Customer Relationship Management* (CRM), *Supply Chain Management* (SCM) und *Business to Business* (B2B) Systeme werden auf der Basis von Prozessmodellen gestaltet, die den Ablauf der auszuführenden Schritte festlegen.
Um die Prozesse effektiv zu unterstützen, muss die Umsetzung durch ein Prozess-Management-System (PMS) so nah wie möglich an den tatsächlich ablaufenden Prozessen sein. Deshalb ist es nicht ausreichend, ein Prozessmodell nur einmal zu erstellen und dann lange Zeit mit dem definierten Modell zu arbeiten. Es muss möglich sein, das Modell an sich ändernde Rahmenbedingungen anzupassen. Wenn z.B. ein kleines Detail an einem Prozessmodell schlecht modelliert wurde oder externe Änderungen den Prozess beeinflussen, sind die Anwender gezwungen vom vorgegebenen Prozessmodell abzuweichen. Weil die Prozessoptimierung eine teure und zeitaufwändige Aufgabe ist, kann das dazu führen, dass die Benutzer „hinter dem Rücken des Systems“ arbeiten, also nicht mehr dem vorgegebenen Prozessmodell folgen. Am Ende ist das PMS im Extremfall eher eine Belastung als die Hilfe, als die es gedacht war. Deshalb wurden adaptive PMS entwickelt, um flexibel auf sich ändernde Bedingungen reagieren zu können. Dem Anwender wird die Möglichkeit gegeben, die Prozessdefinition weiterzuentwickeln, sie an geänderte Situationen anzupassen. Adaptivität kann durch dynamische Änderungen verschiedener Prozessaspekte (z.B. Kontrollfluss und Datenfluss) auf unterschiedlichen Ebenen (z.B. Instanz- und Schemaebene) unterstützt werden. So erlauben Ad-Hoc-Änderungen auf Instanzebene (Hinzufügen oder Löschen eines Schrittes) eine flexible Anpassung einzelner Prozessinstanzen an außergewöhnliche Situationen oder geänderte Rahmenbedingungen. Solche Abweichungen werden typischerweise in Änderungs-Logs festgehalten, die zusätzlich zu Ausführungs-Logs erstellt werden.

Ein anderes Problem ist die Analyse und Erstellung eines Prozessmodells auf der Basis eines existierenden Geschäftsmodells. Solch eine Erzeugung ist ein komplizierter, lang andauernder Prozess und typischerweise gibt es Unterschiede zwischen dem tatsächlich ablaufenden Prozess und dem Prozess, so wie er vom Management wahrgenommen wird. Um die Erstellung eines Prozessmodells zu unterstützen, bieten sich *Process Mining* Techniken an, die aus den Ausführungs-Log-Daten bestehender Systeme das Prozessmodell rekonstruieren können (durch Erkennung sich wiederholender Prozessfragmente). Ist ein Geschäftsprozess bereits modelliert, kann *Process Mining* für eine Delta-Analyse verwendet werden, um den modellierten Prozess mit dem tatsächlich ablaufenden Prozess zu vergleichen. So können anhand von Ausführungs-Log-Daten Fehler oder Engstellen in einem Prozessmodell aufgedeckt werden. Der praktische Nutzen von *Process Mining* hängt stark von der Qualität der verfügbaren Log-Daten ab und die Log-Daten realer Prozessabläufe sind selten fehlerfrei und/oder vollständig. Die fehlenden Log-Einträge schränken den Nutzen von *Process Mining* (Rekonstruierung eines Prozesses) und Delta-Analyse (Vergleich gewünschter/tatsächlicher Prozess) enorm ein.

1.1 Integration von adaptivem Prozess-Management und Process Mining

Heutzutage sind adaptive Prozess-Management-Systeme, wie der *ADEPT Demonstrator* (entwickelt von der Abteilung DBIS[1], Universität Ulm) verfügbar, die den Anwender bei nötigen Abweichungen durch Ad-Hoc-Änderungen unterstützen. Auch die Verbesserung des zugrunde liegenden Prozesses wird durch Prozesstyp-Änderungen ermöglicht. Diese Systeme bieten aber keinerlei Unterstützung, um zu einem Prozessmodell zu gelangen oder von vorhergehenden Änderungen zu lernen. Eine andere Forschungsrichtung beschäftigt sich mit dem *Mining* von Prozessen. Diese Technologie unterstützt die Erfassung von Prozessstrukturen aus Ausführungs-Log-Daten, um so die Modellierung (unbekannter) Prozesse zu erleichtern. Darüber hinaus können modellierte Prozesse mit den tatsächlich ablaufenden Prozessen anhand der so genannten Delta-Analyse verglichen werden um Rückschlüsse für eine Prozessoptimierung zu gewinnen. Hierbei konzentrieren sich heutige *Process Mining* Techniken (wie z.B. im Rahmenwerk *ProM* der TU Eindhoven realisiert) bisher nur auf die Analyse reiner Ausführungs-Logs. Insbesondere fehlt hier die nahtlose Einbringung der gewonnenen Erkenntnisse (seien dies abgeleitete Prozessmodelle oder Ergebnisse einer Delta-Analyse) in das zugrunde liegende Prozess-Management-System.

Beide Technologien sind also für sich betrachtet sehr hilfreich, ihr wahres Potential können sie aber erst durch eine Verschmelzung erreichen. *Process Mining* hilft bei der Modellierung von Prozessen und überwacht für die ablaufenden Prozesse, ob sie modellkonform ablaufen oder nicht. Durch eine zusätzliche Analyse der Änderungs-Logs, die adaptive PMS bereitstellen, könnten häufige Änderungen (auf Instanzebene) erkannt werden und daraus Vorschläge für Prozesstypoptimierungen abgeleitet werden (d.h. Lernen aus vorangegangenen Änderungen). Über eine nahtlose Integration können diese Prozesstyp-Änderungen dann in *ADEPT* vorgenommen werden und – wenn gewünscht – sogar auf laufende Instanzen propagiert werden. Das Ziel wird deshalb im Folgenden sein, eine volle Integration von adaptivem Prozess-Management und *Process Mining* zu erreichen, um den Anwender während des Lebenszyklus eines Prozesses umfassend zu unterstützen.

1.2 Ziele und Aufgabenstellung der Diplomarbeit

Ziel dieser Arbeit ist die Integration von adaptiver Prozess-Management-Technologie und *Process Mining*. Dazu sollen bereits existierende Arbeiten gesichtet werden, um ein Grundverständnis für die Technologien zu bekommen. Nach einer Einarbeitung in die Basiskonzepte und Algorithmen des *Process Mining* soll der *ADEPT Demonstrator* (als Vertreter eines adaptiven Prozess-Management-Systems) erweitert werden, um damit Ausführungs-Log-Daten für Vergleich und Evaluierung der (in *ProM* realisierten) *Process Mining* Techniken erzeugen zu können. Für die Algorithmen, die in Verbindung mit *ADEPT* nützlich sind, soll untersucht werden, wie sich verschiedene Kriterien (Anzahl Instanzen, Menge *Noise*, Komplexität Prozess) auf das Ergebnis des *Mining* Prozesses auswirken und wie konform die ursprünglichen und die rekonstruierten Modelle (aufgrund unterschiedlicher Metamodelle) sind.

[1] Datenbanken und Informationssysteme

Dazu müssen basierend auf unterschiedlichen Beispielprozessen Ausführungs-Log-Daten nach diesen Kriterien erstellt werden. Eine ausführliche Evaluation der Zusammenarbeit der existierenden Technologien (v.a. *Process Mining*) ist wichtig für das Verständnis, denn die Ergebnisse dienen als Basis für die weitere Integration der beiden Technologien. Weil aufgrund häufig auftretender Instanz-Änderungen Vorschläge für eine Prozessoptimierung abgeleitet werden sollen, müssen Änderungs-Log-Daten mit Hilfe von *Process Mining* analysiert werden. Dazu muss erst einmal festgelegt werden, welche Information ein Änderungs-Log enthalten muss. Steht das Format für Änderungs-Logs, müssen solche Änderungs-Log-Daten erzeugt werden, um damit die bestehenden *Mining* Verfahren zu testen. Danach soll der in Zusammenarbeit mit der TU Eindhoven entwickelte Algorithmus für das *Mining* von Änderungs-Logs vorgestellt werden und mit den bestehenden Algorithmen verglichen werden.

1.3 Aufbau der Arbeit

Für eine Integration von *ADEPT* und *ProM* müssen erst die bestehenden Verfahren verstanden werden und sie müssen genutzt werden können, deshalb werden in Kapitel 2 die Grundlagen für das *Mining* von Ausführungs-Logs vorgestellt, bevor in Kapitel 3 die bisherigen Möglichkeiten ausführlich evaluiert werden. Ein gemeinsames Rahmenwerk für die Integration wird in Kapitel 4 eingeführt, danach werden die Grundlagen für das *Mining* von Änderungs-Logs betrachtet. In Kapitel 6 werden zuerst bestehende *Mining* Verfahren auf Änderungs-Logs angewendet, bevor ein völlig neuer Algorithmus vorgestellt wird. Nach dem Aufzeigen verwandter Arbeiten werden die Ergebnisse dieser Arbeit in Kapitel 8 noch einmal zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

2 Grundlagen für das Mining von Ausführungs-Logs

Bevor in Kapitel 3 gezeigt wird, wie Ausführungs-Logs automatisch erstellt werden und verschiedene *Mining* Algorithmen mit diesen Log-Daten verglichen werden, werden in diesem Abschnitt erstmal die Grundlagen für das *Mining* von Ausführungs-Logs vorgestellt. Zuerst wird das *ADEPT*-Prozessmodell eingeführt, auf dessen Basis später die Log-Daten erstellt werden. Danach wird ein Prototyp (*ADEPT Demonstrator*) präsentiert, der die Anforderungen des *ADEPT*-Prozessmodells umsetzt und der im Rahmen dieser Arbeit erweitert wurde, um damit automatisch Log-Daten erzeugen zu können. Um die erstellten Log-Daten analysieren zu können sollen *Process Mining* Techniken verwendet werden, deren grundlegende Funktionsweise gezeigt wird, bevor ein Rahmenwerk für *Process Mining* (*ProM*) vorgestellt wird. Abschließend wird noch der im Folgenden verwendete Beispielprozess ‚*OP-Vorbereitung*' eingeführt.

2.1 ADEPT Workflow Modell

ADEPT[2] [ReDa98] ist ein Prozess-Management-System (PMS) der nächsten Generation, das volle Funktionalität bezüglich Modellierung, Analyse, Ausführung und Überwachung von Geschäftsprozessen bietet [ReDa98, RRD04a, RRD04c]. Es ist eines der wenigen Systeme (wenn nicht gar das einzige), das volle Unterstützung für adaptive Prozesse sowohl auf der Prozessinstanz- als auch auf Prozesstypebene bietet. Bei der Durchführung von Instanzänderungen stellt *ADEPT* die Konsistenz und Korrektheit der Instanzen vor und nach der Änderung sicher. Im Kontext von Prozesstyp-Änderungen erlaubt es zusätzlich laufende Prozessinstanzen auf das neue Prozessmodell zu migrieren [RRD04d]. Es gibt einen mächtigen Prototyp, der diese Funktionalität demonstriert und der von vielen Forschungsgruppen und Industriepartnern genutzt wird.

ADEPT basiert auf dem so genannten *ADEPT Workflow Modell*. Wesentlich für die Spezifikation und Ausführung von Prozessen ist das Konzept symmetrischer Kontrollstrukturen (vgl. strukturierte Programmierung): Sequenzen, Verzweigungen (mit unterschiedlichen Split- und Join-Semantiken) und Schleifen sind als symmetrische Blöcke spezifiziert, mit wohldefinierten Start- und Endknoten. Diese Blöcke können beliebig verschachtelt sein, sie dürfen sich aber nicht überlappen. Zusätzlich wird die Synchronisierung von Schritten paralleler Zweige in einem Workflow Graph unterstützt. Eine detaillierte Beschreibung des *ADEPT Modells* ist außerhalb des Rahmens dieser Arbeit. Deshalb werden im Folgenden nur die Basiskonzepte von Kontroll- und Datenfluss eines Prozesses betrachtet.

2.1.1 Prozessmodellierung

Ein Workflow Schema besteht aus einer Reihe von Aktivitäten und Kontroll- als auch Datenabhängigkeiten zwischen ihnen. Im Folgenden werden nur einfache Schritte betrachtet, Aktivitäten, die nicht weiter geteilt werden können und deren Ausführung durch externe (nicht unbedingt menschliche) Bearbeiter erledigt werden muss.

---

[2] *ADEPT* steht für ***A**pplication **D**evelopment Based on **E**ncapsulated Premodeled **P**rocess **T**emplates*

Kontrollfluss

Der Kontrollfluss eines Prozesses wird als gerichteter, strukturierter Graph (*N, E*) abgebildet. Aktivitäten werden als Menge von Knoten *N* abstrahiert und Kontrollabhängigkeiten zwischen ihnen als Menge gerichteter Kanten *E*. Knoten und Kanten können jeweils verschiedene Typen haben. Jedes Workflow Schema hat einen eindeutigen Startknoten (vom Typ *STARTFLOW*) und einen eindeutigen Endknoten (vom Typ *ENDFLOW*). Der Startknoten hat keinen Vorgänger und der Endknoten keinen Nachfolger. Alle anderen Knoten aus *N* haben mindestens einen Vorgänger und einen Nachfolger. Die sequentielle Ausführung zweier Aktivitäten wird durch eine Kontrollkante modelliert. Die Modellierung von Verzweigungen zeigt Abbildung 2-1. Verzweigungsblöcke starten mit einem Split-Knoten und werden mit einem eindeutigen symmetrischen Join-Knoten synchronisiert. *ADEPT* unterstützt zwei Arten von Verzweigungen: Parallele Ausführung (AND-Split/AND-Join) und Alternative Ausführung (OR-Split/OR-Join). Die Entscheidung bei alternativer Ausführung (siehe Abbildung 2-1(b)) kann entweder datenbasiert sein oder vom Benutzer getroffen werden. Im zweiten Fall werden alle Knoten aktiviert wenn der Split-Knoten beendet ist. Sobald einer der Schritte für die Ausführung ausgewählt wurde werden die Arbeitsschritte der anderen von den entsprechenden Arbeitslisten gelöscht.

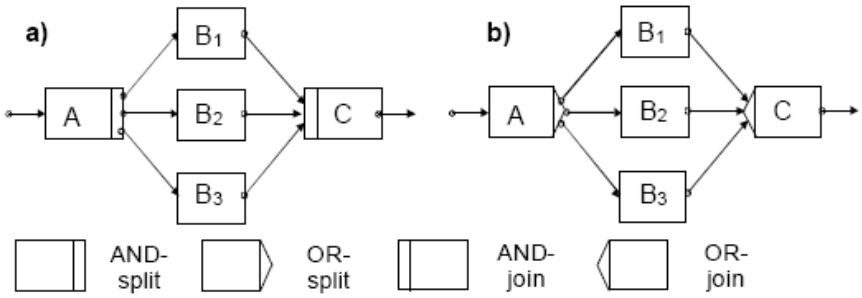


Abbildung 2-1: Parallele Verzweigung (a), Alternative Verzweigung (b) [ReDa98]

Mit Schleifen können sich wiederholende Ausführungen einer Menge von Schritten modelliert werden. Wie eine Verzweigung entspricht eine Schleife einem symmetrischen Block mit einem eindeutigen Startknoten (vom Typ *STARTLOOP*) und einem eindeutigen Endknoten (vom Typ *ENDLOOP*), die über eine Schleifenkante (vom Typ LOOP_EDGE) verbunden sind.

Die Ausdrucksmächtigkeit der bisher gezeigten Kontrollstrukturen ist nicht ausreichend für die Modellierung von Prozessen mit lang-andauernden, gleichzeitigen Ausführungen. Um die Synchronisation von Schritten unterschiedlicher Pfade paralleler Verzweigungen zu unterstützen wird eine spezielle Synchronisationskante (Typ: *SOFT_SYNC_EDGE*) verwendet. Solch eine Kante zwischen zwei Knoten n_1 und n_2 wird verwendet um eine Verzögerungsabhängigkeit zwischen den Schritten n_1 und n_2 festzulegen. Schritt n_2 darf erst dann ausgeführt werden, wenn die Bearbeitung von Schritt n_1 entweder abgeschlossen ist oder nicht mehr gestartet werden kann (weil er z.B. innerhalb einer alternativen Verzweigung liegt). Die Verwendung von Synchronisationskanten unterliegt bestimmten Einschränkungen um redundante Kontrollabhängigkeiten zwischen Schritten und Schleifen oder sogar Beendigungsprobleme (*ter-*

mination problems) des Prozesses zu vermeiden. In jedem Fall dürfen nur Knoten verschiedener Pfade einer parallelen Verzweigung durch solche Kanten synchronisiert werden. Darüber hinaus darf eine Synchronisationskante keinen Knoten innerhalb eines Schleifenkörpers mit einem Knoten außerhalb verbinden. Abbildung 2-2 zeigt ein Beispiel eines *ADEPT*-Prozesses mit einer Synchronisationskante zwischen den Schritten *E* und *H*. Schritt *H* kann nur gestartet werden, wenn Schritt *E* entweder beendet ist oder ausgelassen wird (wenn der entsprechende Pfad nicht ausgeführt wird).

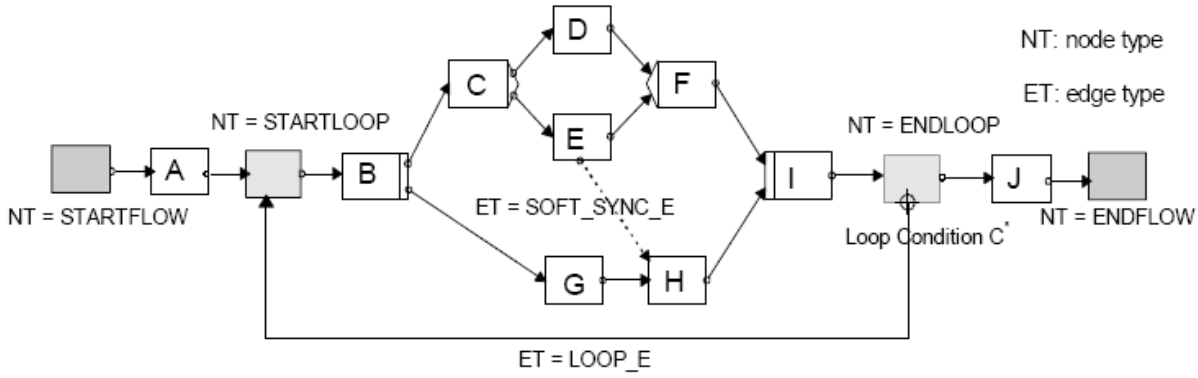


Abbildung 2-2: Beispiel eines *ADEPT*-Prozesses mit einer Synchronisationskante [ReDa98]

Datenfluss

Die Eingangs- und Ausgangsdaten von Aktivitäten und der Fluss der Daten zwischen ihnen sind wichtige funktionale Aspekte eines Workflow-Management-Systems. Im *ADEPT*-Modell basiert der Austausch von Daten zwischen Aktivitäten auf globalen Prozessvariablen. Ein Workflow Schema wird mit einer Menge von Datenelementen (Datencontainer) *D* verbunden, wobei jedes Element *d* aus *D* einen eindeutigen Bezeichner id^d und einen Wertebereich dom^d hat. Der Datenfluss zwischen Aktivitäten wird durch eine Verbindung ihrer Parameter mit *D* definiert. Zur Vereinfachung werden die Eingabe- (Ausgabe-) Parameter eines Workflow Schemas logisch als Ausgabe- (Eingabe-) Parameter seines Start- (End-) Knotens betrachtet. Abbildung 2-3 zeigt ein Beispiel eines einfachen Datenfluss-Schemas. Angenommen *G* hat Lesezugriff auf d_1. Obwohl der Schritt *C* d_1 schreiben könnte bevor *G* gestartet wird, würde dieser Wert für *G* nicht sichtbar sein. *G* darf nur auf den Wert von d_1 zugreifen, der durch den Startknoten des Prozesses geschrieben wurde. Grundsätzlich darf eine Aktivität nur die Werte eines Datenelements lesen, die von einer Aktivität geschrieben wurde, die im Kontrollfluss vor der lesenden Aktivität kommt.

Zusammenfassend wird ein Workflow Schema durch endliche, nicht leere Mengen *N* an Knoten und *E* an gerichteten Kanten dazwischen beschrieben. Dazu kommt noch die Menge an Datenelementen *D* und die Menge der Lese- und Schreibkanten *DF* zwischen Aktivitäten und Elementen von *D*.

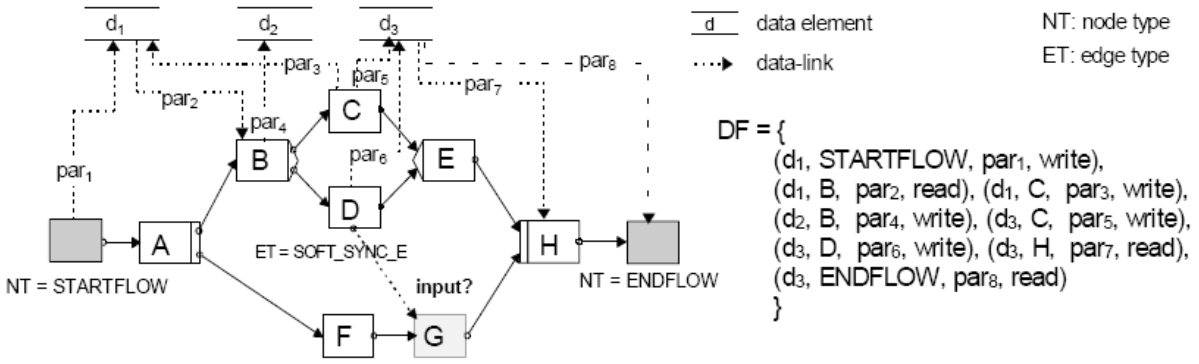


Abbildung 2-3: Beispiel eines einfachen Datenfluss-Schemas [ReDa98]

2.1.2 Prozessausführung und weitere Eigenschaften

Der Status einer Prozessinstanz ist eines der Hauptkriterien bei der Entscheidung, ob eine bestimmte strukturelle Änderung durchgeführt werden kann oder nicht. Außerdem könnten nach der Anwendung struktureller Änderungen an einem Workflow Graph begleitende Änderungen an den Zuständen seiner Knoten und Kanten notwendig werden, um mit dem Kontrollfluss fortfahren zu können.

ADEPT basiert auf wohldefinierten operationalen Semantiken um dies zu unterstützen. Der Status einer Prozessinstanz ist durch die aktuelle Markierung der Knoten und Kanten, durch die in den Datenelementen gespeicherten Werte und durch ihre Ausführungshistorie definiert. Wenn ein Prozess gestartet wird, wird der Startknoten als beendet markiert und seine ausgehende Kontrollkante wird wahr signalisiert (*TRUE_SIGNALED*). Immer wenn eine Kante markiert wird, wird der Status ihres Zielknotens neu bewertet, entsprechend den Ausführungsregeln die durch *ADEPT* definiert sind. Die Ausführungsregeln beschreiben die Bedingungen unter denen ein Knoten aktiviert werden kann, d.h. an die entsprechende Arbeitsliste gegeben werden kann. Die Beendigung einer Aktivität führt zur Signalisierung sowohl ihrer ausgehenden Kontroll- als auch ihrer ausgehenden Synchronisationskanten. Die Markierung der Kanten folgt wohldefinierten Signalisierungsregeln, die auf den operationalen Semantiken der verschiedenen Kontrollstrukturen basieren. Eine Prozessinstanz endet schließlich erfolgreich, wenn die eingehende Kontrollkanten seines Endknotens wahr signalisiert ist. Weitere Details werden an dieser Stelle ausgelassen und stattdessen ein Beispiel (siehe Abbildung 2-4) betrachtet.

Angenommen Aktivität *B* wird beendet und die ausgehende Kontrollkante *B→C* wird wahr signalisiert, und die Kante *B→D* falsch. Daraufhin werden die Knoten *C* und *D* neu bewertet und aktiviert bzw. übersprungen. Nach dem Auslassen von *D* wird sowohl die ausgehende Kontrollkante als auch die ausgehende Synchronisationskante falsch signalisiert (FALSE_SIGNALED). Folglich wird der Zustand von *G* neu bewertet und auf aktiviert gesetzt.

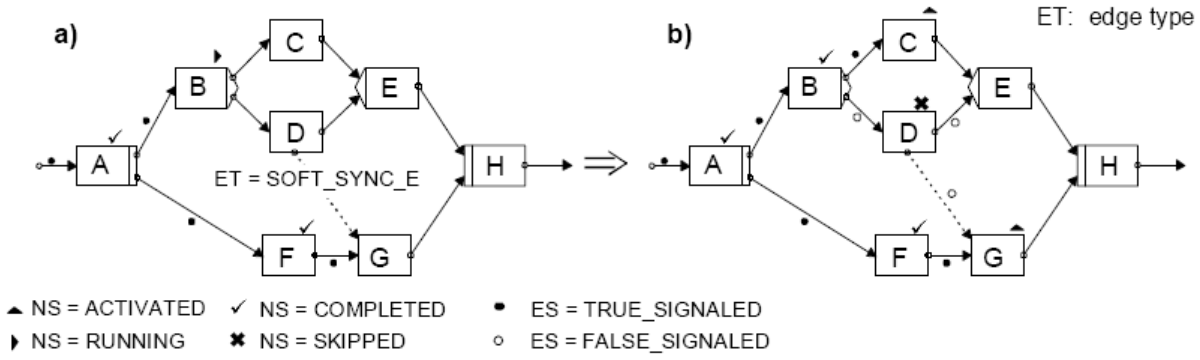


Abbildung 2-4: Synchronisation von Knoten aus verschiedenen Pfaden einer parallelen Verzweigung [ReDa98]

Formale Kriterien werden benötigt um mögliche Ausnahmen nach einer strukturellen Prozessänderung erkennen und Unterstützung für die Behandlung liefern zu können. Nachfolgend werden einige der Korrektheitseigenschaften vorgestellt, die durch *ADEPT* definiert sind.

In einem korrekten Kontrollfluss muss jeder Knoten *n* vom Startknoten aus erreichbar sein. Dazu muss es eine gültige Reihe von Signalisierungsereignissen geben die von der ursprünglichen Markierung zur Aktivierung von *n* führen. Außerdem muss von jedem erreichbaren Zustand des Prozesses der Endzustand (Aktivierung des Endknotens) erreicht werden können. Für nicht-zyklische Workflow Graphen die auf Sequenzen und symmetrischen Verzweigungen basieren sind diese Eigenschaften bereits durch die Konstruktion erfüllt.

Für den Datenfluss kann vereinfacht angenommen werden, dass für eine korrekte Ausführung einer Aktivität *A* alle Eingabeparameter versorgt sind und dass nach der erfolgreichen Beendigung alle Ausgabeparameter geschrieben sind. *ADEPT* erhebt eine Reihe von Beschränkungen die die Art eines korrekten Datenflussschemas regeln.

Basierend auf dem *ADEPT Modell* wurde eine Menge an Operationen (z.B. Einfügen, Löschen von Schritten) entwickelt, die als Rahmenwerk für dynamische Strukturänderungen von Prozessen dienen. Der Schwerpunkt bei der Entwicklung dieser Operationen wurde auf die Themen Korrektheit und Konsistenz gelegt. Die Anwendung einer Änderungsoperation auf einer Prozessinstanz muss in einem Prozess mit einem syntaktisch korrekten Schema und einem gültigen Zustand enden. Näheres zum *ADEPT Workflow Modell* und den möglichen Änderungsoperationen kann in [ReDa98] nachgelesen werden.

2.2 Realisierung: Demonstrator

In der Abteilung Datenbanken und Informationssysteme der Universität Ulm wurde zur systemseitigen Unterstützung von Prozessänderungen jeglicher Art ein umfassendes formales Rahmenwerk entwickelt [Laue04], welches prototypisch implementiert wurde. Dieser Prototyp, im Folgenden (*ADEPT*) *Demonstrator* genannt, basiert auf dem *ADEPT Prozessmodell* und unterstützt sowohl Prozessinstanz- und Prozesstyp-Änderungen als auch deren Verknüpfung, d.h. die Übertragung von Prozesstyp-Änderungen auf bereits individuell abgeänderte

Prozessinstanzen. Mit dem *Demonstrator* ist es darüber hinaus möglich, Templates zu erstellen und Instanzen davon auszuführen. Abbildung 2-5 zeigt einen Screenshot vom *Demonstrator* mit dem zugrunde liegenden Prozess, einer laufenden Instanz und den Einträgen in der Arbeitsliste.

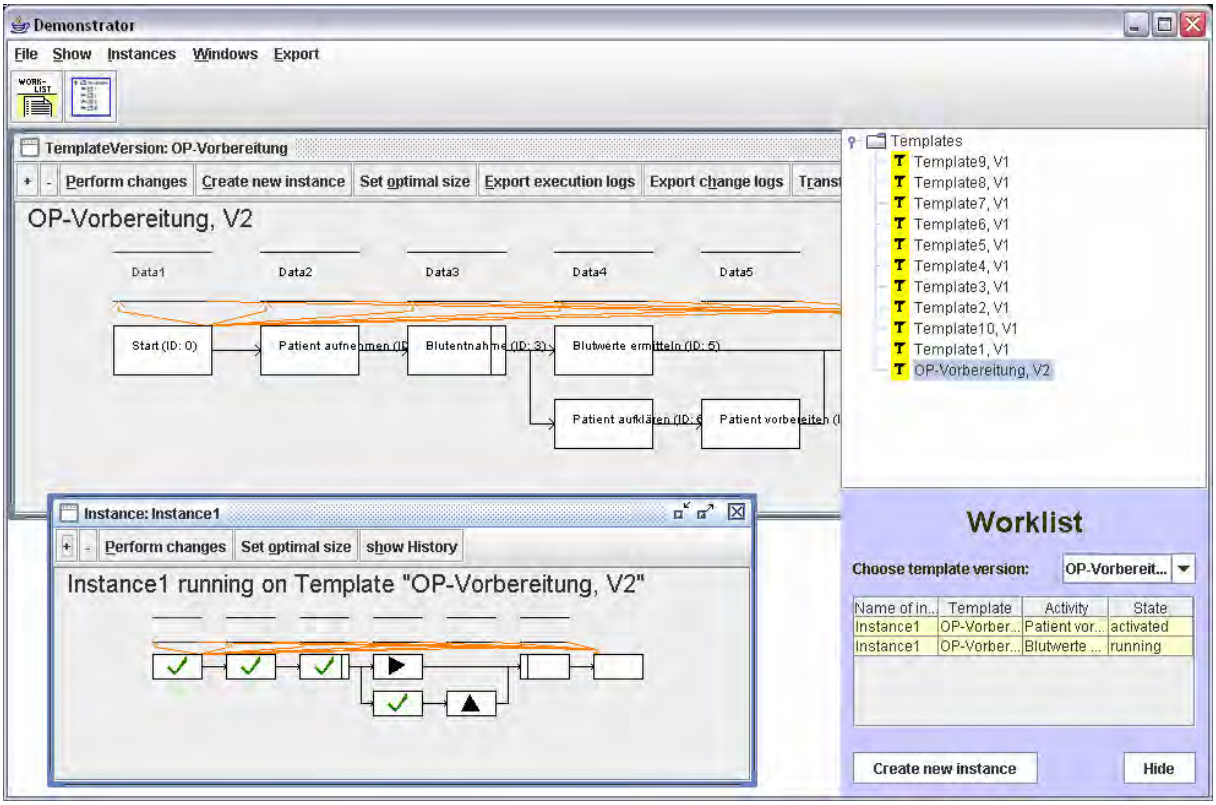


Abbildung 2-5: Beispiel *ADEPT Demonstrator*

Der *ADEPT Demonstrator* wurde im Rahmen dieser Diplomarbeit erweitert, um damit automatisch Log-Daten erzeugen zu können.

2.3 Process Mining - Theorie

Unter klassischem *Process Mining* versteht man die Gewinnung von strukturierten Prozessinformationen aus Transaktionsprotokollen abgeschlossener Prozesse und ausgeführter Anwendungen. Dazu wird angenommen, dass es möglich ist, Ereignisse so aufzuzeichnen, dass erstens jedes Ereignis zu einem (wohldefinierten) Workflow-Schritt gehört, zweitens jedes Ereignis zu einer Workflow-Instanz gehört und drittens die Ereignisse total geordnet sind. Jedes Informationssystem, das auf Transaktionen basiert (z.B. ERP, CRM, WfMS), bietet diese (Ereignis-)Informationen. So ist es für *Process Mining* nicht notwendig, dass ein WfMS vorhanden ist. Es wird nur angenommen, dass es möglich ist, Ablaufprotokolle mit Ereignisdaten zu sammeln [ADHM03, AWM02, AWM04, DMVW05, MAW03, MWAB02, WeAa01].

Das Hauptziel von *Process Mining* ist die effektive Nutzung dieser automatisch gesammelten Daten um aus den Log-Daten Prozessinformationen zu erlangen. *Process Mining* wird insbe-

sondere dazu verwendet, formale Prozessspezifikationen aus Ausführungsaufzeichnungen (*execution logs*) zu extrahieren. Bisher wurde *Process Mining* hauptsächlich für die Gewinnung des Kontrollflusses verwendet. Doch es wurden auch schon erste Ansätze entwickelt, die ereignisbasierte Daten auch für die Extraktion von Organisations- und Performanz-Aspekten nutzen [AaSo04, AaDo02].

Generell wird *Process Mining* bei der Erfassung von Prozessdaten als Alternative zu Interviews oder Fragebögen gesehen. Die Ergebnisse können für weitere Analysen verwendet werden. Zum Beispiel vergleicht die Delta-Analyse [ADHM03] existierende Prozessmodelle mit den Ergebnissen von *Process Mining* um Unterschiede zwischen dem Modell und der tatsächlichen Ausführung aufzudecken. Diese Information wird dann genutzt, um das Modell zu verbessern. Weil sich die Methoden zur Gewinnung der Prozessbeschreibung auf so genannte „Anwendungsfall-getriebene“ (*case-driven*) Prozesse heutiger Workflow-Management-Systeme konzentrieren, wird in diesem Zusammenhang auch der Begriff *Workflow Mining* verwendet [ADHM03]. Die Herausforderung beim *Mining* besteht darin, „gute“ Ablaufmodelle mit möglichst wenigen Informationen zu gewinnen.

Im Folgenden wird zuerst anhand eines kleinen Beispiels gezeigt, wie *Process Mining* funktioniert, danach werden Faktoren aufgezeigt, die das *Mining* erschweren, bevor ein gemeinsames Datenformat für Ausführungs-Log-Daten vorgestellt wird.

2.3.1 Ein kleines Beispiel

Anhand eines kleinen Beispiels soll das Prinzip des *Process Mining* näher erklärt werden. Als Ausgangspunkt dient ein vereinfachtes Ausführungsprotokoll (siehe Tabelle 1), das von Zeit, Datum und Ereignistyp (z.B. *Ereignisstart*, *Ereignisende*) abstrahiert und die Information auf die zeitliche Ablaufreihenfolge beschränkt. Ein typischer Staffware Protokolleintrag beinhaltet z.B. für jede Instanz eine kurze Beschreibung der Schritte, den Ereignistyp, den Bearbeiter, das Datum und die Zeit [ADHM03].

Das Protokoll in Tabelle 1 enthält Informationen über fünf Workflow-Instanzen (*case 1-5*). Das Log zeigt, dass für vier Fälle (*case 1-4*) die Schritte (*task*) A, B, C und D ausgeführt wurden. Für den fünften Fall (*case 5*) wurden nur drei Schritte ausgeführt: A, E und D. Jede Instanz startet mit der Ausführung von Schritt A und endet mit der Ausführung von D. Wenn der Schritt B ausgeführt wird, dann wird auch der Schritt C ausgeführt. Aber in einigen Fällen wird Schritt C auch vor Schritt B ausgeführt.

Tabelle 1: Ein (vereinfachtes) Ablaufprotokoll

Prozessinstanz Kennung	Prozessschritt Kennung
case 1	task A
case 2	task A
case 3	task A
case 3	task B
case 1	task B
case 1	task C
case 2	task C
case 4	task A
case 2	task B
case 2	task D
case 5	task A
case 4	task C
case 1	task D
case 3	task C
case 3	task D
case 4	task B
case 5	task E
case 5	task D
case 4	task D

Basierend auf der Information aus Tabelle 1 und einigen Annahmen bezüglich der Vollständigkeit des Protokolls (d.h. es wird angenommen, dass die Fälle repräsentativ sind und eine ausreichend große Teilmenge möglicher Ausführungsverhalten abdecken) lässt sich z.B. das Prozessmodell aus Abbildung 2-6 ableiten. Um vom Ablaufprotokoll zum Prozess zu kommen, können verschiedene Algorithmen verwendet werden. Welche Algorithmen das sind und wie sie funktionieren wird in Kapitel 3 vorgestellt.

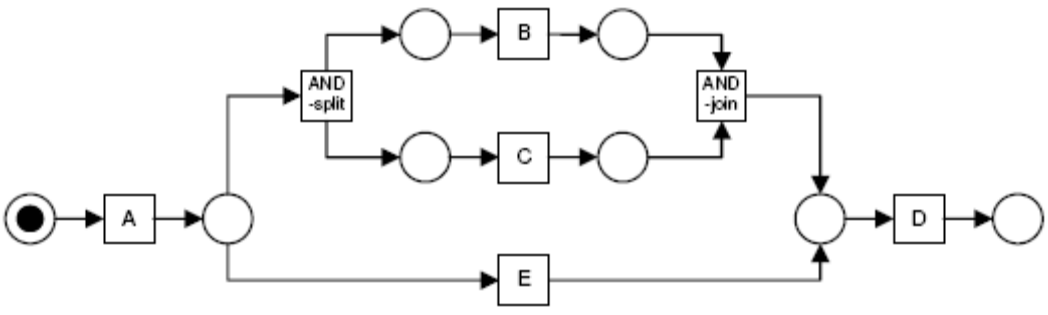


Abbildung 2-6: Ein zum Workflow Log (aus Tabelle 1) passendes Prozessmodell [ADHM03]

Der Graph wurde als Petrinetz modelliert [AWM02], die Startaktivität ist Schritt A, die Endaktivität Schritt D. Die Schritte werden als Transitionen dargestellt. Nach Ausführung von Schritt A besteht die Auswahl, ob entweder die Schritte B und C (zueinander parallel) ausge-

führt werden sollen oder nur der Schritt E. Um die Schritte B und C parallel ausführen zu können wurden zwei Schritte ohne assoziierte Aktivitäten (*silent actions, AND-split* und *AND-join*) hinzugefügt. Solche Schritte werden nur zur Strukturierung hinzugefügt und sind im Ausführungsprotokoll nicht vorhanden. Für dieses Beispiel wurde außerdem angenommen, dass zwei Schritte parallel ausgeführt werden, wenn sie in jeder beliebigen Reihenfolge im Log erscheinen. (Es gibt Fälle, in denen Schritt B vor C ausgeführt wird (1, 3) und Fälle, in denen Schritt C vor B ausgeführt wird (2, 4)) Wenn zwischen *Ereignisstart* und *Ereignisende* bei den einzelnen Schritten unterschieden wird, ist es möglich, Parallelität deutlich zu erkennen.

Das Workflow Log aus Tabelle 1 enthält das Minimale an Information (zeitlicher Ablauf, Unterscheidung der einzelnen Instanzen und deren Schritte), welche für *Process Mining* notwendig ist. Es wird angenommen, dass diese Information von jedem transaktionalen System zur Verfügung gestellt wird. In vielen Anwendungen enthält das Ausführungsprotokoll aber auch einen Zeitstempel für jedes Ereignis, was dazu verwendet werden kann, weitere Kausalitätsinformationen zu gewinnen. Außerdem enthält ein typisches Ablaufmodell auch Informationen über den Ereignistyp, z.B. Ereignisstart, Ereignisende, Ereignisabbruch, etc.

2.3.2 Einflussfaktoren

Für kleine Beispielprotokolle (z.B. Tabelle 1) ist es relativ leicht, ein Prozessmodell zu entwerfen, welches das Ablaufmodell nachbilden kann. Für realistischere Situationen gibt es mehrere Einflussfaktoren, die das *Process Mining* erschweren.

- Erschwerte Gewinnung großer Workflowmodelle: Für (große) Modelle mit Verzweigungen und parallelen Zweigen enthält das Ablaufprotokoll typischerweise nicht alle möglichen Kombinationen. Beispielsweise gibt es für 10 parallele Aktivitäten 10! = 3.628.800 verschiedene Permutationen. Es ist unrealistisch, dass alle Verschränkungen in einem Ablaufprotokoll auftauchen. Außerdem könnten einige Wege durch den Prozessgraphen eine geringere Ausführungswahrscheinlichkeit haben und deshalb nicht protokolliert sein.
- Verfälschungen (*Noise*): Workflow Logs enthalten typischerweise Störungen (*noise*), also unkorrekte oder unvollständige Teile oder Einträge, die sich auf Ausnahmebehandlungen beziehen. Inkorrekte Workflow Logs können sowohl auf menschlichen als auch auf technischen Fehlern beruhen. Ereignisse können fehlen, weil sie von Hand oder von einem anderen System / einer anderen Organisationseinheit bearbeitet wurden. Protokollierte Ereignisse können sich auch auf seltene oder unerwünschte Ereignisse beziehen. Es ist ebenfalls denkbar, dass zwei völlig unabhängige Ereignisse nacheinander ablaufen ohne eine Kausalbeziehung vorauszusetzen. Es ist nachvollziehbar, dass Ausnahmen, die nur einmal protokolliert wurden, nicht automatisch in den „normalen“ Ablaufplan aufgenommen werden sollten.
- Zusatzinformationen: Tabelle 1 zeigt nur die zeitliche Reihenfolge der Ereignisse ohne weitere Information über den Ereignistyp, die Ereigniszeit und die Attribute eines Ereignisses. Es ist klar, dass es schwieriger ist, auch solche Zusatzinformationen zu verwenden, wobei die zusätzlichen Informationen auch hilfreich sein können, bei der Gewinnung eines Prozesses.

Es gibt mittlerweile verschiedene Lösungsansätze, die diese Faktoren mehr oder weniger gut berücksichtigen und behandeln können. Wie gut z.B. die einzelnen Ansätze mit *Noise* umgehen können, wird in Kapitel 3.3 beschrieben.

2.3.3 Ein gemeinsames Datenformat: MXML

Die Basis für alle *Process Mining* Techniken ist ein Prozesslog. Solch ein Protokoll ist eine Datei die von irgendeinem Informationssystem erzeugt wurde und die Information über die Ausführung eines Prozesses enthält. Da jedes Informationssystem ein eigenes Format für die Speicherung von Log-Dateien hat, wurde ein allgemeines XML-Format entwickelt [DMVW05], um ein Protokoll darin zu speichern. Dieses Format, das *MXML* genannt wurde, basiert auf einem gründlichen Vergleich der Eingabeanforderungen verschiedener existierender *Process Mining* Tools und der Information, die typischerweise in einem Protokoll irgendeines komplexen Informationssystems enthalten ist. Es dient somit als Schnittstelle zwischen verschiedenen Informationssystemen (Prozess-Management-Systeme, Customer Relationship Management (CRM-) Systeme, Enterprise Resource Planning (ERP-) Systeme) und verschiedenen *Mining* Algorithmen [ADHM03]. Prinzipiell kann jedes System, das Ereignisse bzgl. der Ausführung von Aktivitäten protokolliert, dieses unabhängige Format nutzen, um Logs zu speichern und auszutauschen. *MXML* beansprucht nicht, alle möglichen Verhalten aller Systeme erfassen zu können, aber unter anderem für *Staffware*, *FLOWer*, *MS Exchange* und *MQSeries* existieren Adapter um deren Log-Daten auf das Format abzubilden. Auf der anderen Seite verwenden alle *Mining* Algorithmen in *ProM* (siehe Kapitel 2.4) *MXML* als Eingabeformat. Durch ein gemeinsames Format verringert sich der Implementierungsaufwand (u.a. bei der Entwicklung neuer *Mining* Techniken), da nicht die Log-Daten vieler unterschiedlicher Systeme eingelesen werden müssen, und die kombinierte Nutzung mehrerer verschiedener *Mining* Techniken wird erleichtert. *MXML* ist durch das in Abbildung 2-7 gezeigte XML-Schema[3] definiert und somit eine wohlgeformte und valide XML-Datei.

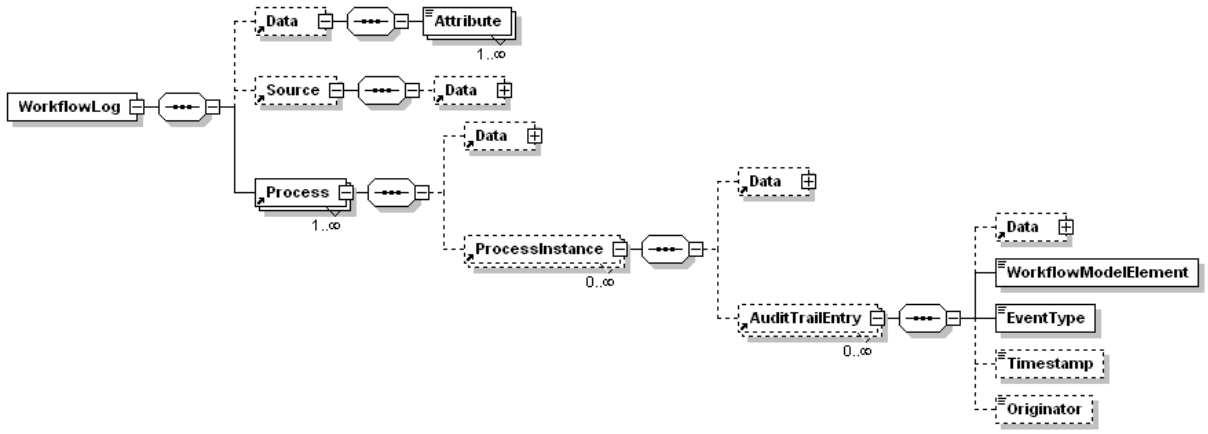


Abbildung 2-7: *MXML*-Schema (Ansicht in XMLSpy)

[3] vgl. www.processmining.org

Das übergeordnete Hauptelement einer *MXML*-Datei ist *WorkflowLog*. Ein Workflow Log kann optional Angaben zum Quellprogramm (*Source*) enthalten, muss aber aus mindestens einem Prozess (*Process*) bestehen, der wiederum mehrere Prozessinstanzen (*Processinstance*) haben kann. Für jedes protokollierte Ereignis gibt es ein so genanntes *AuditTrailEntry*. Hier steht die eigentliche Information über die aufgetretenen Ereignisse, dazu gehören zwingend das *WorkflowModelElement*, also Name und/oder ID der betreffenden Aktivität, und der *EventType*, also der Ereignistyp (z.B. *start, complete*). Für den Ereignistyp wurde ein transaktionales Modell [DMVW05] entwickelt, um über die Ereignisse auf eine standardmäßige Art und Weise reden zu können. Auch dieses Modell basiert auf einer Analyse von verschiedenen Arten von Logs in realen Systemen. Im Schema sind nun zwölf Ereignistypen (plus *unbekannt*) vorgesehen, auf die die Ereignisse in einem Prozess-Management-System abgebildet werden können. Optional kann für jedes Ereignis noch der Zeitpunkt (*Timestamp*) und der Benutzer (*Originator*) angegeben werden. Darüber hinaus können beliebige (z.B. ausführungsspezifische) Daten in den optionalen *Data* Elementen eingetragen werden, die es für Workflow Log, Prozess, Prozessinstanz und Protokolleintrag jeweils gibt. Sie enthalten eine Liste von Attribut-Elementen (*Attribute*).

Für die Umwandlung beliebiger Log-Daten in das *MXML*-Format kann die *MXMLib* Bibliothek[4] verwendet werden, die von den Entwicklern von *MXML* zur Verfügung gestellt wird. Im nächsten Abschnitt wird ein Rahmenwerk für *Process Mining* vorgestellt. Die darin umgesetzten Algorithmen verwenden alle *MXML* als Eingabeformat.

2.4 Process Mining – Realisierung: ProM

Unter dem Schirm von Schlagwörtern wie *Business Activity Monitoring* (BAM) und *Business Process Intelligence* (BPI) wurden sowohl akademische (z.B. *EMiT*, *Little Thumb*, *InWoLvE*, *Process Miner*, *MinSoN*) als auch kommerzielle Tools (z.B. *ARIS PPM*, *HP BPI*, *ILOG JViews*) entwickelt. Das Ziel dieser Tools ist das Gewinnen von Information aus Ereignisprotokollen (Transaktionslogs in ERP Systemen oder Ausführungsprotokolle in WFMS), in anderen Worten *Process Mining*. Leider nutzen diese Tools verschiedene Formate zum Lesen und Speichern von Log-Dateien und präsentieren ihre Ergebnisse auf verschiedene Art und Weise. Das macht es schwierig verschiedene Tools auf dem gleichen Datensatz anzuwenden und die Ergebnisse zu vergleichen. Außerdem implementieren einige dieser Tools Konzepte die in anderen Tools sehr nützlich sein können, es ist aber schwierig, sie zu kombinieren. Um diese Art von Problemen zu überwinden wurde das *ProM*-Rahmenwerk[5] [DMVW05] entwickelt, eine erweiterbare Umgebung für *Process Mining*.

Im Zusammenhang mit *Process Mining* wird zwischen drei verschiedenen Perspektiven unterschieden. Die Prozessperspektive konzentriert sich auf den Kontrollfluss, also die Anordnung der Aktivitäten. Die Organisationsperspektive betrachtet das Bearbeiterfeld im Log, d.h. wer als Ausführender eines Schrittes involviert ist und wie die einzelnen Bearbeiter zusammenhängen, während sich die Fallperspektive auf die Eigenschaften der Instanzen (Fälle)

[4] Die *MXMLib* Bibliothek ist mit einer Open Source Lizenz unter http://promimport.sourceforge.net/ verfügbar.

[5] Das *ProM* Rahmenwerk ist mit einer Open Source Lizenz unter http://prom.sourceforge.net/ verfügbar.

konzentriert. Solche Eigenschaften können gewählte Pfade sein, die Bearbeiter oder auch die Werte der entsprechenden Datenelemente. Nachdem es für die verschiedenen Perspektiven bereits Ad-Hoc-Tools gegeben hat, wurde ein flexibles Rahmenwerk (*ProM*) entwickelt, in das die verschiedenen Algorithmen für jede der Perspektiven eingebunden werden können.

ProM erreicht seine Flexibilität durch Plug-ins, jeder eingebundene *Mining* Algorithmus entspricht einem Plug-in. Ohne großen Aufwand können auch neue Plug-ins zum Rahmenwerk hinzugefügt werden. Dazu muss nur der Name des Plug-ins zu einer *ini*-Datei hinzugefügt werden, *ProM* selbst muss nicht geändert (vor allem nicht rekompiliert) werden. Darüber hinaus ist *ProM* flexibel bzgl. Input- und Output-Format und erlaubt das Wiederverwenden von Code für die Entwicklung neuer *Mining* Ideen. Momentan sind Plug-ins der folgenden Typen in *ProM* eingebunden:

- *Mining*-Plug-ins, sie implementieren irgendeinen *Mining* Algorithmus, z.B. *Alpha-Algorithmus*, *Multi-Phase Mining*. Alle *Mining* Plug-ins verwenden *MXML* als Eingabeformat. Als Ergebnis liefern sie z.B. ein Petrinetz oder eine Ereignis-Prozess-Kette (EPK).
- Export-Plug-ins, sie implementieren eine ‚Speichern unter' Funktionalität für einige Objekte, wie z.B. Prozessgraphen. Es gibt z.B. Plug-ins für das Exportieren von Petrinetzen, EPKs, *MXML*-Logs oder Datentabellen.
- Import-Plug-ins, sie implementieren eine ‚Öffnen' Funktionalität für exportierte Objekte bzw. Log-Daten externer Programme (z.B. für Instanz-EPKs von ARIS PPM[6]).
- Analyse-Plug-ins, sie implementieren typischerweise irgendeine Eigenschaftenanalyse für ein *Mining* Ergebnis. Für Petrinetze gibt es z.B. ein Plug-in, das Invarianten (für Stellen und Transitionen) und einen Deckungsgraph erzeugt. Es gibt aber auch Analyse Plug-ins, die Log und Modell vergleichen (Übereinstimmungstest).
- Umwandlungs-Plug-ins, sie implementieren Umwandlungen zwischen verschiedenen Datenformaten, z.B. von EPK zu Petrinetz und umgekehrt.

Einige dieser Plug-ins, hauptsächlich die *Mining*-Möglichkeiten (von *ProM* in Version 3.1) werden in Kapitel 3.3 näher betrachtet.

ProM ist also eine Zusammenfassung bisheriger *Mining* Tools, es ermöglicht eine gemeinsame Nutzung der Tools und kann damit als Nachfolger derselben gesehen werden. Es gestattet Entwicklern eigene Algorithmen mit bestehenden zu vergleichen und zu kombinieren. Darüber hinaus ermöglicht es *ProM* an viele existierende Tools, sowohl kommerzielle als auch öffentliche, anzukoppeln. Abbildung 2-8 zeigt einen Screenshot von *ProM*.

[6] Der ARIS Process Performance Manager (ARIS PPM) ist ein patentiertes Werkzeug zur Analyse, Bewertung und zum Monitoring von Unternehmensprozessen. Näheres dazu kann unter ‚http://www.ids-scheer.com/germany/products/aris_controlling_platform/49532' nachgelesen werden.

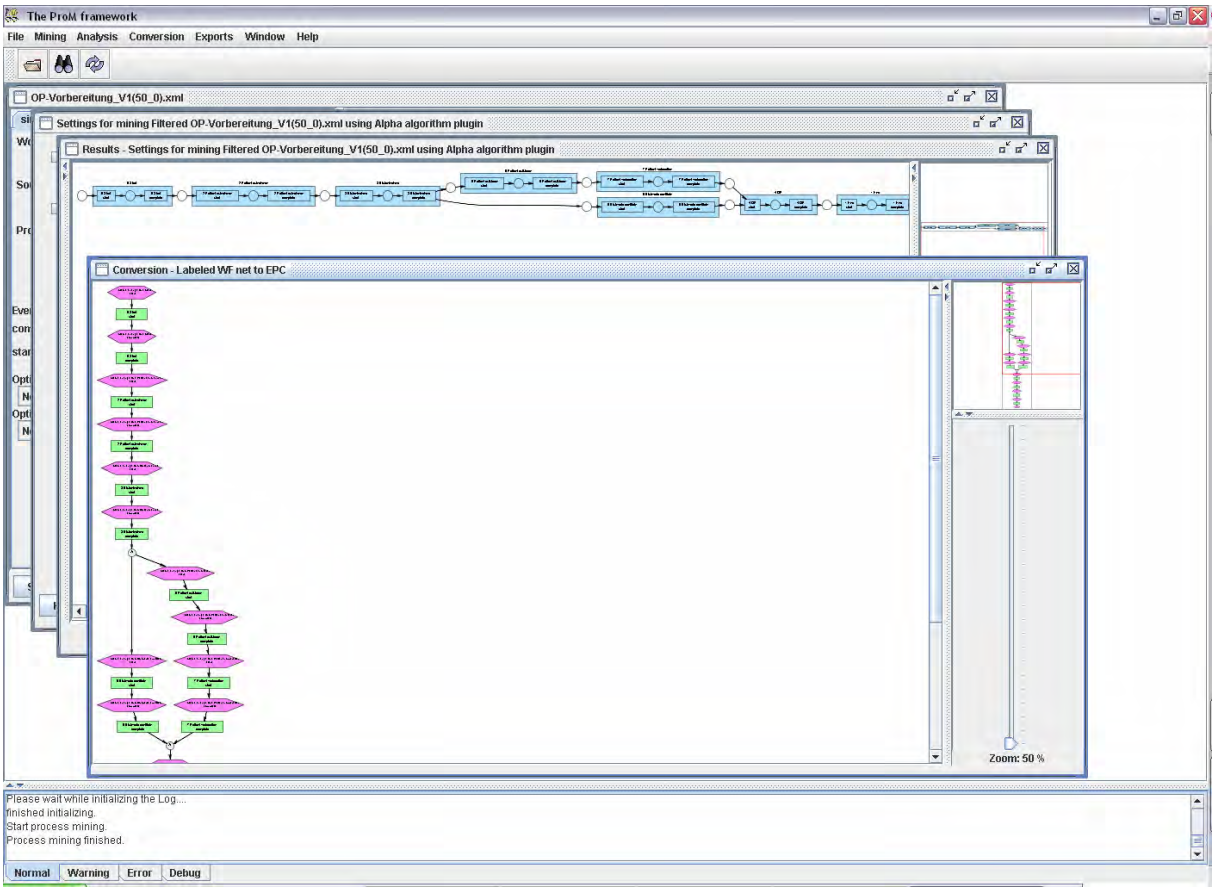


Abbildung 2-8: Das *ProM*-Rahmenwerk

2.5 Beispielprozess ‚OP-Vorbereitung'

Nachfolgend wird der in den folgenden Kapiteln verwendete Beispielprozess eingeführt und seine Details erläutert. Für die Erstellung (und das *Mining*) von Ausführungs-Logs (siehe Kapitel 3.2) wird nur der Kontrollfluss des Prozesses benötigt, da die meisten *Mining* Verfahren den Kontrollfluss eines Prozesses aus den Log-Daten extrahieren. Abbildung 2-9 zeigt den Prozess (Version 1) in seiner ursprünglichen, vom Modellierer entworfenen Struktur.
Nach der obligatorischen *Start*-Aktivität kommt die Aktivität ‚*Patient aufnehmen*'. Danach folgt die ‚*Blutentnahme*', nach deren Ende die Schritte ‚*Blutwerte ermitteln*' und ‚*Patient aufklären*' parallel stattfinden können. Nach der Aufklärung des Patienten, schließt sich die Aktivität ‚*Patient vorbereiten*' an, danach (und nachdem die Blutwerte ermittelt sind) findet die ‚*OP*' statt. Nach der ‚*OP*' folgt dann noch die obligatorische *End*-Aktivität.

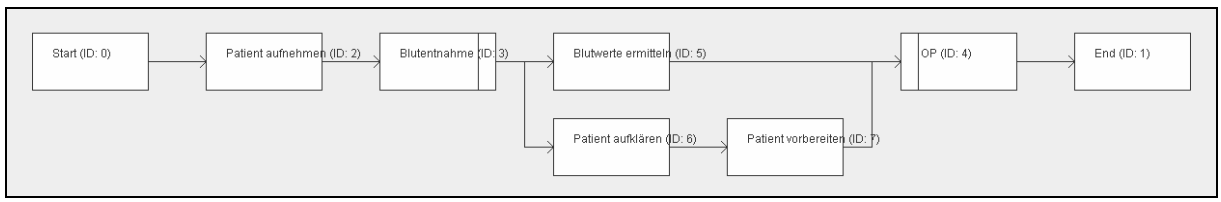


Abbildung 2-9: Beispielprozess OP-Vorbereitung, V1

Für die Erstellung von Änderungs-Logs (siehe Kapitel 5.2) wird zusätzlich zum Kontrollfluss auch der Datenfluss benötigt, da anhand der Werte in den Datencontainern entschieden wird, ob eine Änderung stattfinden soll oder nicht (z.B. Patient älter als 45 → Änderung findet statt). Zukünftig könnte man mit den Informationen aus dem Datenfluss z.B. feststellen, warum Änderungen durchgeführt wurden. Abbildung 2-10 zeigt den Prozess mit Kontroll- und Datenfluss (Version 2). Für den Datenfluss sind zusätzlich zum ursprünglichen Prozess noch sechs Datencontainer und die jeweiligen Schreib- und Lesekanten hinzugekommen, ansonsten hat sich nichts geändert. Insbesondere der Kontrollfluss ist immer noch der gleiche wie beim ursprünglichen Prozess. Auch bei realen Prozessen gibt es meist Daten (z.B. Patient Raucher, über 45), die Änderungen an einem Prozess beeinflussen. So kann es in bestimmten Fällen nötig sein, z.B. weitere Untersuchungen durchzuführen, abhängig davon, wie frühere Untersuchungen verlaufen sind. Im *ADEPT* Metamodell werden Datencontainer verwendet, um solche ‚Zwischenergebnisse' zu halten bzw. sie von einem Prozessschritt zum nächsten weiterzugeben. Die Datencontainer enthalten Datenwerte, die beim Beispielprozess alle von der *Start*-Aktivität geschrieben und von der *End*-Aktivität gelesen werden. Dies ist etwas unrealistisch, wurde aber der Einfachheit halber so gemacht, damit die Entscheidungsgrundlage auch für alle möglichen Änderungen rechtzeitig vorhanden ist. Soll z.B. die Aktivität ‚*Patient aufnehmen*' aus dem Prozess gelöscht werden, so muss das vor deren Start entschieden werden, und das geht nur, wenn der Entscheidungswert von der *Start*-Aktivität geschrieben wird.

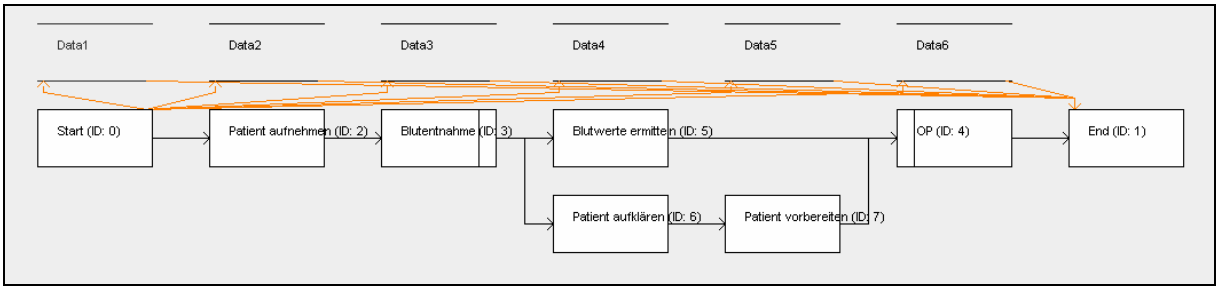


Abbildung 2-10: Beispielprozess OP-Vorbereitung, V2

3 Mining von Ausführungs-Logs – Evaluation

Nachdem nun die Grundlagen von adaptivem Prozess-Management und *Process Mining* vorgestellt wurden soll nun aufgezeigt werden, wie die beiden Techniken zusammen verwendet werden können. Um die Wirkungsweise von verschiedenen *Process Mining* Algorithmen vergleichen und bewerten zu können müssen zuerst einmal Log-Daten vorhanden sein. Hier liegt die Herausforderung in der Erstellung möglichst realitätsnaher Log-Daten, um auch Rückschlüsse auf reale Prozesse und reale Umgebungen treffen zu können. Damit möglichst einfach ausreichend viele Log-Daten erzeugt werden können, muss deren Erstellung automatisiert werden. Dabei sollte erreicht werden, dass alle Interaktionen mit dem Anwender und sämtliche Benutzereingaben entfallen und trotzdem sinnvolle Log-Daten entstehen. Erschwerend kommt hinzu, dass die Log-Daten realer Prozesse selten fehlerfrei und/oder vollständig sind. Deshalb sollte bei einer Simulation solcher realistischer Log-Daten solche Fehler und Verzerrungen nachgebildet werden können. Wenn z.B. manche Prozessschritte am System vorbei ausgeführt werden, weil der Prozess im PMS schlecht modelliert wurde, dann kommen solche Schritte nicht im Log vor, obwohl sie für die Ausführung des tatsächlichen Prozesses von Bedeutung sind. Auch bei Prozessen mit vielen parallelen Schritten sind selten alle theoretisch möglichen Ausführungsreihenfolgen im Log enthalten, da z.B. schon bei acht parallelen Schritten über (8!≈) 40000 verschiedene Ausführungsreihenfolgen möglich sind. Eine erste Herausforderung ist also die automatische Erzeugung beliebig vieler Log-Daten. Die zweite Herausforderung ist es, realistische Log-Daten mit Verzerrungen (z.B. ausgelassene Schritte) erzeugen zu können. Schließlich müssen die generierten Log-Daten noch in das *MXML*-Format (siehe Kapitel 2.3.3) gebracht werden, damit sie die *Process Mining* Umgebung *ProM* auch verwenden kann.

Wie die beschriebenen Herausforderungen zu bewältigen sind, wird im Folgenden beschrieben. Danach werden die Grundlagen für das *Mining* von Ausführungs-Logs, die Erzeugung und Transformierung von Log-Daten vorgestellt. Die erstellten Log-Daten werden anschließend verwendet, um verschiedene *Process Mining* Algorithmen zu evaluieren. Nach einem Fazit werden dann noch weitere Algorithmen und Möglichkeiten von *ProM* kurz vorgestellt.

3.1 Konzept für die Erzeugung von realitätsnahen Ausführungs-Log-Daten

Für die Erzeugung der Log-Daten soll der Benutzer nur angeben, wie viele Instanzen erzeugt werden sollen und wie viel *Noise* diese ggf. enthalten sollen. Sämtliche Entscheidungen, die während der Ausführung getroffen werden müssen, also welcher Eintrag von der Arbeitsliste als nächstes ausgewählt werden soll (u. A. bei UND-Verzweigungen) oder welcher Ausführungspfad beschritten werden soll (bei ODER-Verzweigungen), werden durch eine Zufallsfunktion entschieden. Diese wählt aus der jeweils aktuellen Menge der Möglichkeiten eine aus. Bei realen Prozessen hat meist der Benutzer die Möglichkeit, aus mehreren Schritten auf seiner Arbeitsliste den Schritt zu wählen, den er als nächstes bearbeiten möchte. Im Fall von Einträgen auf der Arbeitsliste müssen die nicht gewählten bei der nächsten Entscheidung wieder zur Wahl stehen, da sie ja weiterhin abgearbeitet werden müssen. Die Verwendung von Zufallsfunktionen ermöglicht also eine Erzeugung von Log-Daten, die der Realität nahe kom-

men, da auch hier nicht immer die gleiche Möglichkeit gewählt wird, wenn eine Auswahl zu treffen ist. Bei ODER-Verzweigungen z.B. entscheiden normalerweise die Kontextdaten, welcher Pfad weiterhin beschritten wird. Da dies vom *Demonstrator* aber nur ansatzweise unterstützt wird, muss hier mit einer Zufallsfunktion abgeholfen werden. Um die Ausführung der Instanzen weiterhin zu automatisieren müssen evtl. benötigte Datenfelder mit zufälligen Daten gefüllt werden, da eine Abfrage vom Benutzer den Aufwand zur Log-Erstellung hier deutlich in die Höhe treiben würde, wenn man von hunderten und tausenden Instanzen ausgeht, die so automatisch erzeugt werden können. Am einfachsten ist hier eine zufällige Zahl zwischen 0 und 99 zu realisieren, für die Erstellung von Ausführungs-Logs sind solch einfache Datenwerte aber mehr als ausreichend, da es sowieso nur auf den Kontrollfluss und nicht auf den Datenfluss ankommt.

Im Kontext dieser Arbeit sind fehlerfreie Log-Daten nur ein Aspekt, da davon ausgegangen werden kann, dass alle *Mining* Algorithmen bei einer ausreichenden großen Menge an fehlerfreien Log-Daten den ursprünglichen Prozess korrekt rekonstruieren können. Deshalb gibt es außer der Anzahl der Instanzen noch eine weitere Möglichkeit, die Log-Daten zu beeinflussen, und zwar durch die Angabe von *Noise*. Denn bei unvollständigen Logs zeigt sich schnell, welche Algorithmen mit diesem Umstand gut zurechtkommen und welche weniger gut. Dies ist speziell deshalb interessant, weil reale Log-Daten auch selten fehlerfrei sind bzw. nicht davon ausgegangen werden kann, dass sie fehlerfrei sind. Wird das *Process Mining* zur Erkennung eines Prozesses verwendet, kennt man den zugrunde liegenden Prozess gar nicht. Darum ist es wichtig zu wissen, wie sich ein bestimmter Algorithmus bei fehlerhaften Log-Daten verhält, um bei unbekannten Prozessen die richtigen Schlüsse ziehen zu können. Für die Erzeugung von *Noise* wird per Zufall entschieden, ob ein Ereignis in die Log-Datei aufgenommen wird, oder nicht. Es werden nämlich nur alle Einträge aufgenommen, deren Zufallswert über der angegebenen Schwelle liegt. D.h. wenn z.B. der Schwellwert mit 5 % angegeben ist, dann wird eine Zufallszahl ermittelt, und wenn diese über 5 % liegt, wird der Eintrag in die Log-Datei übernommen, ansonsten wird der Eintrag verworfen. Speziell bei kürzeren Prozessen kann sich der Wunschwert auch mal von der tatsächlich realisierten Menge *Noise* unterscheiden, was hauptsächlich an der verwendeten Zufallsfunktion liegt. Im Folgenden wird diese Form von *Noise* als *einfache Noise* bezeichnet.

Neben dieser einfachen Form von *Noise* gibt es noch eine weitere Möglichkeit, Verfälschungen in die Log-Daten zu bekommen. Durch die zusätzliche Verwendung von Logs geänderter Instanzen entstehen Log-Daten, die der Realität ziemlich nahe kommen. Das liegt an den realistischeren Änderungen an den Log-Daten. Anstelle des Weglassens einzelner Ereignisse können komplette Aktivitäten eingefügt, verschoben oder gelöscht werden. Die automatische Erstellung von Log-Daten veränderter Instanzen wird in Kapitel 4 behandelt. Für die Erstellung von Ausführungs-Logs reicht es an dieser Stelle zu wissen, dass solche Log-Daten erstellt und verwendet werden können. Der Anteil *Noise* kann durch das Verhältnis ‚normaler' Ausführungs-Logs (mit oder ohne *einfache Noise*) zu geänderten Ausführungs-Logs beeinflusst werden. Sollen z.B. Log-Daten mit 5 % *Noise* (ohne *einfache Noise*) erstellt werden, so müssen zum einen Log-Daten für eine größere Anzahl ‚normaler' Instanzen (z.B. 95) erstellt werden, zum anderen müssen die Log-Daten für die geänderten Instanzen (z.B. 5) erstellt werden. Danach müssen die erstellten Log-Daten gemeinsam in das *MXML*-Format transformiert werden. Näheres zur Transformierung kann in Kapitel 3.2.2 nachgelesen werden.

Abgesehen von der Anzahl der Instanzen und der Menge an *Noise* beeinflusst die Komplexität des zugrunde liegenden Graphen das Ergebnis der *Mining* Prozedur. Um diesen Einfluss bewerten zu können, wurden im Kontext dieser Arbeit mehrere unterschiedlich komplexe Beispielprozesse (zusätzlich zum Prozess ‚*OP-Vorbereitung*') erstellt und als Basis für die Erzeugung von Log-Daten verwendet.

Nach der Erstellung der Log-Daten ist die Umwandlung derselben in das *MXML*-Format ein zusätzlicher Schritt, da kein Prozess-Management-System seine Log-Daten bereits in diesem Format abspeichert, aber *ProM* dieses Format als Eingabe benötigt. Die Umwandlung soll direkt nach der Erzeugung der Log-Daten stattfinden, so dass kein weiteres Programm für die Transformierung benötigt wird. Für einige kommerzielle Prozess-Management-Systeme (z.B. Staffware) existiert solch ein Tool (*ProM Import*[7]) zur Umwandlung von Log-Daten in das benötigte *MXML*-Format.

3.2 Grundlagen für das Mining von Ausführungs-Logs

Der *ADEPT Demonstrator* als Vertreter eines adaptiven Prozess-Management-Systems wurde im Rahmen dieser Diplomarbeit erweitert, so dass er verwendet werden kann, um mit wenig Aufwand (für den Benutzer) Log-Daten für das *Process Mining* zu erstellen und diese in das benötigte Format (*MXML*) zu transformieren.

3.2.1 Die Erstellung von Ausführungs-Logs

Um Ausführungs-Logs zu erzeugen, muss entweder zuerst ein neues Template erstellt werden, oder ein bereits vorhandenes Template in den *Demonstrator* geladen werden. Wenn man nun ein Template angezeigt bekommt, gibt es die Option „*Export execution logs*" (siehe Abbildung 3-1).

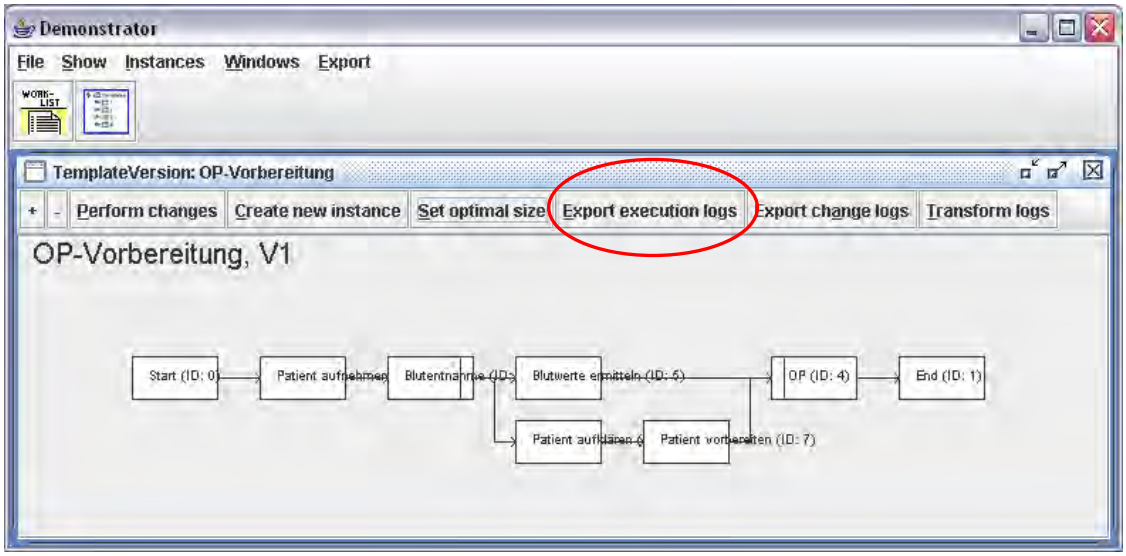


Abbildung 3-1: *Demonstrator* mit ausgewähltem Prozess ‚OP-Vorbereitung, V1'

[7] *ProM Import* ist mit einer Open Source Lizenz unter http://promimport.sourceforge.net/ verfügbar.

Klickt man diese Schaltfläche an, erscheint ein Hilfsdialog, in dem angezeigt wird, welches Template ausgewählt ist (siehe Abbildung 3-2). Des Weiteren muss der Benutzer nun angeben, wie viele Instanzen erzeugt werden sollen und gegebenenfalls, wie viel Prozent *einfache Noise* die Log-Dateien haben sollen. Mittels einer Checkbox kann der Benutzer angeben, ob die erzeugten Log-Dateien sofort in das *MXML*-Format umgewandelt werden sollen (Häkchen gesetzt), oder ob die Transformierung erst später stattfinden soll (Häkchen entfernt).

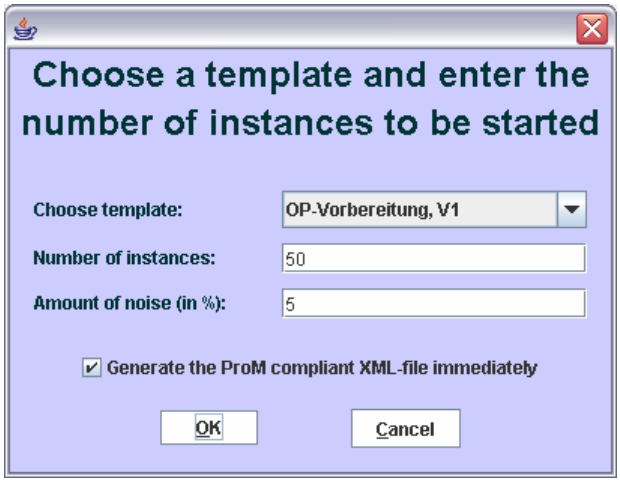


Abbildung 3-2: Hilfsdialog für die Erzeugung von Ausführungs-Logs

Wenn der Benutzer schon weiß, welches bestehende Template er für die Log-Erzeugung verwenden will, kann er auch direkt über den Menüeintrag *Export → Export Execution Logs* den Hilfsdialog (siehe Abbildung 3-2) aufrufen und dann in der Combobox das entsprechende Template auswählen.

Wenn alle Angaben gemacht sind und der Benutzer *OK* angeklickt hat, werden im Hintergrund die angegebene Anzahl Instanzen erzeugt und ausgeführt und Log-Dateien mit der angegebenen Menge *Noise* erzeugt. Im Beispiel sollen also 50 Instanzen mit 5 % *einfacher Noise* erzeugt werden und die erstellten Log-Daten auch gleich in das *MXML*-Format umgewandelt werden (Häkchen gesetzt). Werden die erstellten Log-Dateien sofort transformiert dann können sie nur Verfälschungen aus *einfacher Noise* enthalten, da für *Noise* aus Ad-Hoc-Änderungen (also Einfügen, Verschieben, Löschen von Aktivitäten) ein weiterer Erzeugungsschritt notwendig ist (genaueres dazu in Kapitel 5.2).

Der *Demonstrator* erstellt nach dem Bestätigen mit *OK* erstmal 50 Instanzen und führt diese aus, d.h. jede Aktivität jeder Instanz wird gestartet und beendet. Jedes dieser Ereignisse wird in den Log-Daten festgehalten, sofern keine *einfache Noise* gewünscht ist oder der Schwellwert der *Noise* überschritten ist. Nach diesem ersten Schritt gibt es für jede Instanz eine Log-Datei, im Beispiel werden also 50 XML-Log-Dateien erzeugt. Nachfolgend ist der relevante Ausschnitt aus einer (der 50) Log-Dateien aufgeführt.

```
<LoggingEntries>
  <StartTag activityID="0" date="1145458734210" />
  <EndTag activityID="0" date="1145458734220" />
  <StartTag activityID="2" date="1145458734240" />
```

```
    <EndTag activityID="2" date="1145458734270" />
    <EndTag activityID="3" date="1145458734270" />
    <StartTag activityID="6" date="1145458734280" />
    <StartTag activityID="5" date="1145458734290" />
    <EndTag activityID="5" date="1145458734290" />
    <EndTag activityID="6" date="1145458734290" />
    <StartTag activityID="7" date="1145458734300" />
    <EndTag activityID="7" date="1145458734300" />
    <StartTag activityID="4" date="1145458734310" />
    <EndTag activityID="4" date="1145458734310" />
    <StartTag activityID="1" date="1145458734320" />
  </LoggingEntries>
```

StartTag markiert den Zeitpunkt, wann die Ausführung der Aktivität gestartet wurde und *EndTag* den Zeitpunkt, wann die Ausführung abgeschlossen wurde. Dahinter steht jeweils auf welche Aktivität sich der Eintrag bezieht und der genaue Zeitpunkt[8] des Eintrags. Weil im Hilfsdialog angegeben wurde, dass ein *Noise*-Faktor von 5 % verwendet werden soll, sind die Einträge in der XML-Datei auch nicht vollständig. So fehlt im Beispiel der *StartTag* von Aktivität „*3*“. Der *EndTag* von Aktivität „*1*“ fehlt, weil der *Demonstrator* nur Log-Dateien von Instanzen speichern kann, die noch nicht komplett abgeschlossen sind. Dieser *EndTag* wird aber bei der Umwandlung der einzelnen XML-Dateien zu einer *MXML*-Datei noch eingefügt. Das ist für die Praxis auch nicht weiter von Bedeutung, da Aktivität „*1*“ im *Demonstrator* per Definition die Endaktivität ist, die jeder Prozess haben muss und für die auch keine Aktivität mehr hinterlegt ist.

3.2.2 Ausführungs-Logs transformieren

Nachdem nun von allen Instanzen Ausführungs-Logs erzeugt wurden (im Beispiel wurden 50 Instanz-Dateien erstellt), müssen diese in einem zweiten Schritt in eine *ProM*-konforme *MXML*-Datei übersetzt werden, d.h. die erzeugte XML-Datei muss dem *MXML*-Schema (siehe Kapitel 2.3.3) entsprechen.

Die einzelnen protokollierten Ereignisse werden nach den jeweiligen Prozessinstanzen gruppiert in denen sie stattgefunden haben. Für jedes *StartTag* und jedes *EndTag* aus der ursprünglichen Log-Datei wird nun je ein *AuditTrailEntry* angelegt. Dabei entspricht das *StartTag* dem Ereignistyp *start* und das *EndTag* dem Ereignistyp *complete*. Alle weiteren im *MXML*-Schema vorgesehenen Ereignistypen werden nicht benötigt, da der *Demonstrator* nur diese zwei Ereignisse aufzeichnet. Im ursprünglichen Log ebenfalls enthalten sind die ID der Aktivität, zu der das Ereignis aufgezeichnet wurde, und der genaue Zeitpunkt des Ereignisses. Diese Daten werden in der zu erzeugenden *MXML*-Log-Datei für *WorkflowModelElement* und *Timestamp* verwendet.

Weil zusätzliche Daten für das *Mining* von Ausführungs-Logs nicht benötigt werden und der *Demonstrator* auch keine zusätzlichen ausführungsspezifischen Daten liefert, werden in diesem Zusammenhang die optionalen *Data*-Elemente des *MXML*-Schemas nicht benötigt.

[8] Die Zeit wird in computerlesbarer Form gespeichert. Es handelt sich hierbei um die Zeit in Millisekunden, die seit dem 01.01.1970 0:00 Uhr GMT verstrichen ist.

Im Folgenden ist ein Ausschnitt aus der erzeugten *MXML*-Datei aufgeführt.

```
<ProcessInstance id="EXECLOG_OP-Vorbereitung_39">
  <AuditTrailEntry>
    <WorkflowModelElement>0 Start</WorkflowModelElement>
    <EventType>start</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>0 Start</WorkflowModelElement>
    <EventType>complete</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>2 Patient aufnehmen</WorkflowModelElement>
    <EventType>start</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>2 Patient aufnehmen</WorkflowModelElement>
    <EventType>complete</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>3 Blutentnahme</WorkflowModelElement>
    <EventType>complete</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>6 Patient aufklären</WorkflowModelElement>
    <EventType>start</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>5 Blutwerte ermitteln</WorkflowModelElement>
    <EventType>start</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <AuditTrailEntry>
    <WorkflowModelElement>5 Blutwerte ermitteln</WorkflowModelElement>
    <EventType>complete</EventType>
    <Timestamp>2006-04-19T14:58:54Z</Timestamp>
  </AuditTrailEntry>
  <!--  ...  -->
</ProcessInstance>
```

Wie man sieht, enthält die *ProM*-konforme *MXML*-Datei auch nicht mehr Informationen als die einzelnen Instanz-Logs zusammen. Nur aus Gründen der besseren Lesbarkeit wurde die Aktivitäts-ID um den Namen der Aktivität erweitert und bildet nun ein *WorkflowModelElement*. Wie bereits erwähnt wird ein *StartTag* auf den *EventType start* und ein *EndTag* auf den *EventType complete* abgebildet. Und der Zeitstempel muss für *ProM* in ein anderes Format umgewandelt werden, enthält aber dadurch auch nicht mehr Informationsgehalt.

Die Transformierung kann auf zwei Arten gestartet werden. Zum einen können die erstellten Instanzen gleich nach der Erstellung automatisch transformiert werden, dazu muss im Hilfsdialog zur Log-Erzeugung (siehe Abbildung 3-2) nur das Häkchen bei der automatischen Generierung gesetzt werden. Ist das Häkchen gesetzt, werden die Instanzen erzeugt und sofort im Anschluss werden die erzeugten Instanzen für die Transformierung verwendet. Bei der automatischen Transformierung werden auch nur die soeben erzeugten Instanzen umgewandelt. Sollen zusätzlich auch andere Instanzen mit umgewandelt werden oder die Instanzen nur zu einem späteren Zeitpunkt transformiert werden, dann kann dazu der Hilfsdialog für die Log-Transformierung verwendet werden (siehe Abbildung 3-3). In diesem Dialog kann angegeben werden, ob nur automatisch generierte Ausführungs-Logs oder auch automatisch veränderte Logs oder ob sogar manuell erstellte Log-Daten für die Transformierung verwendet werden sollen. Dazu kann dann noch der Zielordner für die Ausgabedatei angegeben werden. Dieser Dialog kann entweder über den entsprechenden Menüeintrag *Export → Transform XML-logs* oder über die Schaltfläche „*Transform logs*“, die bei der Anzeige eines Templates vorhanden ist, gestartet werden.

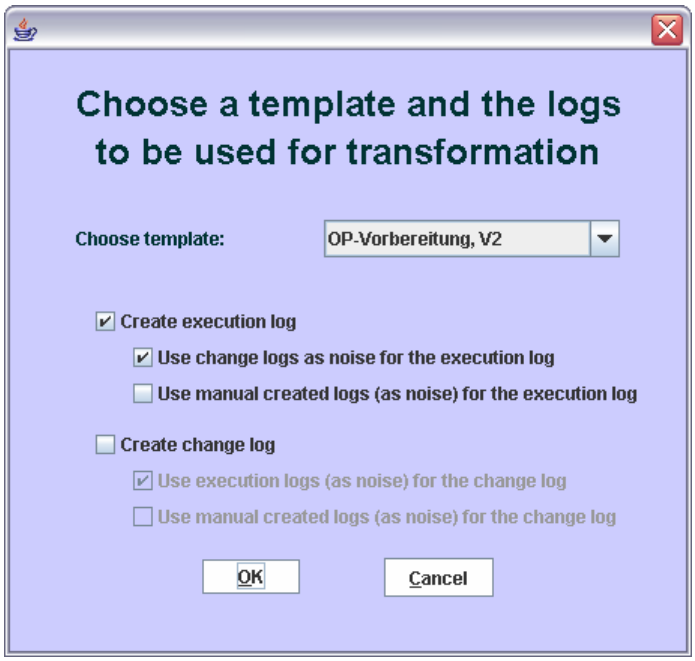


Abbildung 3-3: Hilfsdialog für die Log-Transformierung

Für die Transformierung selbst werden alle Instanzen verwendet, die zu der ausgewählten Prozessvorlage (Template) gehören, und die den entsprechenden Gruppen angehören, also entweder Ausführungs-Logs, Änderungs-Logs oder manuelle Log-Daten. Es können nur Instanzen verwendet werden, die auf der Festplatte gespeichert sind, da die benötigten Informationen direkt aus den XML-Dateien der einzelnen Instanzen entnommen werden. Wurden also z.B. in einem vorhergehenden Schritt die Ausführungs-Logs für 100 Instanzen erzeugt, werden bei der Transformierung derselben 100 XML-Dateien eingelesen und die entsprechenden Informationen in eine einzelne *MXML*-Datei übernommen. Dementsprechend kann die Transformierung bei vielen Instanzen und je nach Rechnerleistung etwas länger dauern. (Die Er-

zeugung und Transformierung von 5000 Instanzen kann durchaus 2-3 Stunden dauern[9], weil die 5000 Instanzen erstmal alle im Speicher gehalten werden, bevor sie am Ende der Erzeugungsphase auf Platte gespeichert werden.)

Die Umwandlung selbst erfolgt durch zu Hilfenahme der *MXMLib* Bibliothek (siehe Kapitel 2.3.3), die von den Entwicklern von *ProM* zur Verfügung gestellt wird. Diese Bibliothek ermöglicht eine relativ einfache Umwandlung der eigenen Log-Dateien in das Format, das *ProM* versteht (*MXML*). Der große Vorteil dieser Bibliothek ist die einfache Wartbarkeit, da auf eine Änderung des Formats nur der Austausch der Bibliothek erfolgen muss. Dies ist deutlicher leichter zu realisieren, wie den eigenen Code bei jeder Änderung des Log-Formats anzupassen.

3.3 Evaluierung von Mining-Verfahren

Nachdem nun mit dem *Demonstrator* eine *MXML*-Log-Datei erzeugt wurde, kann mit dieser Datei in *ProM* weitergearbeitet werden. Im Kontext dieser Arbeit ist es dabei besonders interessant, welche verschiedenen Möglichkeiten es gibt, um aus der Datei wieder ein Prozessmodell zu gewinnen. Die wichtigsten werden im Folgenden näher beschrieben. Als Ausgangsbasis dient weiterhin der Prozess ‚*OP-Vorbereitung, V1*', wie er in Abbildung 3-1 zu sehen ist. Dieser einfache Prozess mit einer parallelen Verzweigung wurde benutzt, um damit Log-Dateien zu erzeugen. Um die einzelnen Algorithmen vergleichen und bewerten zu können, werden drei verschiedene Kriterien verwendet: Die Anzahl der Instanzen, die Menge an *Noise* und die Komplexität des zugrunde liegenden Prozesses.

Über die Anzahl der Instanzen kann bestimmt werden, wie viel Informationen ein Algorithmus für die Rekonstruierung eines Prozesses bekommt. Ein guter Algorithmus kommt dabei mit relativ wenig Information aus. Um Unterschiede feststellen zu können wurden für die Beispieldaten zwischen 10 und 1000 Instanzen verwendet. Die Anzahl der Instanzen kann direkt als Maßzahl für den Vergleich der einzelnen Algorithmen verwendet werden.

Anhand von *Noise* kann festgestellt werden, wie anfällig ein Algorithmus für fehlerhafte Log-Daten ist. Bei fehlerfreien Log-Daten und einer ausreichend großen Datenmenge sollte jeder Algorithmus in der Lage sein, den ursprünglichen Prozess korrekt zu rekonstruieren. Weil aber reale Prozesse nicht immer fehlerfrei sind, ist es interessant zu wissen, wie sich die einzelnen Algorithmen bei fehlerhaften Log-Daten verhalten. Durch *Noise* können Fehler in die erzeugten Log-Daten eingebracht werden, einmal durch Weglassen bestimmter Ereignisse oder durch Ad-Hoc-Änderungen (Einfügen, Verschieben und/oder Löschen einzelner Aktivitäten). Je mehr *Noise* in den Log-Daten vorhanden ist, umso schwieriger wird es für die Algorithmen, den zugrunde liegenden Prozess zu erkennen. Ein in Bezug auf *Noise* guter Algorithmus kann trotz einer gewissen Menge an *Noise* den ursprünglichen Prozess wieder herstellen. Interessant ist in dieser Hinsicht auch, wie sich die unterschiedlichen Arten von *Noise* auf das Ergebnis auswirken.

Weil reale Prozesse oftmals größer und komplexer sind als der Beispielprozess ‚*OP-Vorbereitung, V1*', sollen Beispiele mit komplexeren Prozessen zeigen, wie gut sich die *Mi-*

[9] Auf einem 2,0 GHz Intel Centrino Notebook mit 1024 MB Hauptspeicher und Windows XP Professional.

ning Ergebnisse auf größere Prozesse übertragen lassen. Komplexere Prozesse sind in diesem Zusammenhang Prozesse mit mehreren parallelen oder alternativen Verzweigungen, Verschränkungen davon und Prozesse mit Synchronisations-Kanten (siehe Kapitel 2.1). Ein in Hinsicht auf komplexe Prozesse guter Algorithmus kann mit relativ wenigen Informationen den Prozess korrekt rekonstruieren. Die Anzahl der benötigten Instanzen dient auch hier wieder als Maßzahl, weil mit einer ausreichenden Menge an Informationen auch die komplexesten Prozesse wieder exakt hergestellt werden können.

Im Folgenden werden vier in *ProM* eingebundene *Mining* Algorithmen bezüglich dieser Kriterien untersucht und verglichen.

3.3.1 Alpha-Algorithmus

Der *Alpha-Algorithmus* [AWM04, MDAW04] ist einer der ersten Algorithmen, die implementiert wurden, um Prozesse aus Ereignisdaten zu gewinnen. Als Ergebnis liefert er ein so genanntes *Workflow-Netz*, ein Petrinetz, das den Kontrollfluss eines Prozesses abbildet. Petrinetze werden in *ProM* verwendet, weil sie theoretisch fundiert sind und deshalb eine gute Grundlage für Analysen rund um den Prozess sind.

Der *Alpha-Algorithmus* basiert auf vier Anordnungsbeziehungen, die aus den Log-Daten abgeleitet werden können, $>_W$, $\rightarrow_W$, $\#_W$ und $||_W$.

- $A >_W B$ trifft zu, wenn es (mindestens) eine Ausführungsreihenfolge gibt, in der B direkt auf A folgt. Das muss aber nicht bedeuten, dass eine Abhängigkeitsbeziehung zwischen A und B besteht, denn A und B können auch in einer parallelen Beziehung stehen.
- $A \rightarrow_W B$ trifft zu, wenn $A >_W B$ und $B \not> _W A$ zutreffen.
- $A \#_W B$ trifft zu, wenn sowohl $A \not> _W B$ als auch $B \not> _W A$ zutreffen.
- $A ||_W B$ trifft zu, wenn sowohl $A >_W B$ als auch $B >_W A$ zutreffen.

Die Beziehung $\rightarrow_W$ zeigt eine Abhängigkeitsbeziehung an und die Beziehungen $\#_W$ und $||_W$ werden verwendet, um zwischen Auswahl/Verzweigung ($\#_W$) und Parallelität ($||_W$) zu unterscheiden. Weil alle diese Beziehungen von $>_W$ abgeleitet werden können, genügt es anzunehmen, dass die Log-Datei in Bezug auf $>_W$ vollständig ist. Damit die Beziehungen aus dem Log erfolgreich extrahiert werden können, müssen bestimmte Voraussetzungen erfüllt sein (die Ausgangsprozesse müssen z.B. fehlerfrei und nach bestimmten Kriterien strukturiert sein), die aber von den meisten real vorkommenden Prozessen erfüllt werden.

Basierend auf den Abhängigkeitsbeziehungen funktioniert der (ursprüngliche) *Alpha-Algorithmus* folgendermaßen:

1. Die Log-Daten werden untersucht und die Menge der Transitionen ($\hat{=}$ den protokollierten Ereignissen, also *start* und *complete*) im Workflow wird erstellt.
2. Danach wird die Menge der Ausgangstransitionen der Quelle ($\hat{=}$ Startknoten, da es bei *ADEPT* nur einen Startknoten gibt) und
3. die Menge der Eingangstransitionen der Senke ($\hat{=}$ Endknoten, da es bei *ADEPT* nur einen Endknoten gibt) erzeugt.

4. Jetzt findet der *Alpha-Algorithmus* heraus, welche Transitionen voneinander abhängen und bestimmt so die möglichen Stellen im Graph.
5. Aus der Menge der möglichen Stellen werden die Stellen entfernt, die in einer anderen Beziehung enthalten sind. Dadurch erhält man die exakte Menge an Stellen, die der Graph benötigt (außer Quelle und Senke).
6. Diese Stellen werden nun erstellt und
7. mit den entsprechenden Transitionen verbunden.
8. Schließlich wird das entdeckte *Workflow-Netz* als Menge von Transitionen, Stellen und Flussbeziehungen zurückgegeben.

Weil der Algorithmus in dieser Form nicht in der Lage ist, kurze (ein- und zweielementige) Schleifen zu erkennen, wurde der ursprüngliche *Alpha-Algorithmus* später erweitert. In *ProM* ist bereits der erweiterte *Alpha+-Algorithmus* implementiert, der in [MDAW04] entwickelt wurde. Um kurze Schleifen erkennen zu können, müssen zuerst die Voraussetzungen an die Vollständigkeit der Log-Daten angepasst werden, dann müssen die Anordnungsbeziehungen umdefiniert werden und neue eingeführt werden. Mit diesen Änderungen können nun zweielementige Schleifen erkannt werden. Damit auch einelementige Schleifen erkannt werden können, müssen vor und nach der Anwendung des Algorithmus weitere Schritte durchgeführt werden (*pre-/post-processing*). So müssen einelementige Schleifen zuerst aus den Log-Daten entfernt werden, bevor der *Alpha-Algorithmus* die Grundstruktur des Graphen erkennen kann. Nach der Anwendung des Algorithmus müssen diese einelementigen Schleifen wieder in den Prozess eingebunden werden. Weil der *Demonstrator* Schleifen nicht unterstützt, soll an dieser Stelle nicht näher auf die einzelnen Schritte des *Alpha+-Algorithmus* eingegangen werden. Die Einzelheiten dazu können in [MDAW04] nachgelesen werden. Nachfolgend ist mit *Alpha-Algorithmus* der *Alpha+-Algorithmus* gemeint, so wie er in *ProM* implementiert wurde.

Abhängig von der Güte der Log-Daten (die hier zufällig erstellt wurden), und der Größe des Prozesses, reichen dem *Alpha-Algorithmus* (bei diesem einfachen Prozess) schon relativ wenige Instanzen, um den ursprünglichen Prozess wieder abzubilden. So kann es sein, dass bei zehn Instanzen schon sämtliche Informationen zur Verfügung stehen, um den Prozess abzubilden. Es kann aber auch sein, dass die benötigten Informationen erst bei 25 oder 50 Instanzen zur Verfügung stehen. Abbildung 3-4 zeigt nun ein Beispiel mit zehn Instanzen, bei dem der Prozess nicht ausreichend erkannt wurde. Vergleicht man den Ergebnisgraphen mit dem ursprünglichen Prozess (oben links im Bild), dann sieht man, dass die Abhängigkeiten zwischen den parallel ausgeführten Aktivitäten nicht richtig erkannt werden.

Die Ursache hierfür ist, dass der *Alpha-Algorithmus* grundsätzlich zwischen den einzelnen Ereignissen einer Aktivität unterscheidet, in unserem Beispiel also zwischen *start* und *complete*. Weil im Log diese beiden Ereignisse nicht immer hintereinander stehen, sondern auch verschränkt mit den Ereignissen anderer Aktivitäten vorkommen können, nimmt der Algorithmus (fälschlicherweise) erst einmal an, dass hier Abhängigkeiten bestehen. Erst wenn genügend Log-Daten zur Verfügung stehen, kann der Algorithmus entscheiden, welche Abhängigkeiten tatsächlich bestehen, und welche nur zufällig im Log enthalten sind.

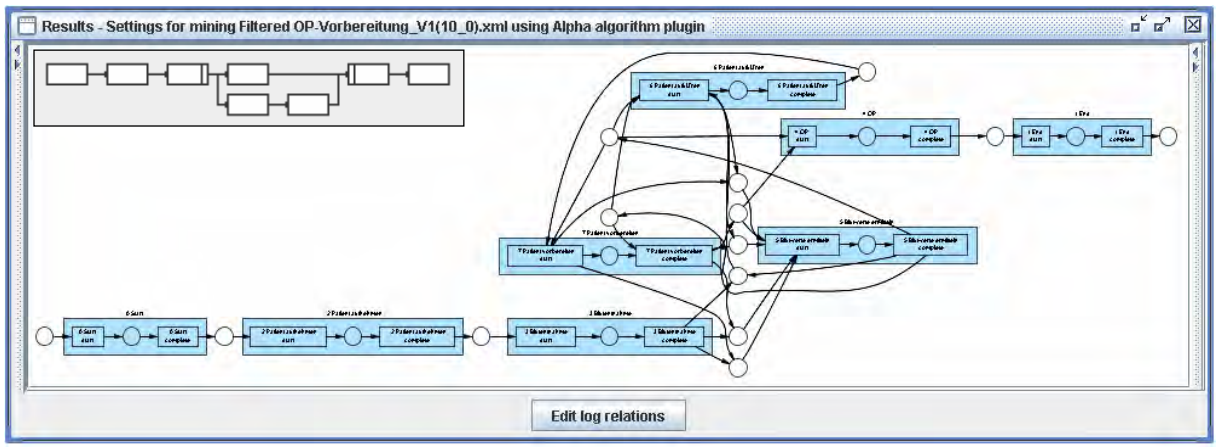


Abbildung 3-4: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V1, 10 Instanzen, keine Noise

Abbildung 3-5 zeigt nun denselben Prozess mit 50 Instanzen und hier wurden alle Abhängigkeiten korrekt erkannt. Der Übersichtlichkeit halber gruppiert *ProM* die Ereignisse einer Aktivität zu einem übergeordneten Knoten, damit man den Ausgangsgraph besser erkennen kann. Dies ist insbesondere wichtig, wenn man Log-Daten verwendet, die nicht nur zwei Ereignisse (*start* und *complete*) aufzeichnen, sondern vielleicht alle zwölf von *ProM* unterstützen Ereignisse protokolliert haben.

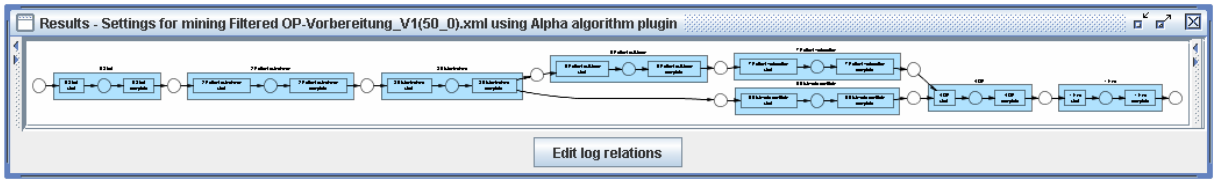


Abbildung 3-5: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V1, 50 Instanzen, keine Noise

Mit dem Button „*Edit log relations*“ können die Abhängigkeiten zwischen den einzelnen Ereignissen verändert werden, um so in kritischen Fällen vielleicht manuell zu einem „guten“ Prozess zu gelangen. Dies ist wiederum von Bedeutung, wenn man annehmen muss, dass die Log-Daten nicht vollständig sind, also (*einfache*) *Noise* beinhalten. Denn wenn nur 5 % der Ereignisse nicht im Log auftauchen (in jeder Log-Datei fehlen ein paar zufällig gewählte Ereignisse), kann der *Alpha-Algorithmus* den Prozess nicht mehr so darstellen, wie er ursprünglich war. Selbst bei 1000 Instanzen kann der Algorithmus nicht korrekt entscheiden, welche Abhängigkeiten zwischen den einzelnen Ereignissen tatsächlich existieren und welche nur fälschlicherweise im Log enthalten sind. Aber die Grundstruktur des Prozesses lässt sich ab einer bestimmten Anzahl Instanzen (im Beispiel ab ca. 50) schon recht gut visuell erkennen. Abbildung 3-6 zeigt denselben Ausgangsprozess, diesmal bei 1000 Instanzen und 5 % *einfacher Noise*. Wie man deutlich erkennen kann, vermutet der Algorithmus Abhängigkeiten zwischen verschiedenen Ereignissen, die so tatsächlich gar nicht existieren. Alle Kanten, die auch im ursprünglichen Prozess (oben rechts im Bild) vorhanden sind, sind zur Verdeutlichung fett markiert. Alle anderen Kanten sind durch *Noise* verursacht.

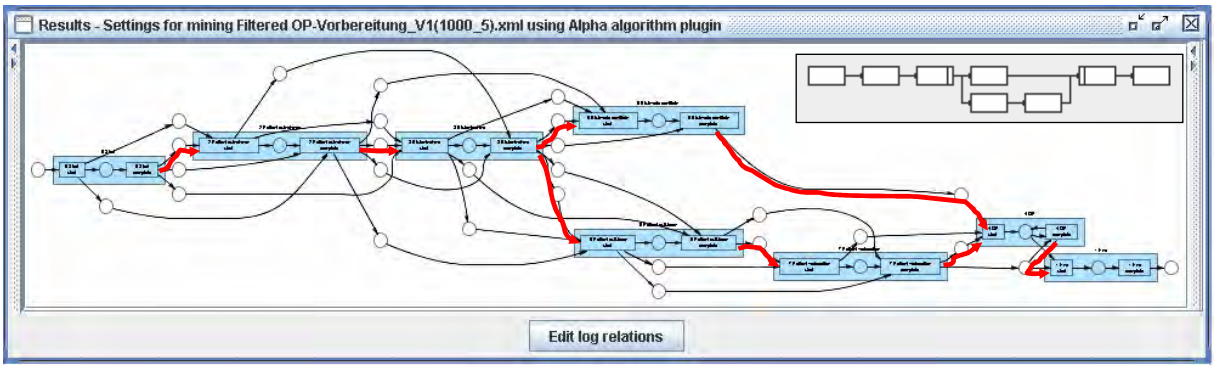


Abbildung 3-6: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V1, 1000 Instanzen, 5 % *einfache Noise*

Enthalten die Log-Daten *einfache Noise*, so gibt es im Ergebnisgraph zusätzliche Kanten (siehe Abbildung 3-6), die der zugrunde liegende Prozess nicht enthält. Die zusätzlichen Kanten zeichnen sich oftmals dadurch aus, dass sie entweder vom *Start*-Ereignis einer Aktivität zu einem Ereignis einer anderen Aktivität gehen, oder von einem Ereignis einer Aktivität zum *End*-Ereignis einer anderen gehen.

Noch schlechter kommt der *Alpha-Algorithmus* mit *Noise* zurecht, die durch geänderte Instanzen entsteht, d.h. wenn unter den verwendeten Instanzen individuell geänderte sind. Solch eine Änderung kann das Einfügen, Löschen oder Verschieben einer Aktivität[10] sein, wobei der *Alpha-Algorithmus* auf die einzelnen Arten von Änderungen verschieden reagiert. Am besten kommt der Algorithmus mit dem Löschen einzelner Aktivitäten zurecht (*einfache Noise* entspricht auch am ehesten dem Löschen von Aktivitäten). Hier erkennt er recht gut, dass eine Aktivität meistens im Log enthalten ist, aber in selteneren Fällen fehlt (siehe Abbildung 3-7; die gelöschte Aktivität ist ‚*Patient aufnehmen*', die zweite Aktivität von links). Im Beispiel war nur bei 50 von 1000 Instanzen eine Änderung erlaubt, tatsächlich wurden nur zwei Instanzen geändert (also die betreffende Aktivität gelöscht).

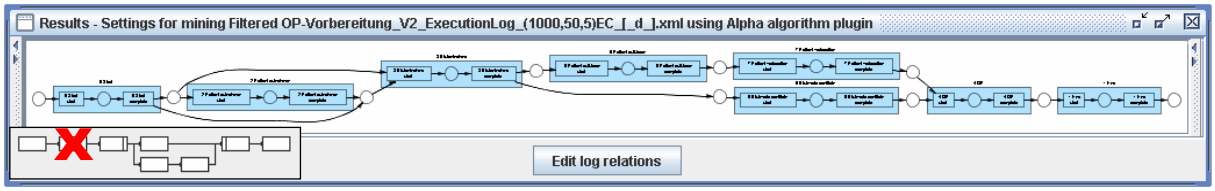


Abbildung 3-7: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 2 individuell geändert (nur Löschen erlaubt)

Beim Verschieben (der Aktivität ‚*Patient aufklären*') kommt der Algorithmus schon mehr durcheinander, was hier auch teilweise daran liegt, dass die Aktivität sowohl nach vorn als auch nach hinten verschoben werden kann (und im Beispiel auch wurde). Durch die unterschiedliche Ausführungsreihenfolge der einzelnen Instanzen nimmt der Algorithmus annä-

[10] Die einzelnen Änderungen, die auf den Prozess ‚*OP-Vorbereitung*' angewendet werden, sind in Kapitel 5.2.1 genau aufgeführt.

hernd eine parallele Ausführung an, die so tatsächlich nicht gegeben ist (siehe Abbildung 3-8).

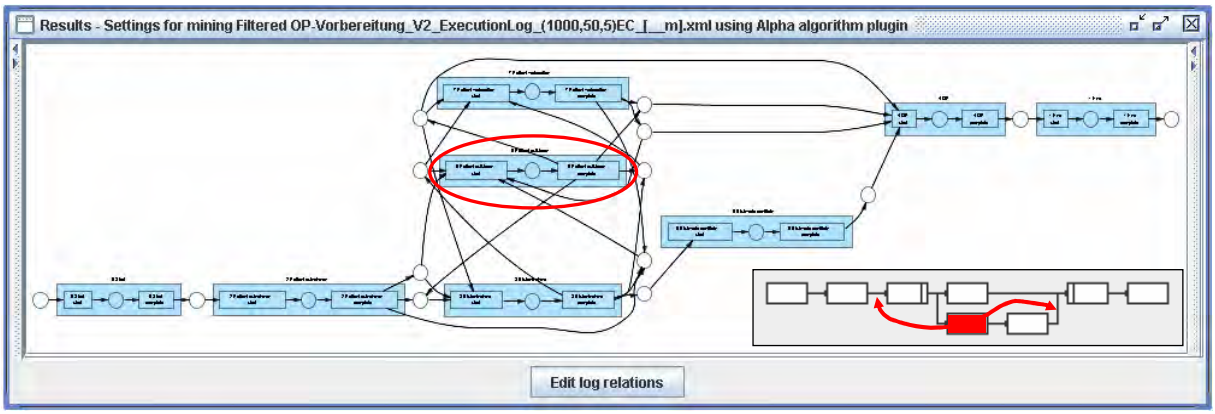


Abbildung 3-8: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 3 individuell geändert (nur Verschieben erlaubt)

Am schlechtesten kommt der Algorithmus mit dem Einfügen von Aktivitäten zurecht, wobei erwähnt werden muss, dass das serielle Einfügen einer einzelnen Aktivität noch recht gut erkannt wird (analog zum Löschen, selten ist die Aktivität vorhanden, meistens nicht). Werden allerdings in den verschiedenen Instanzen unterschiedliche Einfügeoperationen durchgeführt (was auch bei realen Prozessen durchaus der Fall sein kann, wenn nicht sogar die Regel), so ist der *Alpha-Algorithmus* nicht mehr in der Lage, den tatsächlichen Prozess zu erkennen. Im Beispiel war wieder bei 50 von 1000 Instanzen erlaubt, eine Änderung (nur Einfügen) durchzuführen, nur acht Instanzen wurden tatsächlich verändert. Durch das parallele Einfügen der Aktivitäten und die damit zusammenhängende verschränkte Ausführung ist es für den Algorithmus sehr schwer, den tatsächlichen Prozess zu erkennen (siehe Abbildung 3-9). Im Graph markiert sind die vier Aktivitäten, von denen jeweils zwei zusammen eingefügt wurden (entweder ‚*Erweiterte Bluttests*' und ‚*Urintest*' oder ‚*Urintest*' und ‚*Speicheltest*').

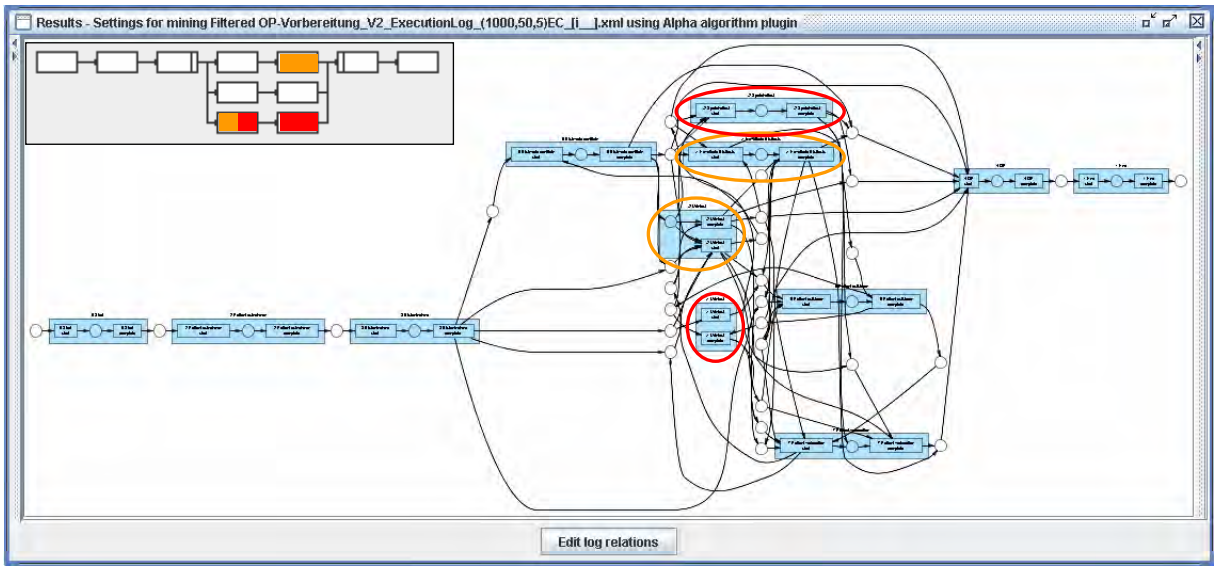


Abbildung 3-9: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 8 individuell geändert (nur Einfügen erlaubt)

Zusammenfassend kann man sagen, dass einzelne, einfache Veränderungen durchaus noch erkannt werden können. Erst die Kombination von mehreren unterschiedlichen Änderungen, also auch verschiedene Einfüge-, Verschiebe- und Löschoperationen, wie sie in realen Prozessen durchaus vorkommen, macht es dem Algorithmus beinahe unmöglich, den zugrunde liegenden Prozess zu erkennen. Abbildung 3-10 zeigt nun den Beispielprozess mit 1000 Instanzen, wovon bei 50 Instanzen eine individuelle Änderung erlaubt war. Tatsächlich wurden 18 Instanzen geändert, wobei alle möglichen Änderungen (Einfügen, Verschieben, Löschen und Kombinationen davon) auch mindestens einmal durchgeführt wurden. Durch die Kombination dieser über wenige Instanzen verteilten Änderungen ergibt sich für den *Alpha-Algorithmus* ein Datensatz, aus dem er den tatsächlichen Prozess nicht mehr herleiten kann. Im Gegensatz zum Beispiel mit *einfacher Noise* (siehe Abbildung 3-6) lässt sich der zugrunde liegende Prozess auch graphisch nur noch schwer erkennen.

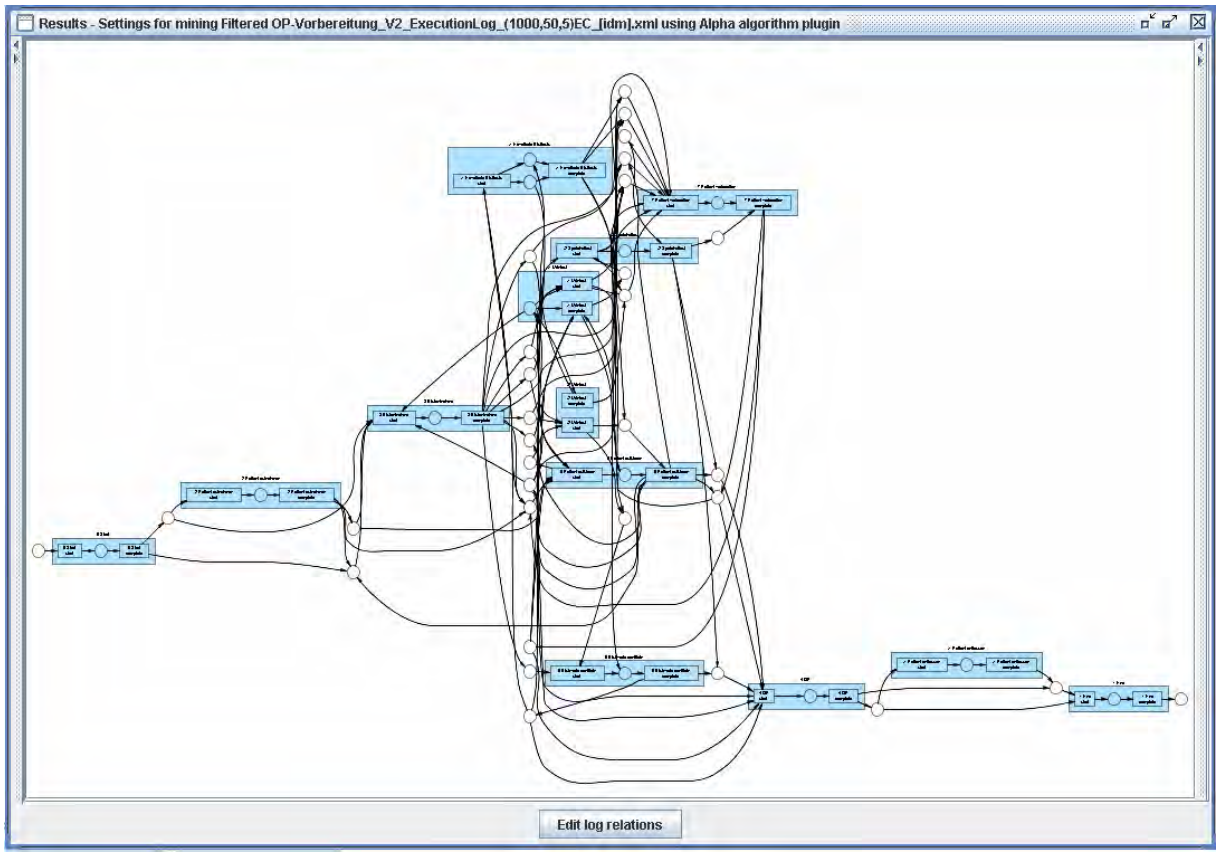


Abbildung 3-10: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 18 individuell geändert (alles erlaubt, entspricht ca. 2 % *Noise*)

3.3.2 Multi-Phase Mining

Die meisten *Mining*-Ansätze versuchen, aus den erfassten Daten ein komplettes Prozessmodell zu generieren. Da ein komplettes Modell aber nicht immer das oberste Ziel ist, wird solch ein Modell nicht immer benötigt. Stattdessen kann eine gute Visualisierung der individuellen Prozessinstanzen ausreichen. Aus diesen individuellen Instanzen kann dann ein Gesamtmodell erstellt werden, wenn es benötigt wird. *Multi-Phase Mining* [DoAa04] ist ein Ansatz, der für jede Prozessinstanz einen Instanzgraphen erstellt, basierend auf der Information des kompletten Datensatzes. Solch ein Instanzgraph kann entweder als Petrinetz oder als Ereignis-Prozess-Kette (EPK) angezeigt werden.

Die Erstellung eines Graphen läuft folgendermaßen ab: Zuerst werden anhand der Log-Daten die kausalen Beziehungen zwischen den einzelnen Prozessschritten ermittelt. Zwei Schritte A und B haben eine kausale Beziehung ($A \rightarrow_W B$), wenn in irgendeiner Instanz B direkt auf A folgt und A nie direkt auf B folgt. Eine Ausnahme hiervon bilden zweielementige Schleifen (ABA bzw. BAB). Wenn zusätzlich gilt, dass A nie direkt auf A und B nie direkt auf B folgt, dann haben die Schritte A und B auch eine kausale Beziehung, obwohl B auf A und A auf B folgen. Danach werden die nächsten kausalen Nachbarn ermittelt, also der direkte Vorgänger und der direkte Nachfolger eines Prozessschrittes, um somit die einzelnen Schritte zu ordnen. Gibt es keinen direkten kausalen Vorgänger oder Nachfolger, also mehr als einen Vorgänger oder Nachfolger, dann ist der Schritt entsprechend parallel zu (all) seinen Vorgängern oder

Nachfolgern angeordnet.
Mit den ermittelten Daten wird dann ein Instanzgraph generiert, der den Kontrollfluss einer Prozessinstanz, also die kausalen Beziehungen und die Parallelität (soweit vorhanden) zwischen den einzelnen Prozessschritten zeigt. ODER-Verzweigungen und Schleifen kommen in einem Instanzgraphen nicht vor, weil er typischerweise einen Ausführungspfad eines Prozessmodells beschreibt und die Entscheidungen, die den auszuführenden Pfad betreffen, bereits zur Laufzeit getroffen wurden. Dementsprechend können dieselben getroffenen Entscheidungen bei verschiedenen Instanzen zum gleichen Instanzgraphen führen, während parallele Schritte in jeder Reihenfolge auftreten können, ohne den Instanzgraphen zu beeinflussen. Bei der Aggregation mehrerer Instanzgraphen zu einem gemeinsamen Graphen werden ODER-Verzweigungen wieder angezeigt, wenn die unterschiedlichen Pfade in den unterschiedlichen Instanzen ausgewählt wurden.

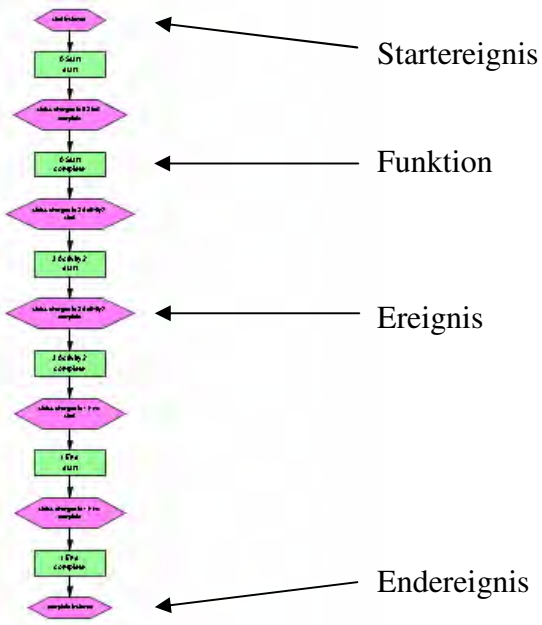
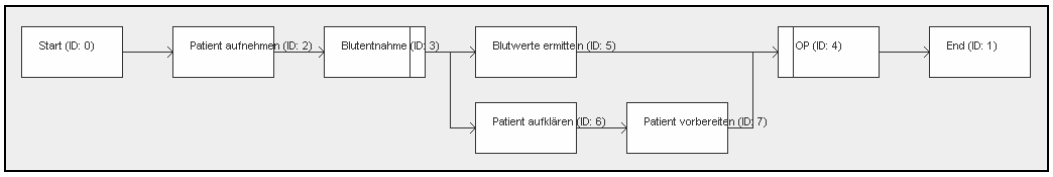


Abbildung 3-11: Beispiel für eine Instanz-EPK

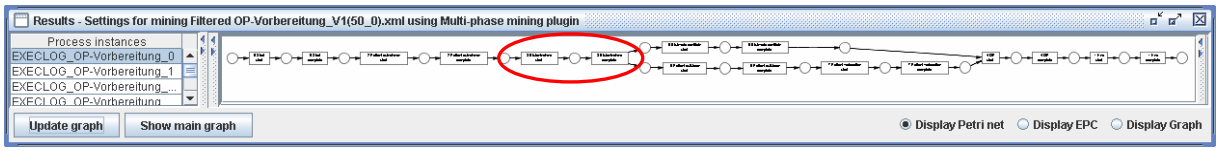
Die Instanzgraphen werden schließlich auf eine spezielle Form von Ereignis-Prozess-Ketten, so genannte Instanz-EPK's (siehe Abbildung 3-11) abgebildet. Instanz-EPK's haben genau ein Startereignis und genau ein Endereignis, vergleichbar mit einem *ADEPT*-Graph, wo es auch jeweils einen Start- und Endknoten gibt. Die Funktionen in der Ereignis-Prozess-Kette entsprechen den Einträgen in den Log-Daten, während die Ereignisse nicht aus dem Log entnommen werden können. Hier wurde festgelegt, dass jedes Ereignis eine Funktion startet, d.h. jeder Funktion geht ein Ereignis voraus, wobei für Start- und Endereignis spezielle Ausnahmen gelten. Wie üblich in einer EPK wechseln sich Ereignisse und Funktionen ab. Die er-

zeugten Instanz-EPK's können nun z.B. in ARIS PPM[11] eingelesen werden und dort zur Visualisierung, Aggregation oder Analyse verwendet werden.

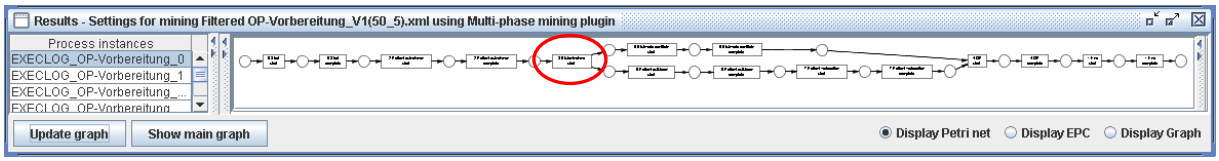
Ein Vorteil von *Multi-Phase-Mining* gegenüber dem *Alpha-Algorithmus* ist die Tatsache, dass die Log-Daten nicht in irgendeiner Weise vollständig sein müssen, um einen Instanzgraphen zu erzeugen. Es gibt also von vornherein weniger Voraussetzungen, die für das *Mining* erfüllt sein müssen. Solange kausale Beziehungen zwischen Log-Einträgen vorhanden sind, können Instanzgraphen, und damit Instanz-EPK's erzeugt werden. Das lässt sich auch im Zusammenhang mit *Noise* beobachten. Bei der Einzelbetrachtung einer Prozessinstanz mit oder ohne *einfache Noise* ist der einzige Unterschied, dass der ein oder andere Knoten im Graph fehlt, der angezeigte Graph aber (meist) fehlerfrei ist und den Daten im zugrunde liegenden Log entspricht. Abbildung 3-12(a) zeigt den ursprünglichen Prozess, so wie er im *Demonstrator* angezeigt wird. Abbildung 3-12(b) zeigt eine Prozessinstanz ohne *einfache Noise*, Abbildung 3-12(c) zeigt eine Prozessinstanz mit 5 % *einfacher Noise*.



(a) ursprünglicher Prozess: OP-Vorbereitung, V1



(b) Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 50 Instanzen (davon Instanz 0 ausgewählt), keine Noise



(c) Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 50 Instanzen (davon Instanz 0 ausgewählt), 5 % *einfache Noise*

Abbildung 3-12: Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 50 Instanzen (davon Instanz 0 ausgewählt), mit und ohne *Noise*

Die beiden Graphen sehen auf den ersten Blick beinahe identisch aus, erst bei näherem Betrachten erkennt man, dass bei dem Graph mit *Noise* das *complete*-Ereignis von ‚Blutentnahme' fehlt. Dieses Ereignis entspricht in Abbildung 3-12(b) dem sechsten Knoten, also dem Splitknoten vor der Verzweigung, während sich der Prozess in Abbildung 3-12(c) bereits nach dem fünften Knoten verzweigt. Der Übersichtlichkeit halber sind beide Prozesse als Petrinetz

[11] Der ARIS Process Performance Manager (ARIS PPM) ist ein patentiertes Werkzeug zur Analyse, Bewertung und zum Monitoring von Unternehmensprozessen. Näheres dazu kann unter ‚http://www.ids-scheer.com/germany/products/aris_controlling_platform/49532' nachgelesen werden.

abgebildet, da eine Ereignis-Prozess-Kette in *ProM* von oben nach unten dargestellt wird (s.u.).

Bei der Betrachtung einzelner Instanzen kommt das *Multi-Phase Mining* also beinahe problemlos mit *Noise* zurecht. Und wenn nur die einzelnen Instanzen betrachtet werden sollen, spielt auch die Menge der zugrunde liegenden Log-Daten eine untergeordnete Rolle. Da mit diesem Algorithmus aber auch mehrere Instanzen ausgewählt werden können, die als Graph angezeigt werden sollen, zeigt sich auch hier, dass es besser ist, wenn genügend Daten zur Verfügung stehen. Im Beispiel sind erst ab ca. 50 Instanzen genügend Daten vorhanden, um den ursprünglichen Prozess wieder herstellen zu können. Dies hängt aber auch hier stark von der Güte der Log-Daten ab, denn bei diesem einfachen Prozess können auch 25 „gut gewählte“ Instanzen alle wichtigen Informationen liefern. Da die Log-Daten aber zufällig generiert wurden, kann und soll nicht sichergestellt werden, dass die Instanzen „gut gewählt“ sind. Abbildung 3-13 zeigt einen Prozessgraph als Petrinetz, von den 25 zur Verfügung stehenden Instanzen werden nur die ausgewählten fünf für die Erstellung des Graphen verwendet.

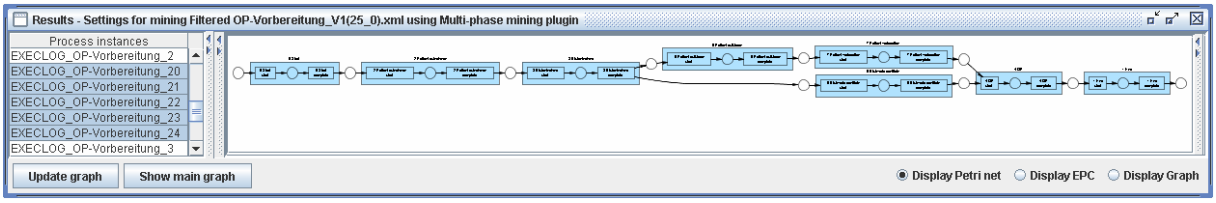


Abbildung 3-13: Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 25 Instanzen (davon Instanzen 20-24 ausgewählt), keine Noise

Werden z.B. fünf andere Instanzen gewählt, und ist unter diesen Instanzen auch ein „Ausreißer“, was die Ausführungsreihenfolge betrifft, können auch fehlerhafte Graphen entstehen. Abbildung 3-14 zeigt solch einen Graphen. Auch wenn alle Instanzen ausgewählt werden, ergibt sich ein Graph wie in Abbildung 3-14, da auch hier die „fehlerhafte“ Instanz mit berücksichtigt wird.

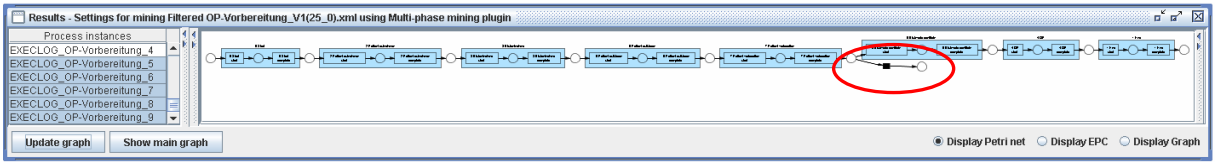


Abbildung 3-14: Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 25 Instanzen (davon Instanzen 5-9 ausgewählt), keine Noise

Wird bei den Optionen ausgewählt, dass der Graph als EPK (*EPC*) angezeigt werden soll, ergibt sich ein Graph, wie er ausschnittsweise in Abbildung 3-15 gezeigt wird. Es wurden wieder dieselben fünf Instanzen wie in Abbildung 3-13 für die Grapherstellung verwendet.

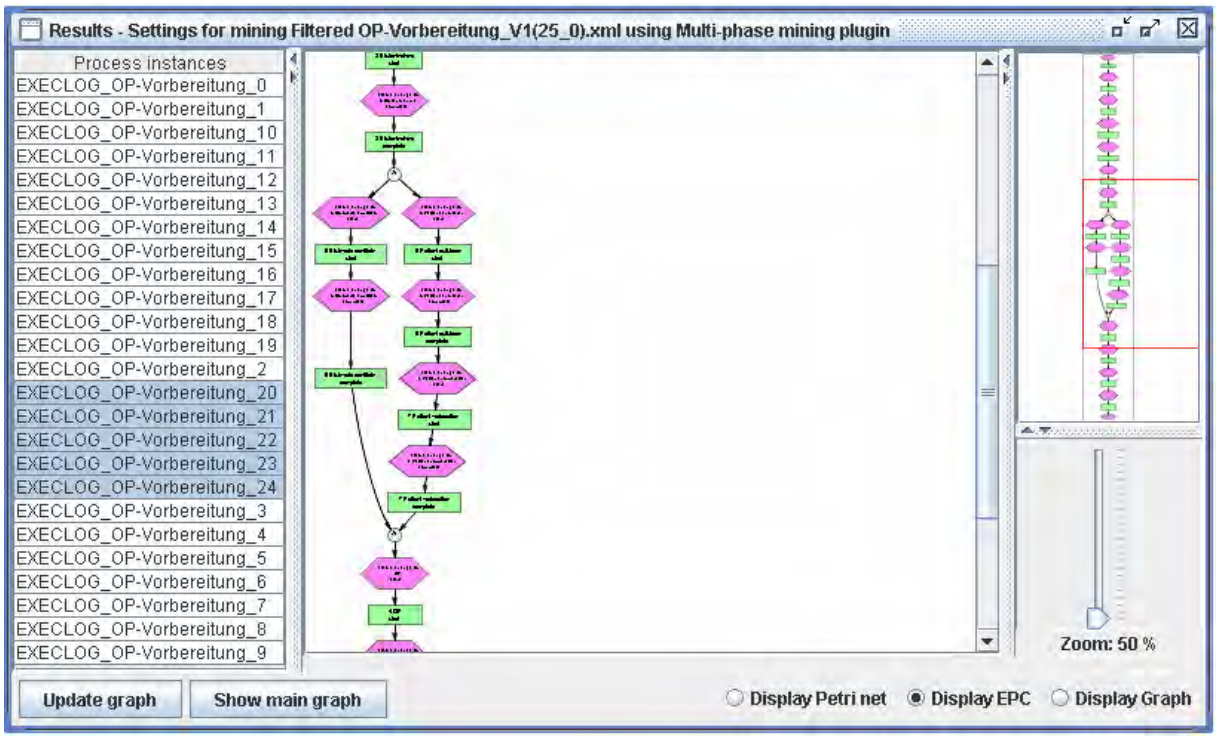


Abbildung 3-15: Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 25 Instanzen (davon Instanzen 20-24 ausgewählt), keine Noise – Ausschnitt aus dem Graphen

Dass auch das *Multi-Phase Mining* anfällig für *Noise* ist, zeigt Abbildung 3-16. Auch hier sind bei 1000 Instanzen und 5 % *einfacher Noise* nicht genügend Informationen enthalten, um den ursprünglichen Graphen wiederherstellen zu können. Und im Gegensatz zum *Alpha-Algorithmus* lässt sich die Grundstruktur des Graphen selbst bei 1000 Instanzen visuell nicht erkennen.

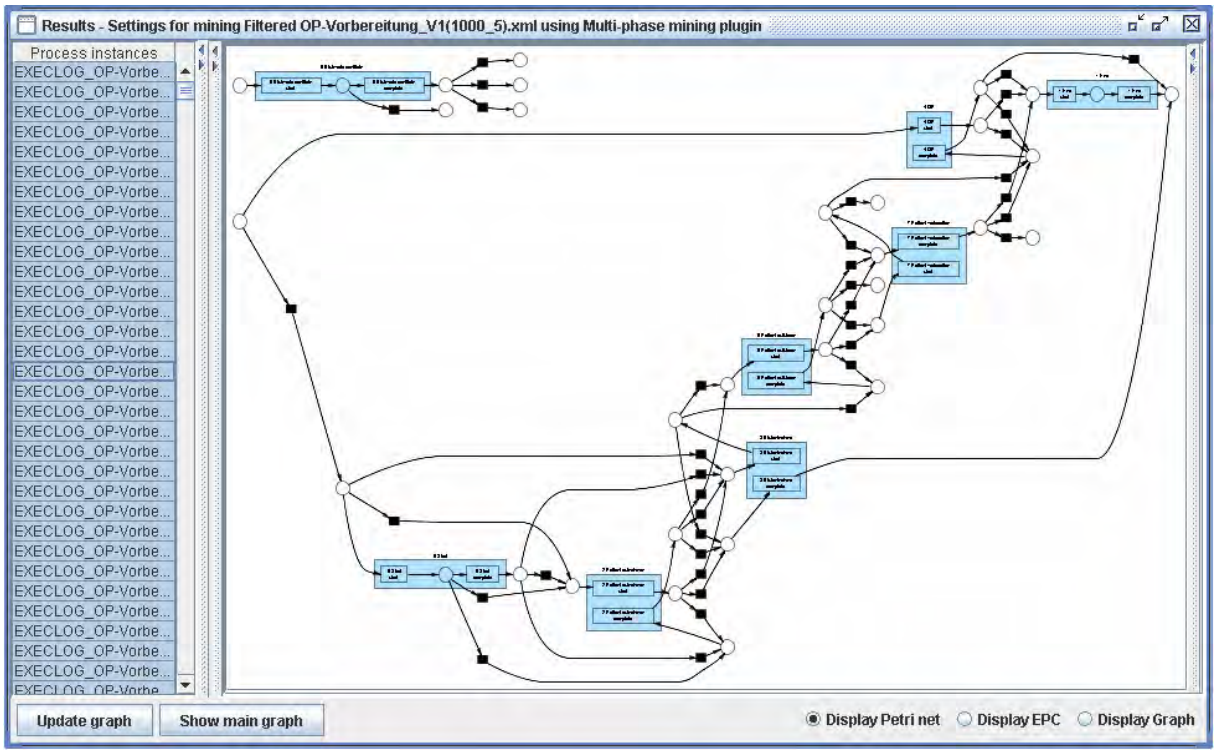


Abbildung 3-16: Beispiel Multi-Phase Mining: OP-Vorbereitung, V1, 1000 Instanzen (alle ausgewählt), 5 % *einfache Noise*

Ähnlich wie der *Alpha-Algorithmus* verhält sich *Multi-Phase Mining* im Zusammenhang mit *Noise* aus geänderten Instanzen. Das Löschen einer einzelnen Aktivität bei zwei von 1000 Instanzen wird vom *Multi-Phase Mining* so erkannt, als ob es eine seltene Option ist, die Aktivität ausfallen zu lassen (siehe Abbildung 3-17, vgl. Abbildung 3-7).

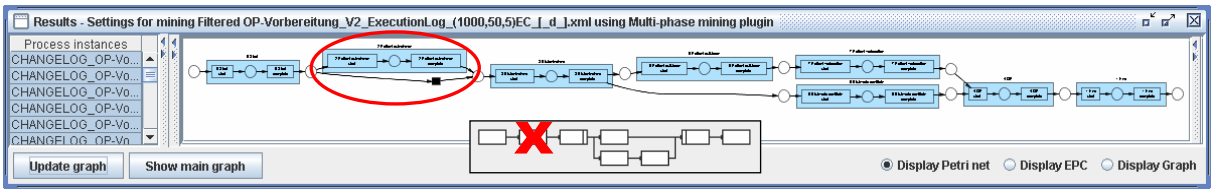


Abbildung 3-17: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (alle ausgewählt), davon 2 individuell geändert (nur Löschen erlaubt)

Werden die Änderungen komplexer, so wird das Ergebnis auch immer unklarer. Abbildung 3-18 zeigt den Beispielprozess, wobei von den 1000 Instanzen nur drei tatsächlich geändert wurden. Allerdings wurde die Aktivität ‚*Patient aufklären*' einmal nach vorn und zweimal nach hinten verschoben, so dass sich für den Algorithmus kein klares Muster mehr erkennen lässt. In der Realität sind solche Änderungen trotzdem keine Seltenheit, gerade eine Aktivität ‚*Patient aufklären*' kann in einem Krankenhausprozess an mehreren Stellen angeordnet werden, je nachdem wann Arzt und Patient gemeinsam Zeit für ein Gespräch haben.

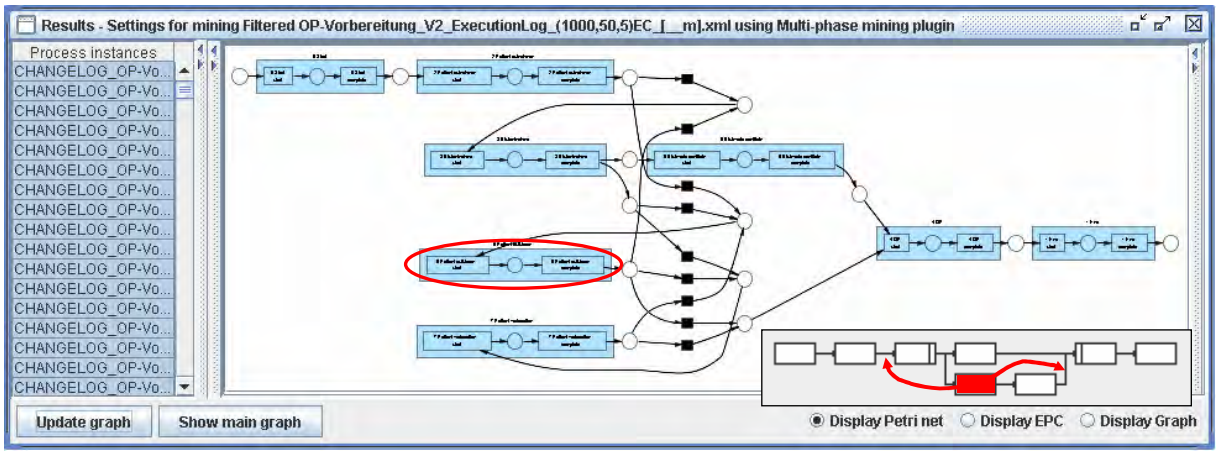


Abbildung 3-18: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (alle ausgewählt), davon 3 individuell geändert (nur Verschieben erlaubt)

Am schlechtesten von den allgemeinen Änderungsoperationen kommt auch der *Multi-Phase Mining Algorithmus* mit dem Einfügen verschiedener Aktivitäten zurecht. Zum einen liegt das sicher daran, dass von den 1000 Instanzen diesmal acht geändert wurden, zum anderen aber auch daran, dass nicht nur eine einzelne Aktivität eingefügt wurde, sondern zwei, und diese teilweise auch noch parallel zu den bestehenden und unabhängig voneinander, so dass für den Algorithmus eine Beziehung nicht ohne weiteres erkennbar ist. Abbildung 3-19 zeigt das Ergebnis des *Minings* der Prozessinstanzen, bei denen Einfügen erlaubt war. Im Graph markiert sind die vier Aktivitäten, von denen jeweils zwei zusammen eingefügt wurden (entweder ‚*Erweiterte Bluttests*' und ‚*Urintest*' oder ‚*Urintest*' und ‚*Speicheltest*').

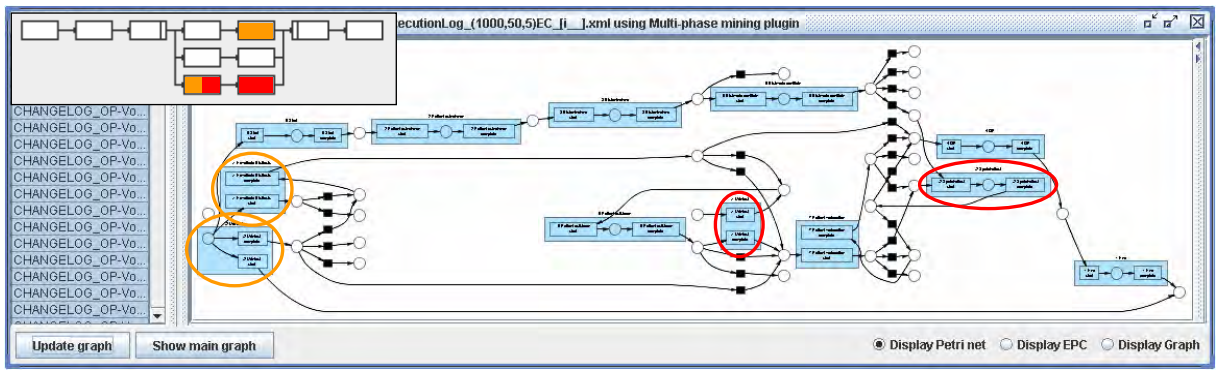


Abbildung 3-19: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (alle ausgewählt), davon 8 individuell geändert (nur Einfügen erlaubt)

Auch beim *Multi-Phase Mining* lässt sich wieder zusammenfassend sagen, dass einzelne, einfache Änderungen beim *Mining* erkannt werden können. Aber auch hier ergibt sich durch die Kombination der einzelnen Änderungsoperationen ein Ergebnis, in dem der ursprüngliche Prozess nur noch sehr schwer, wenn überhaupt noch erkannt werden kann (siehe Abbildung 3-20). Und das, obwohl von den verwendeten 1000 Instanzen nur 18 Instanzen tatsächlich verändert wurden, was einem Prozentsatz von unter 2 % *Noise* entspricht. Und mit ‚*so wenig*' *Noise* muss bei realen Prozessaufzeichnungen durchaus gerechnet werden.

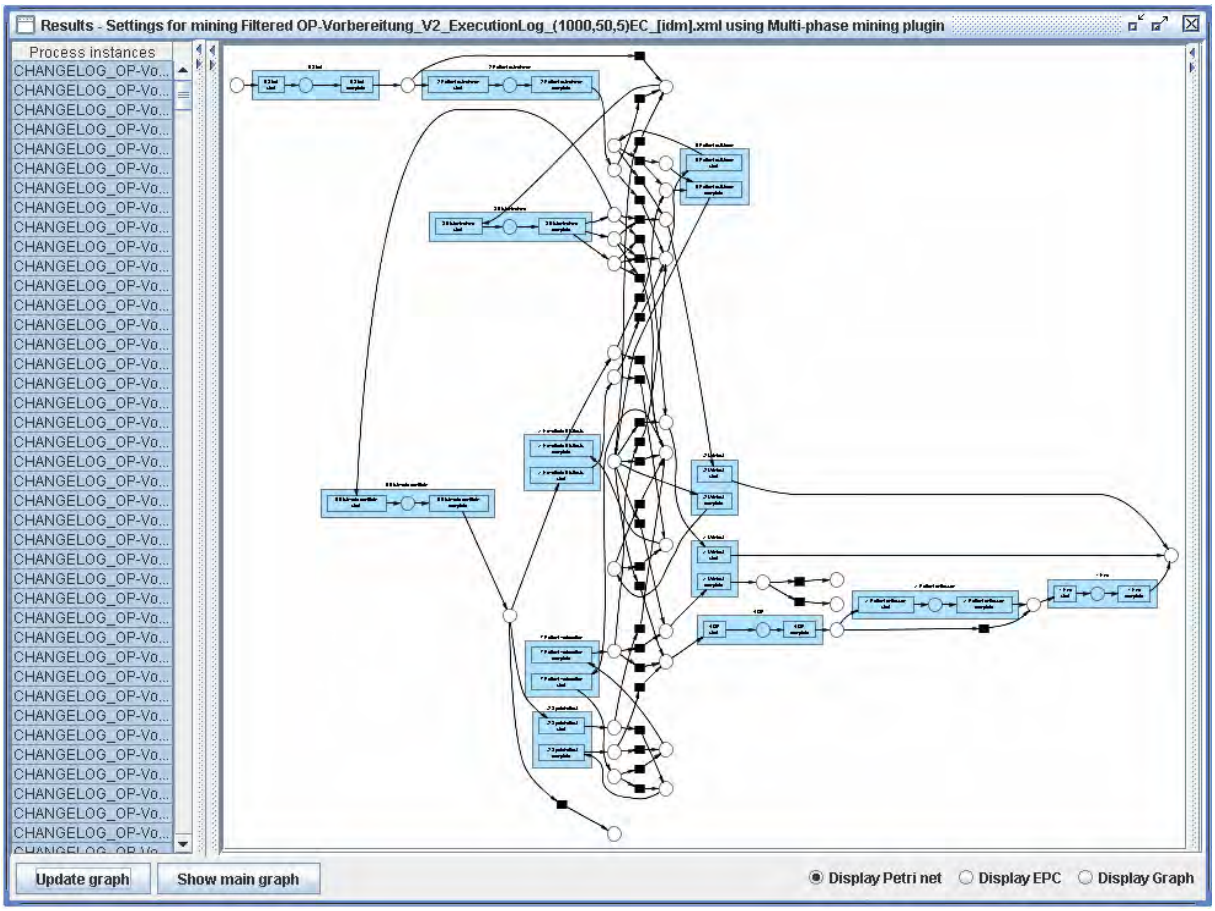


Abbildung 3-20: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (alle ausgewählt), davon 18 individuell geändert (alles erlaubt, entspricht ca. 2 % *Noise*)

Etwas anders sieht die Situation aus, wenn nur eine einzelne Prozessinstanz betrachtet wird, was ja der Grundgedanke hinter dem *Multi-Phase Mining* war. Hier spielt der Algorithmus seine Stärken aus und liefert einen Prozess, der rein äußerlich vollkommen korrekt und ohne viele Verzweigungen ist. Abbildung 3-21 zeigt eine Prozessinstanz, bei der die Aktivität ‚*Patient aufnehmen*' gelöscht wurde, der Rest des Prozesses wird so dargestellt, wie er auch ohne *Noise* dargestellt würde.

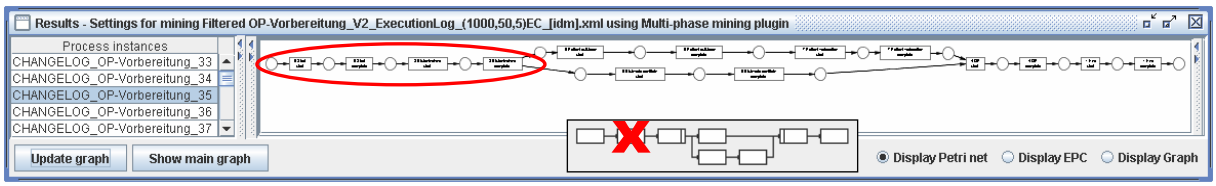


Abbildung 3-21: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (individuell geänderte Instanz 35 ausgewählt)

Abbildung 3-22 zeigt eine Prozessinstanz (ausschnittsweise), in der zwei zusätzliche Aktivitäten parallel zu den vorhanden Schritten eingefügt wurden (entspricht dem Einfügen eines zusätzlichen Pfades innerhalb der UND-Verzweigung). Hier zeigt sich, dass diese Parallelität

nicht komplett richtig erkannt wurde, da ja für die Berechnung der kausalen Beziehungen zwischen den Aktivitäten alle (1000) Instanzen verwendet werden, während die Änderungen nur in dieser einen Instanz vorkommen und deshalb die zugrunde liegenden Daten für eine komplett richtige Wiederherstellung des Graphen nicht ausreichend sind. Nichtsdestotrotz lässt sich sagen, dass *Multi-Phase Mining* besser mit *Noise* zurecht kommt als der *Alpha-Algorithmus*, so lange nur eine einzelne Instanz betrachtet wird. Das liegt daran, dass hier die fehlerhaften Instanzen besser herausgefiltert werden, d.h. von den anderen Instanzen werden nur die grundsätzlichen Beziehungen der Schritte untereinander verwendet. Werden alle Instanzen zu einem Graphen zusammengefasst, dann zeigt sich der *Alpha-Algorithmus* (der ja immer alle Instanzen zusammenfasst) etwas weniger anfällig gegenüber *Noise*, aber nur so wenig, dass es bei realen Prozessen nicht weiter ins Gewicht fallen dürfte.

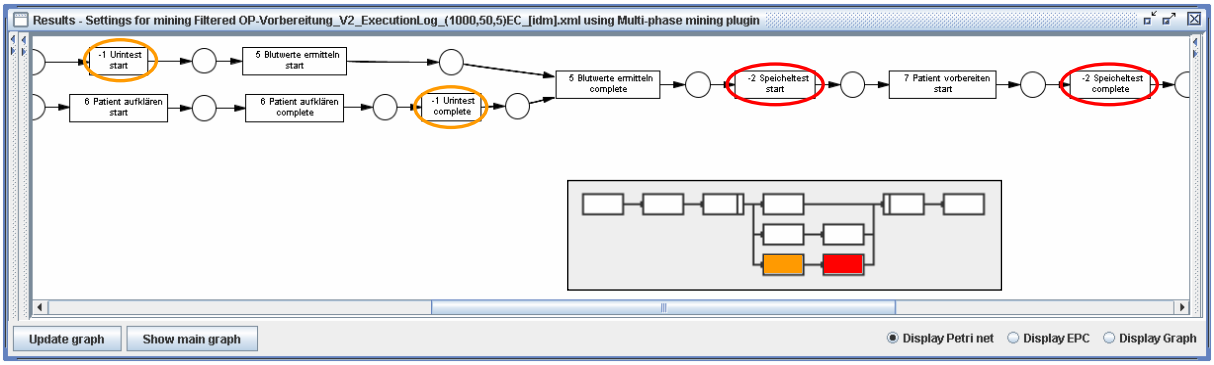


Abbildung 3-22: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 1000 Instanzen (individuell geänderte Instanz 20 ausgewählt) – Ausschnitt aus dem Graphen

3.3.3 Tsinghua-Alpha-Algorithmus

Der *Tsinghua-Alpha-Algorithmus* [WWAW04] ist eine Variante des *Alpha-Algorithmus* (siehe Kapitel 3.3.1), der die beiden Ereignistypen *start* und *complete* verwendet, um damit explizit parallele Abhängigkeiten zwischen einzelnen Prozessschritten zu erkennen. Andere Algorithmen benutzen die Ereignistypen nicht, zumindest nicht für die explizite Erkennung von Parallelität. Sie gehen stattdessen entweder davon aus, dass die Prozessschritte atomar sind bzw. betrachten die einzelnen Ereignisse eines Schrittes als atomare Schritte (z.B. *Alpha-Algorithmus*, *Multi-Phase Mining*) oder sie benutzen nur das *complete*-Ereignis eines Schrittes für das Gewinnen des Prozesses. Die beim *Tsinghua-Alpha-Algorithmus* verwendeten kausalen Beziehungen und die Voraussetzungen für die Vollständigkeit der Log-Daten unterscheiden sich deutlich von denen des *Alpha-Algorithmus*. So ist es dem Algorithmus z.B. möglich, kurze Schleifen zu erkennen, die der *Alpha-Algorithmus* in seiner ursprünglichen Form [AWM04] nicht erkennen konnte. Der grundsätzliche Ablauf des Algorithmus und ein Ergebnis in Form eines *Workflow-Netzes* zeigen aber die Nähe zum *Alpha-Algorithmus*.

Weil der *Tsinghua-Alpha-Algorithmus* Start- und End-Ereignis eines Schrittes getrennt betrachtet, müssen Paare von Ereignissen für die Anordnungsbeziehungen betrachtet werden, also ein Start-Ereignis und das dazugehörige End-Ereignis. Der Algorithmus basiert auf zwei grundsätzlichen Anordnungsbeziehungen, die aus den Log-Daten abgeleitet werden können, Aufeinanderfolgen ($>_W$) und Überschneidungen (x_W).

- $A >_W B$ trifft zu, d.h. A und B sind in einer Abfolge, falls es mindestens eine Ereignisfolge im Log gibt, in der B direkt auf A folgt, wenn es also in mindestens einer Instanz keine weiteren *kompletten* Schritte zwischen dem End-Ereignis von A und dem Start-Ereignis von B gibt.
- $A \; x_W \; B$ trifft zu, d.h. A und B überschneiden sich, falls sich die Vorkommen von A und B in mindestens einer Ereignisfolge im Log überlappen, wenn also z.B. das Start-Ereignis von B vor dem End-Ereignis von A auftritt. Diese Beziehung ist symmetrisch, d.h. $A \; x_W \; B$ trifft nur dann zu, wenn auch $B \; x_W \; A$ zutrifft.

Aus diesen zwei grundsätzlichen Beziehungen können dann die vier für ein Prozessmodell typischen Beziehungen Sequenz, Parallelität, Verzweigung und Iteration bzw. Kombinationen daraus abgeleitet werden.

- $A \rightarrow_W B$ trifft nur zu, wenn $A >_W B$ und $\neg(A \; x_W \; B)$ zutreffen. ($\hat{=}$ Sequenz)
- $A \; ||_W \; B$ trifft nur zu, wenn $A \; x_W \; B$ zutrifft. ($\hat{=}$ Parallelität)
- $A \; \#_W \; B$ trifft nur zu, wenn $\neg(A >_W B)$ und $\neg(A \; x_W \; B)$ zutreffen. ($\hat{=}$ Verzweigung)
- $A \; \not\parallel_W \; B$ trifft nur zu, wenn $\neg(A \; x_W \; B)$ zutrifft.

Um die Beziehungen richtig zu erkennen benötigt der Algorithmus vollständige und konsistente Log-Daten. Es müssen alle möglichen Schritte (in irgendeiner Instanz) im Log auftauchen, um die Grundbeziehungen $>_W$ und x_W korrekt bestimmen zu können (Vollständigkeit). Darüber hinaus muss für jedes Start-Ereignis im Log ein dazugehöriges End-Ereignis vorhanden sein (Konsistenz). Sind die zugrunde liegenden Daten fehlerhaft, so wird der Algorithmus zwar einen Prozess zurückliefern, aber es ist dann sehr wahrscheinlich, dass er vom ursprünglichen Prozess abweicht (siehe unten, *Noise*).

Der Algorithmus läuft folgendermaßen ab:

1. Die Menge der Transitionen ($\hat{=}$ den kompletten Schritten, also Start- und End-Ereignis zusammen) wird aus dem Log extrahiert.
2. Danach werden die Startknoten und
3. die Endknoten bestimmt.
4. Jetzt findet der *Tsinghua-Alpha-Algorithmus* heraus, welche Transitionen voneinander abhängen, d.h. welche (Paare von) Transitionen in einer Sequenz mit anderen (Paaren von) Transitionen sind. Alle gefundenen Abhängigkeiten sind mögliche Stellen im Graph.
5. Aus den in Schritt 4 gefundenen Abhängigkeiten werden nun die größten Elemente ermittelt, so dass alle Stellen wegfallen, die in anderen Beziehungen enthalten sind. Dadurch erhält man die exakt benötigte Menge an Stellen für den Prozess.
6. Quelle und Senke und die in Schritt 5 ermittelten Stellen werden nun erstellt und
7. mit den entsprechenden Transitionen verbunden.
8. Schließlich wird das entdeckte *Workflow-Netz* als Menge von Transitionen, Stellen und Flussbeziehungen zurückgegeben.

Die einzelnen Ereignisse einer Aktivität werden nur für den Algorithmus separat betrachtet, graphisch werden sie im Gegensatz zum *Alpha-Algorithmus* (u.a.) zu einer einzigen Transition zusammengefasst. Dies dient der Übersichtlichkeit des Graphen und ist insbesondere dann interessant, wenn von den zwölf zur Verfügung stehenden Ereignissen mehr als zwei genutzt werden. Das Ergebnis lässt sich sehr gut mit dem Ausgangsprozess vergleichen, da auch da die einzelnen Ereignisse einer Aktivität graphisch selten unterschieden werden. Durch die grundsätzlich anderen Anordnungsbeziehungen kommt der *Tsinghua-Alpha-Algorithmus* tendenziell auch mit etwas weniger Log-Daten aus, so reichen im Beispiel schon zehn bis 25 Instanzen, um den korrekten Graphen wieder herzustellen. Abbildung 3-23 zeigt den Graphen, der bei zehn Instanzen erstellt wurde, wobei anzumerken ist, dass es sich hierbei um dieselben zehn Instanzen handelt, die für *Alpha-Algorithmus* und *Multi-Phase Mining* nicht ausreichend waren.

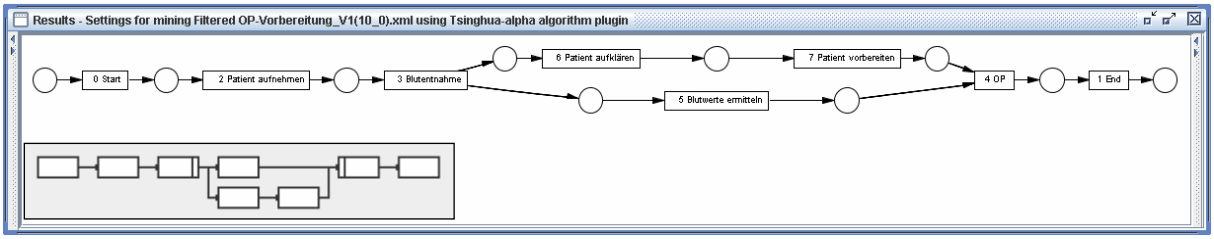


Abbildung 3-23: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V1, 10 Instanzen, keine Noise

Mehr noch als bei den anderen Algorithmen benötigt der *Tsinghua-Alpha-Algorithmus* korrekte und vollständige Log-Daten, enthalten die Logs auch nur geringfügige Fehler bzw. fehlen nur wenige Daten, gibt der *Tsinghua-Alpha-Algorithmus* nur noch Teilgraphen aus, die keinerlei Rückschlüsse auf den ursprünglichen Prozess mehr zulassen. Dies liegt daran, dass die *einfache Noise* eine Schwachstelle des Algorithmus trifft, weil dieser von konsistenten Daten ausgeht, d.h. dass zu jedem Start-Ereignis auch das dazu gehörende End-Ereignis im Log ist. Genau das ist bei *einfacher Noise* nicht gegeben, da hier einzelne Ereignisse aus dem Log entfernt werden und so grundsätzlich inkonsistente Daten übrig bleiben. Abbildung 3-24 zeigt den Graphen, der bei 1000 Instanzen und 5 % *einfacher Noise* ausgegeben wird.

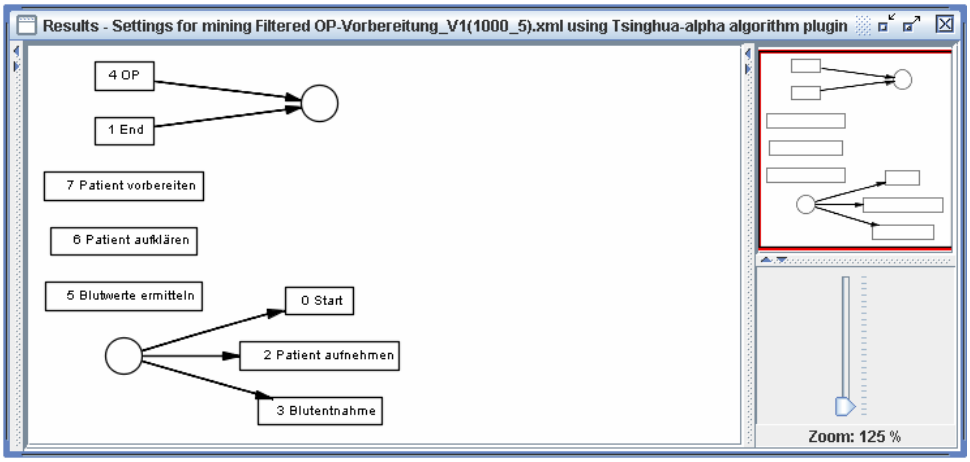


Abbildung 3-24: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V1, 1000 Instanzen, 5 % *einfache Noise*

Etwas besser sieht es aus, wenn die *Noise* aus geänderten Instanzen kommt, da hier die Log-Daten konsistenter sind und somit für den Algorithmus besser zu verarbeiten sind. Ähnlich wie der *Alpha-Algorithmus* und *Multi-Phase Mining* erkennt der *Tsinghua-Alpha-Algorithmus* den zugrunde liegenden Prozess relativ gut, wenn nur wenige Änderungen vorgenommen wurden. So ist es für den Algorithmus kein Problem zu erkennen, dass die zweite Aktivität ‚*Patient aufnehmen*' in einigen Instanzen (im Beispiel zwei Instanzen) entfallen kann. Im Ergebnisprozess kann diese Aktivität nun optional ausgeführt werden oder eben nicht (siehe Abbildung 3-25).

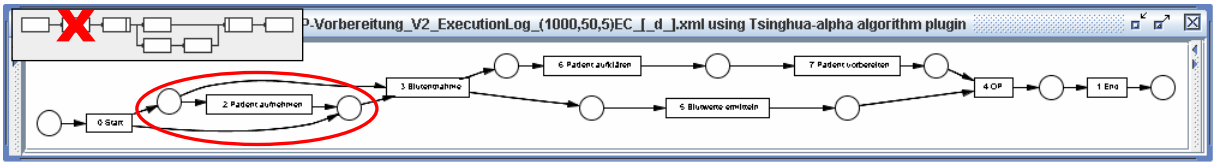


Abbildung 3-25: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 2 individuell geändert (nur Löschen erlaubt)

Werden die Änderungen wieder komplexer, wird auch der Ergebnis-Prozess wieder ungenauer. Abbildung 3-26 zeigt den Ergebnis-Graphen, wenn bei drei Instanzen die Aktivität ‚*Patient aufklären*' verschoben wurde, einmal nach vorn und zweimal nach hinten. Durch die Änderungen an den drei Instanzen gibt es drei komplett verschiedene Ausführungsreihenfolgen, die es dem Algorithmus schwer machen, alle in einen Prozess zu vereinen. Man kann aber immer noch eine gewisse Grundstruktur erkennen, was ja bei *einfacher Noise* (vgl. Abbildung 3-24) nicht möglich ist.

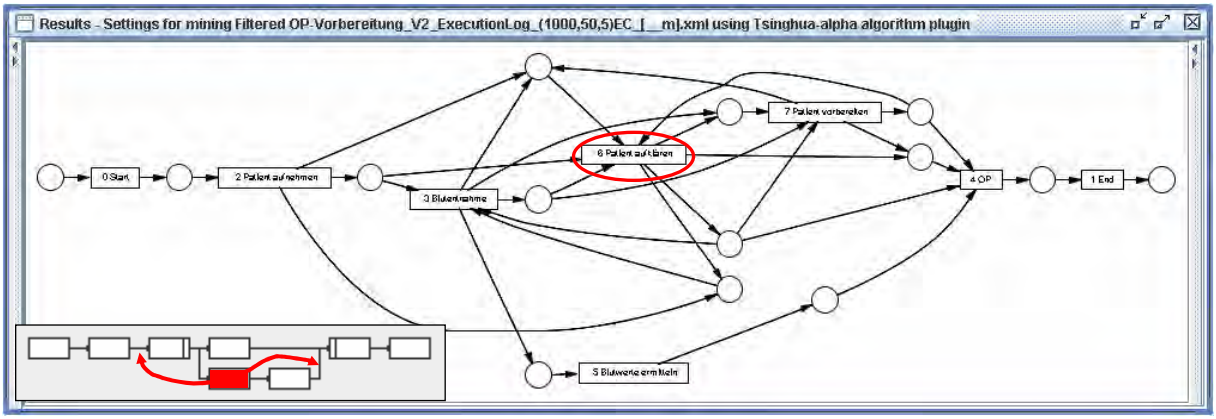


Abbildung 3-26: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 3 individuell geändert (nur Verschieben erlaubt)

Abbildung 3-27 zeigt den Ergebnis-Graphen, wenn bei acht Instanzen zusätzliche Aktivitäten (parallel) eingefügt wurden. Im Graph markiert sind die vier Aktivitäten, von denen jeweils zwei zusammen eingefügt wurden (entweder ‚*Erweiterte Bluttests*' und ‚*Urintest*' oder ‚*Urintest*' und ‚*Speicheltest*'). Auch hier lässt sich nur noch eine gewisse Grundstruktur erkennen, weil für eine genauere Prozessrekonstruktion zu wenig geänderte Daten vorhanden sind.

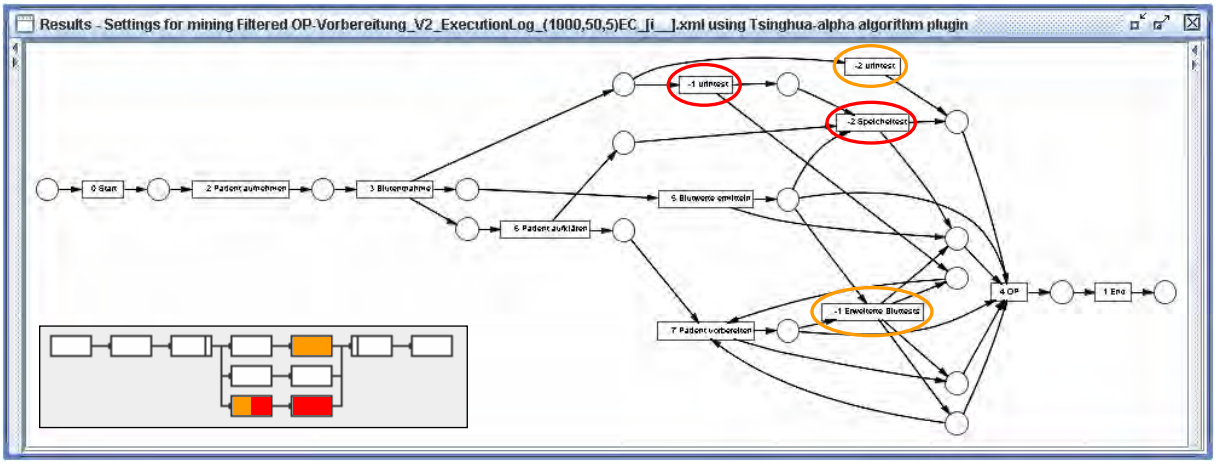


Abbildung 3-27: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 8 individuell geändert (nur Einfügen erlaubt)

Das zeigt sich, wenn man im Vergleich dazu Abbildung 3-28 (die eingefügten Schritte sind wieder markiert) betrachtet. Hier wurden 38 Instanzen individuell geändert (was ca. 4 % *Noise* entspricht) und der ursprüngliche Prozess wird fast richtig wiedergegeben. Vor allem die Parallelität der einzelnen Pfade wird diesmal richtig erkannt, auch wenn einige Beziehungen zwischen Transitionen und Stellen nicht dem Ausgangsprozess entsprechen. Man kann also sagen, dass bei zunehmender Menge *Noise* bei gleichzeitig konstant vielen verschiedenen Änderungen die Wiedererkennungsrate eines Prozesses steigt. Dies lässt sich dadurch erklären, dass wenn bestimmte Änderungen häufiger im Log vorkommen, diese vielleicht gar keine instanzspezifischen Änderungen abbilden, sondern seltene Pfade in einem Prozess be-

schreiben. Wenn also bestimmte Änderungen (sehr) häufig im Log auftreten, werden diese nicht mehr als *Noise*, sondern als optionale Schritte erkannt. Der *Tsinghua-Alpha-Algorithmus* benötigt bei drei unterschiedlichen Einfügeoperationen nicht einmal 5 % Vorkommen im Log, um sie als optionale Schritte zu erkennen. Auch die anderen Algorithmen können solche optionalen Schritte erkennen, benötigen aber deutlich mehr Einträge im Log (z.B. *Alpha-Algorithmus*: ca. 20 %).

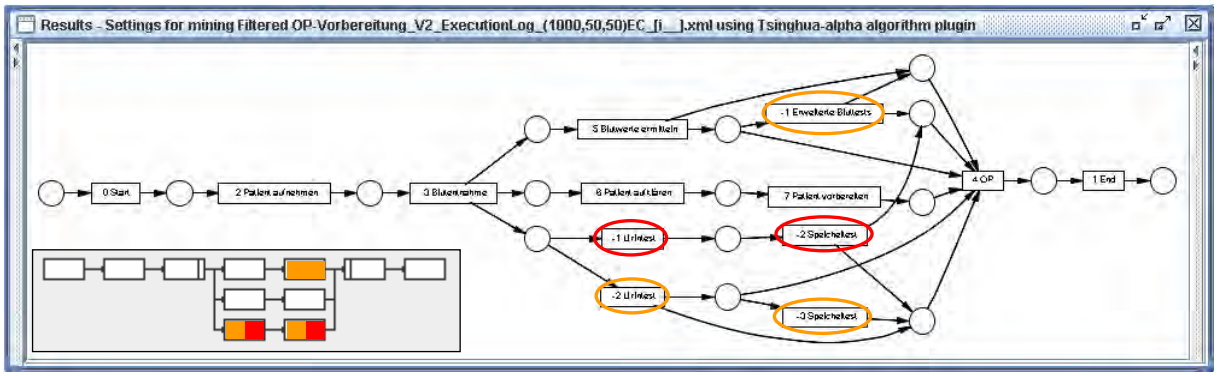


Abbildung 3-28: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 38 individuell geändert (nur Einfügen erlaubt)

Abbildung 3-29 zeigt nun den Ergebnis-Graphen, wenn von den 1000 Instanzen 18 individuell verändert wurden, wobei sowohl Einfügen, Löschen als auch Verschieben erlaubt war. Nur noch mit viel gutem Willen lässt sich die Grundstruktur des Prozesses erkennen, weil 18 Instanzen nicht ausreichen, um genügend Daten für sechs verschiedene Änderungen zu liefern. Zusammenfassend lässt sich also auch für den *Tsinghua-Alpha-Algorithmus* sagen, dass einzelne einfache Änderungen erkannt werden können, da hier bereits wenige Instanzen im Log ausreichend Daten liefern, während mehrere, evtl. kombinierte Änderungen schwerer zu erkennen sind, weil dafür selten genügend Daten im Log sind. Erwähnenswert ist die Schwäche des Algorithmus bei *einfacher Noise*, da durch das Wegfallen einzelner Ereignisse im Log grundlegende Informationen für die Ausführung des Algorithmus verloren gehen.

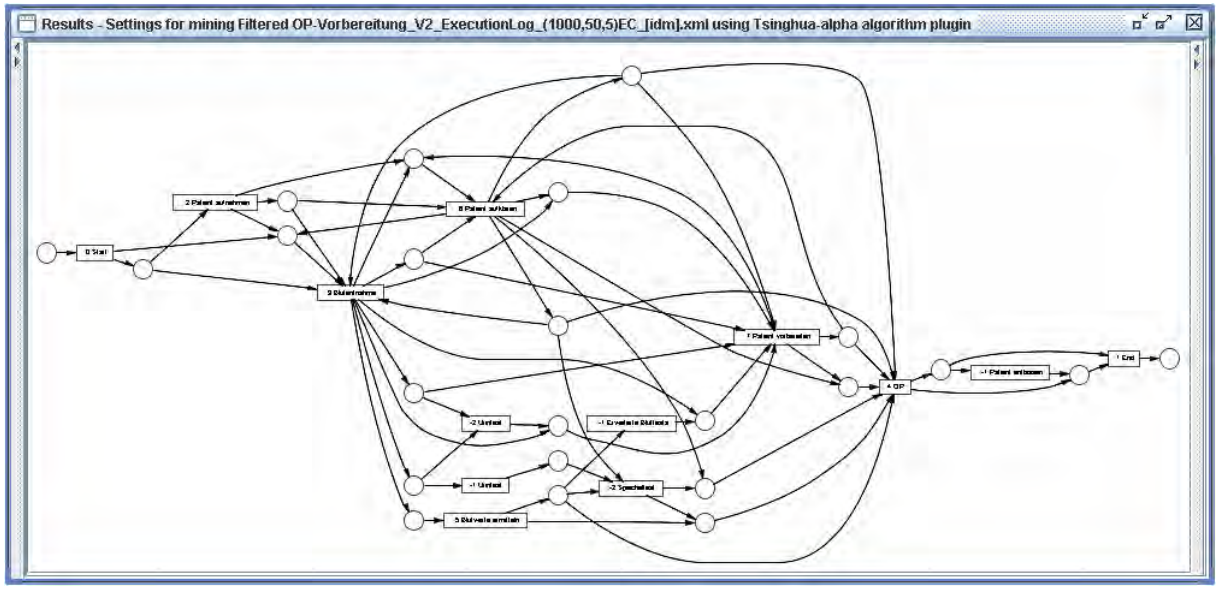


Abbildung 3-29: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 1000 Instanzen, davon 18 individuell geändert (alles erlaubt, entspricht ca. 2 % *Noise*)

3.3.4 Heuristics Miner

Weil in der Praxis die Log-Daten selten vollständig und/oder fehlerfrei sind und viele Algorithmen (z.B. *Alpha-Algorithmus*) Probleme mit *Noise* haben, wurde der *Heuristics Miner* [WeAa01, WeAa03] gezielt dahingehend entwickelt, mit fehlerhaften und unvollständigen Log-Daten zurecht zu kommen. So wird anhand von Maßzahlen entschieden, ob zwei Ereignisse zueinander in einer Nachfolgebeziehung oder in einer parallelen Beziehung stehen. Auf diese Weise werden einzelne Ausreißer im Log nicht so schwerwiegend betrachtet wie z.B. beim *Alpha-Algorithmus*, wo ein einziges fehlerhaftes Auftreten von $B>A$ das Erkennen von $A \rightarrow B$ komplett durcheinander bringen kann.

Der Algorithmus selbst läuft in drei Stufen ab. Zuerst wird eine *Dependency/Frequency-Tabelle* (D/F-Tabelle) erstellt, daraus wird dann der D/F-Graph abgeleitet und schließlich wird aus D/F-Tabelle und D/F-Graph ein Workflow-Netz rekonstruiert. Für die D/F-Tabelle wird für jeden Prozessschritt folgende Information aus dem Log entnommen:

(i) Das Gesamtvorkommen von Schritt *A* (Notation: *#A*)

(ii) Die Häufigkeit von Schritt *A* direkt nach einem anderen Schritt *B* (Notation: *#B<A*)

(iii) Die Häufigkeit von Schritt *A* direkt vor einem anderen Schritt *B* (Notation: *#A>B*)

(iv) Eine lokale Maßzahl, die die Stärke der Abhängigkeitsbeziehung zwischen *A* und einem anderen Schritt *B* angibt (Notation: $\$A \rightarrow^{L} B$)

(v) Eine globale Maßzahl, die die Stärke der Abhängigkeitsbeziehung zwischen *A* und einem anderen Schritt *B* angibt (Notation: $\$A \rightarrow B$)

Die ersten drei Maßzahlen können durch einfaches Abzählen aus dem Log entnommen werden, die lokale Maßzahl (iv) ist wie folgt definiert: $\$A \rightarrow^{L} B = (\#A>B - \#B>A) / (\#A>B + \#B>A + 1)$. Weil für die Berechnung nur lokale Information verwendet wird (Beziehung

A>B), heisst diese Maßzahl lokale Maßzahl. Sie gibt an, wie sicher es ist, dass *B* direkt auf *A* folgt, also dass *A* das Vorkommen von *B* bedingt. Bei der globalen Maßzahl (v) werden außer den lokalen Informationen auch Beziehungen von *A* und *B* berücksichtigt, die nicht direkt sind. Und zwar wird die *$A → B*-Maßzahl um einen Faktor $(\delta)^n$ erhöht, wobei δ selbst einen Wert zwischen 0.0 und 1.0 annehmen kann, so dass die Zugabe zur Maßzahl maximal 1 sein kann (wenn *B* direkt auf *A* folgt). *n* steht für die Anzahl der zwischen *A* und *B* liegenden Schritte. Diese Maßzahl wird also immer kleiner, je weiter *A* und *B* entfernt sind. Wenn umgekehrt Schritt *B* vor Schritt *A* im Log steht, wird die Abhängigkeitsmaßzahl um einen Faktor $(\delta)^n$ vermindert, wobei *n* wieder die Anzahl der dazwischen liegenden Schritte ist. Ausgehend von einem Schritt *A* wird solange nach *B* gesucht, bis entweder *A* oder *B* gefunden wird. Nachdem alle Instanzen im Log berücksichtigt wurden wird die *$A → B*-Maßzahl noch durch die kleinere Gesamtzahl der Vorkommen von *A* bzw. *B* (*min (#A, #B)*) geteilt.

Nachdem für alle Prozessschritte eine D/F-Tabelle erstellt wurde, wird daraus der D/F-Graph erzeugt. Dazu muss zuerst eine Abhängigkeitszahl *DS* (*dependency score*) berechnet werden, um zu ermitteln, welche Ereignisse (direkt) voneinander abhängen, also welche Ereignisse im Graph (direkt) aufeinander folgen. Die Abhängigkeitszahl *DS* ist dabei für zwei Ereignisse *X* und *Y* wie folgt definiert: $DS(X,Y) = (($X \rightarrow^L Y)^2 + ($X \rightarrow Y)^2)/2$. Sie wird also aus der lokalen und der globalen Maßzahl aus der D/F-Tabelle errechnet. Mit Hilfe dieser Abhängigkeitszahl werden nun zu einem gegebenen Schritt *A* die direkten Vorgänger und Nachfolger bestimmt, wobei die Beziehung mit der größten Abhängigkeitszahl *DS* am wahrscheinlichsten ist. Dazu gibt es noch zwei weitere Regeln um Rekursion und kurze Schleifen erkennen zu können, wobei Details an dieser Stelle nicht relevant sind (Näheres dazu in [WeAa03]).

Im dritten Schritt des Algorithmus wird dann aus D/F-Tabelle und D/F-Graph ein Workflow-Netz rekonstruiert. Die Grundstruktur des Graphen wird aus dem D/F-Graph entnommen, während die D/F-Tabelle dazu verwendet wird, die Typen der Splits und Joins zu ermitteln, da diese im D/F-Graph noch nicht repräsentiert sind. Dazu werden die Beziehungen von zwei im D/F-Graph parallelen Schritten *B* und *C* zueinander betrachtet. Sind beide Schritte nach einer UND-Verzweigung *A*, dann können beide Werte *#B>C* und *#C>B* in der Tabelle positiv sein, da beide Beziehungen B,C und C,B im Log vorhanden sein können. Sind *B* und *C* nach einer ODER-Verzweigung, können die Muster B,C und C,B nicht vorkommen, da nur ein Schritt von beiden in einer Instanz vorkommen kann. Alternativ dazu kann der Typ eines Split- bzw. Join-Knotens auch anhand der Häufigkeitszahlen im D/F-Graph ermittelt werden (gleiche Häufigkeit → UND-Verzweigung; ungleiche Häufigkeit, zusammen genau so häufig wie der Split/Join-Knoten → ODER-Verzweigung). In *ProM* werden die Häufigkeitszahlen aus dem D/F-Graph verwendet, um das erzeugte Workflow-Modell zu validieren, während die Ergebnisse aus der D/F-Tabelle verwendet werden, um die Typen von Split und Join zu ermitteln. Wenn dann für alle Schritte *A* und *B* bestimmt wurde, in welcher Beziehung sie zueinander stehen (*A→B, B→A, A#B* oder *A||B*), kann der *Alpha-Algorithmus* verwendet werden, um die Information in ein Workflow-Netz zu übersetzen.

Enthalten die zugrunde liegenden Log-Daten keine *Noise*, dann hat der *Heuristics Miner* auch keine Probleme, den Workflow-Prozess wiederherzustellen. Und beim Beispielprozess OP-Vorbereitung reichen auch bei 5 % *einfacher Noise* bereits 50 Instanzen, um den Prozess korrekt zu erkennen. Das Ergebnis wird ausschnittsweise in Abbildung 3-30 gezeigt. An den

Häufigkeitszahlen kann man erkennen, dass nicht jedes Ereignis in jedem Log enthalten ist (z.B. das Startereignis von ‚*OP*' ist nur 44-mal im Log enthalten). Trotzdem werden alle Abhängigkeiten zwischen den einzelnen Schritten korrekt erkannt.

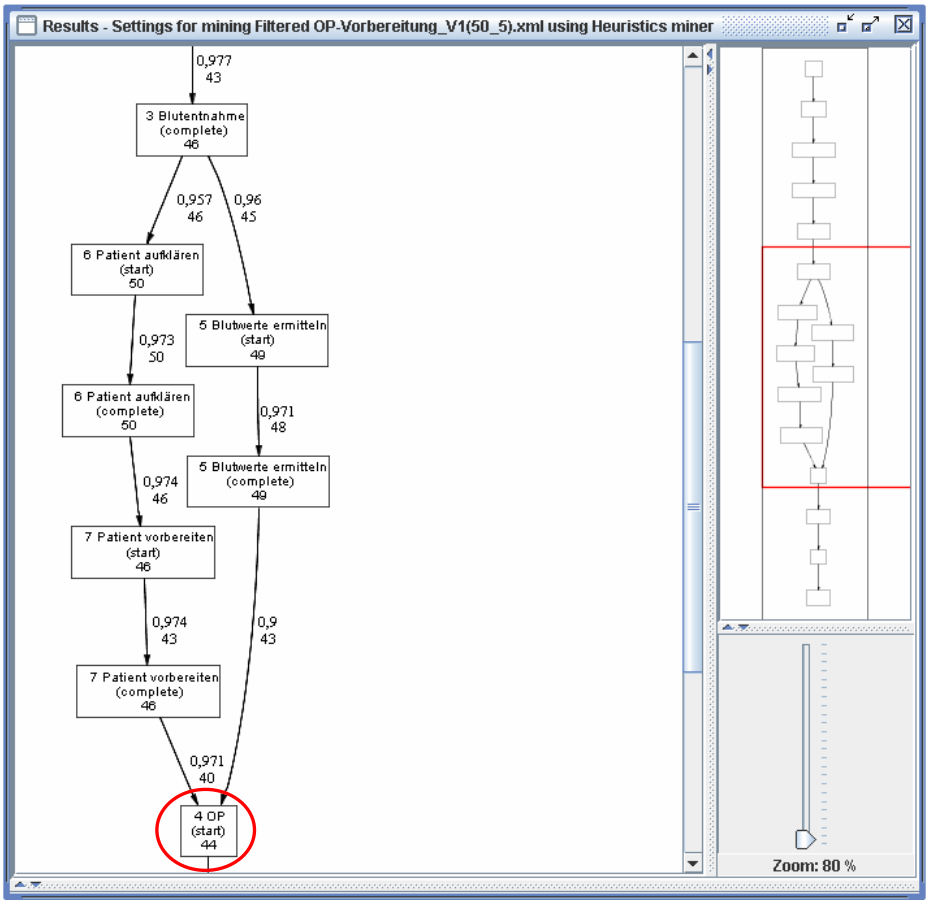


Abbildung 3-30: Beispiel Heuristics Miner: OP-Vorbereitung, V1, 50 Instanzen, 5 % *einfache Noise*

Der Prozess wird als so genanntes heuristisches Netz (bzw. Heuristiknetz) angezeigt, er kann aber auch in ein Petrinetz umgewandelt werden. Ein heuristisches Netz hat im Gegensatz zu einem Petrinetz keine Stellen (siehe Abbildung 3-30), die Knoten selbst entsprechen den Ereignissen im Log. Für jeden Knoten (≙ Ereignis) und jede Kante (≙ Beziehung zwischen zwei Ereignissen) wird annotiert, wie oft diese(r) im Log vorkommt, da aufgrund dieser Häufigkeitswerte ja berechnet wird, wie wahrscheinlich eine Beziehung ist. Denn eine Beziehung, die nur wenige male im Log vorkommt, lässt den Verdacht aufkommen, dass diese Beziehung nur zufälligerweise im Log enthalten ist, aus welchen Gründen auch immer. Deshalb ist es auch für den *Heuristics Miner* besser, wenn mehr Log-Daten vorhanden sind. Denn je mehr Instanzdaten vorhanden sind, umso mehr Störungen können als solche erkannt werden. So sind bei 100 Instanzen vom bekannten, einfachen Beispielprozess auch 15 % *einfache Noise* kein Problem und der Prozess wird wieder korrekt dargestellt, erst bei 20 % *einfacher Noise* zeigen sich erste Fehler.
Diese können durch gezieltes Setzen der Parameter (z.B. Abhängigkeitsschwellwert) weiter

verringert werden, wie Abbildung 3-31 zeigt. Links mit den Standardparametern (z.B. Abhängigkeitsschwellwert = 0.9), rechts wurde der Abhängigkeitsschwellwert (entspricht der lokalen bzw. globalen Maßzahl in der D/F-Tabelle) etwas erhöht (auf 0.95). Neben dem Abhängigkeitsschwellwert bringt auch die Veränderung des Schwellwertes ‚*relative-to-best*' einen gewissen Effekt, so werden nur die Schritte berücksichtigt, deren berechnete Maßzahl sich innerhalb des angegebenen Rahmens befindet. Des Weiteren können die Schwellwerte für ein- und zweielementige Schleifen und UND-Verzweigungen, sowie die Anzahl der positiven Beobachtungen und ein Abhängigkeitsdivisor verändert werden. Je nachdem, ob fehlerhafte Beziehungen entfernt, oder seltene Beziehungen erkannt werden sollen, müssen die Parameterwerte erhöht oder verringert werden. Weitere Erklärungen zu den einzelnen Parametern befinden sich in der *ProM*-Hilfe zum *Heuristics Miner*.

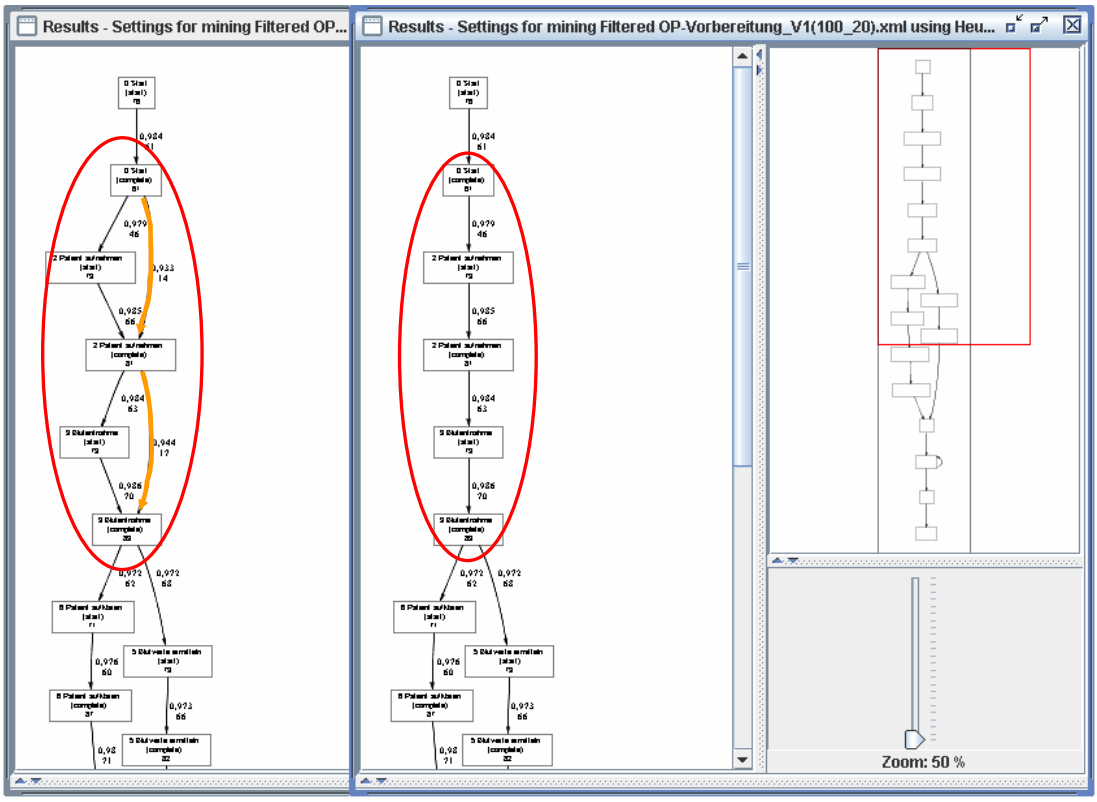


Abbildung 3-31: Beispiel Heuristics Miner: OP-Vorbereitung, V1, 100 Instanzen, 20 % *einfache Noise*

Auch wenn statt einzelner Ereignisse (entspricht *einfacher Noise*) komplette Schritte in manchen Log-Daten fehlen, hat der *Heuristics Miner* keinerlei Probleme den ursprünglichen Prozess wieder herzustellen. Dies zeigt Abbildung 3-32, hier wurde in zwei von 1000 Instanzen der Schritt ‚*Patient aufnehmen*' gelöscht, was an den Häufigkeitswerten (Zahlen im markierten Bereich) abgelesen werden kann (Die Aktivität ist nur 998-mal im Log vorhanden). Soll dieses Fehlen des Schrittes im Log nicht als Fehler, sondern als seltene Ausnahme behandelt werden und deshalb auch im Graph erscheinen, so müssen nur die Parameter entsprechend verändert werden.

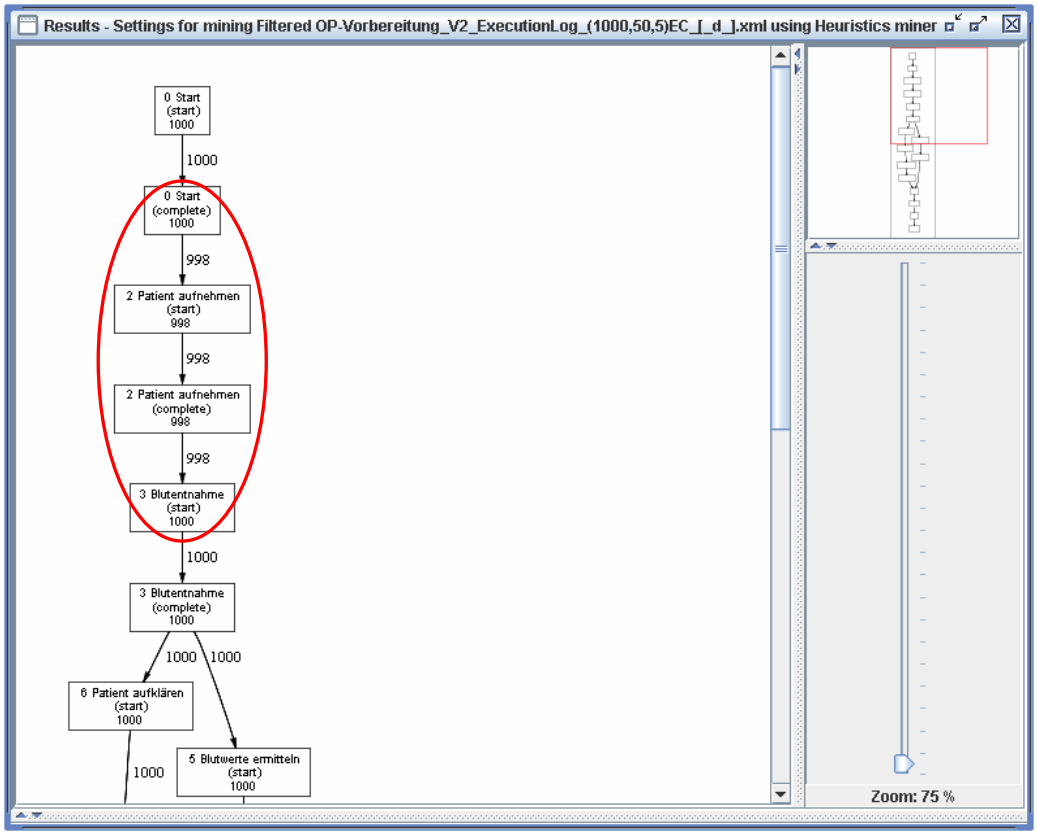


Abbildung 3-32: Beispiel Heuristics Miner: OP-Vorbereitung, V2, 1000 Instanzen, davon 2 individuell verändert (nur Löschen erlaubt)

Ähnlich unbeeindruckt zeigt sich der Algorithmus auch, wenn sich im Log verschobene Aktivitäten befinden. Abbildung 3-33 zeigt den Beispielprozess bei 1000 Instanzen, wobei bei dreien davon die Aktivität ‚*Patient aufklären*' verschoben wurde. Wieder verraten nur die Häufigkeitsangaben, dass nicht alle 1000 Instanzen gleich abgelaufen sind.

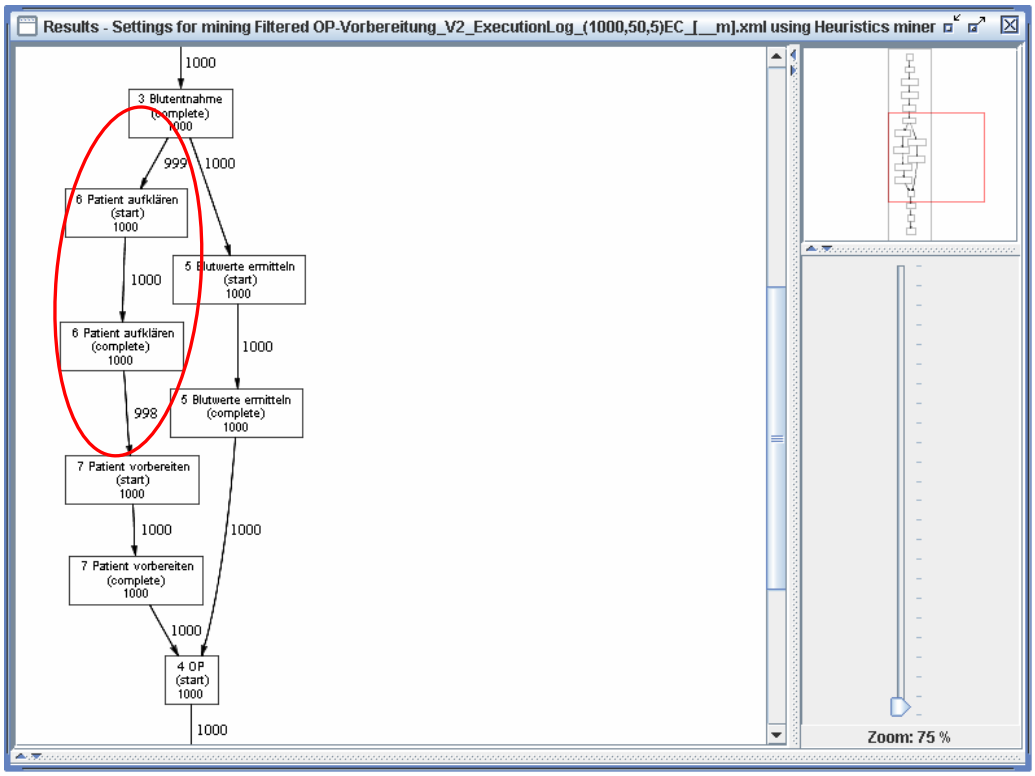


Abbildung 3-33: Beispiel Heuristics Miner: OP-Vorbereitung, V2, 1000 Instanzen, davon 3 individuell geändert (nur Verschieben erlaubt)

Das Einfügen von Schritten gehört für den *Heuristics Miner* zu einer Art von *Noise*, mit der er nicht zurechtkommt. Auch wenn sämtliche Parameter verändert werden, kann der Algorithmus die zusätzlichen Aktivitäten bei acht von 1000 Instanzen nicht als Fehler ausblenden. Das liegt wahrscheinlich daran, dass die Beziehungen der ‚eingefügten' Aktivitäten zu den ‚normalen' eine hohe Abhängigkeit haben, auch wenn sie nur selten im Log stehen. Abbildung 3-34 zeigt einen Ausschnitt aus dem erzeugten Graphen, wobei deutlich zu sehen ist, dass die eingefügten Aktivitäten (im Graph markiert) das Erkennen des Graphen deutlich erschweren.

Ähnlich wie der *Tsinghua-Alpha-Algorithmus* erkennt der *Heuristics Miner* aber schon bei weniger als 5 % dieser Änderungen (also nur bei ein paar Änderungen mehr), dass es sich um seltene Ausnahmesituationen handelt, und gibt den Graph entsprechend korrekt zurück. Die Änderungen in Abbildung 3-34 sind also zu viel, um als *Noise* erkannt zu werden und zu wenig, um als seltene Optionen erkannt zu werden. Für reale Situationen ist dieses Verhalten nicht von Nachteil, da das Einfügen zusätzlicher Aktivitäten nicht fehlerhaft im Log erscheinen kann. Wenn eine Aktivität aber tatsächlich eingefügt wurde, und sei es nur in sehr speziellen Ausnahmefällen, dann ist es ja nur korrekt, wenn diese Änderung nicht als *Noise* unterschlagen wird.

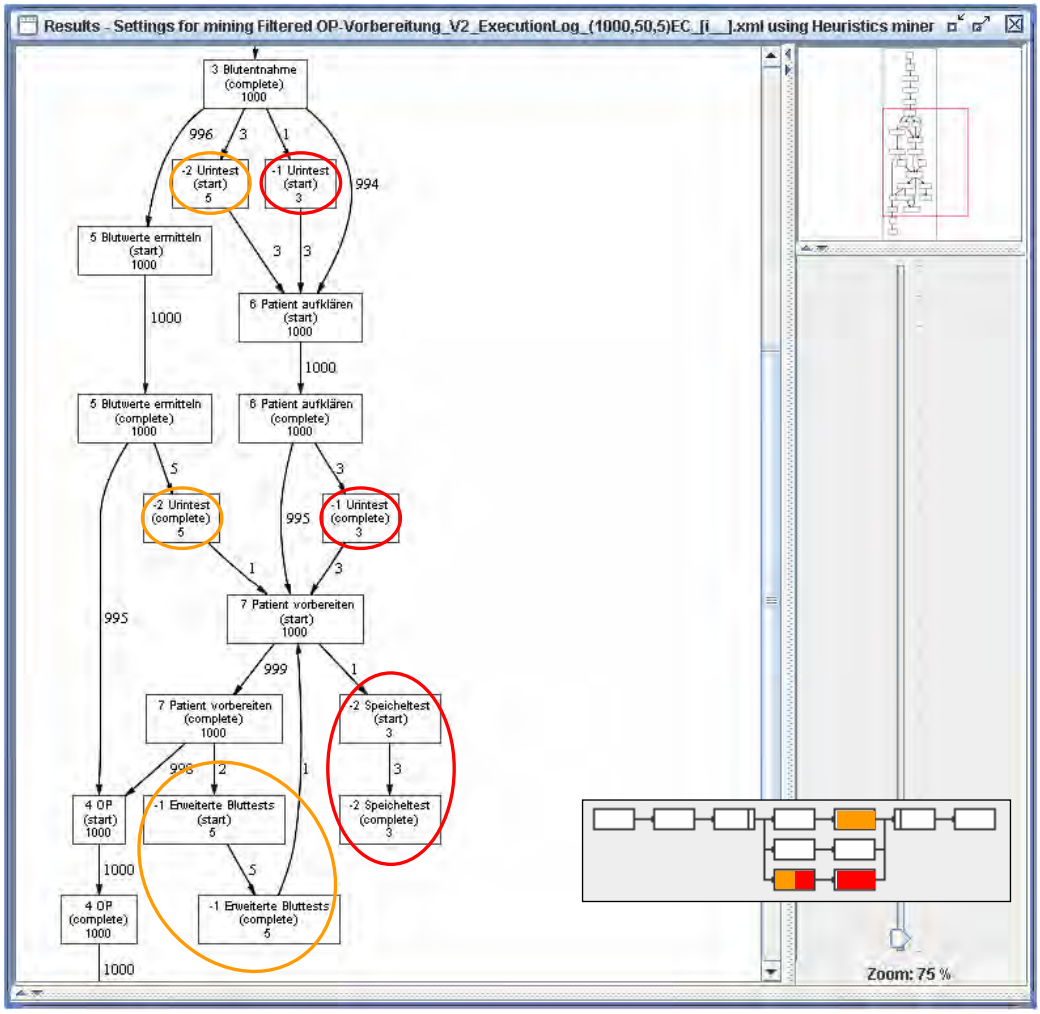


Abbildung 3-34: Beispiel Heuristics Miner: OP-Vorbereitung, V2, 1000 Instanzen, davon 8 individuell geändert (nur Einfügen erlaubt)

Aufgrund der Schwierigkeiten bei *Noise* durch Einfügen gibt es auch Probleme, wenn alle drei Änderungsoperationen erlaubt sind, wie auch Abbildung 3-35 zeigt. Hier lässt sich auch nicht mehr so einfach sagen, dass alle seltenen Änderungen erkannt werden sollen, weil Löschungen und Verschiebungen ja durchaus auch auf Fehler im Log zurückzuführen sein könnten. So ist es also auch für den *Heuristics Miner* schwierig, fehlerhafte Logs in korrekte Graphen umzuwandeln, wenn sich auch noch wenige seltene Änderungen in den Log-Daten befinden. Und genau das kann bei realen Log-Daten durchaus der Fall sein, dass sich einige Fehler und Unvollständigkeiten (also *Noise*) mit wenigen seltenen Änderungen vermischen.

Zusammenfassend lässt sich aber sagen, dass der *Heuristics Miner* am besten von den hier vorgestellten Algorithmen mit fehlerhaften Logs zurecht kommt, weil er fehlende/gelöschte Einträge in den Log-Daten problemlos behandeln kann, genauso wie er mit verschobenen Einträgen gut zurecht kommt. Und auch wenn alle Änderungen zusammen kommen, lässt sich immer noch eine gewisse Grundstruktur im Graphen erkennen, während andere Algorithmen hier noch abwegigere Graphen zurückliefern.

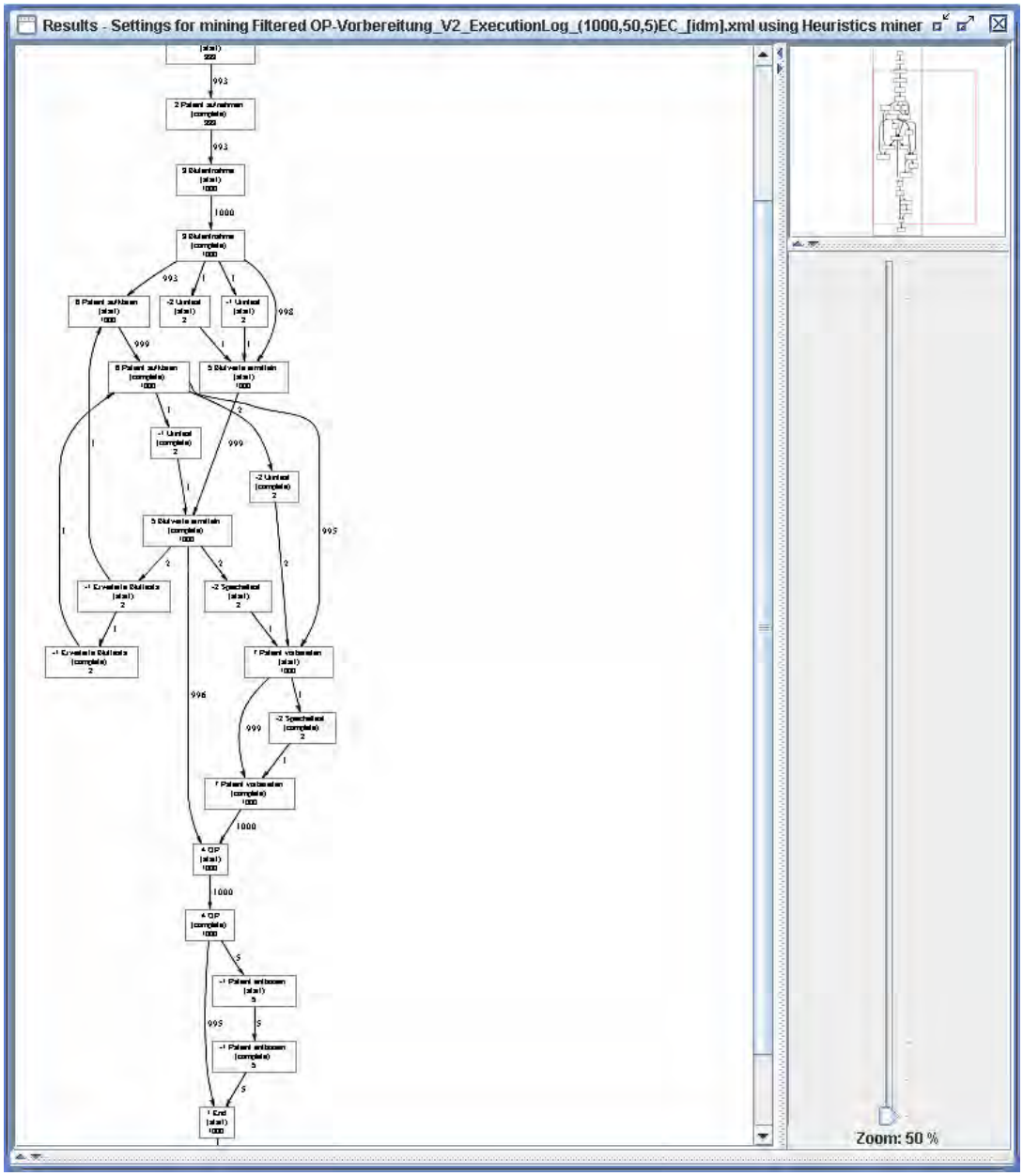


Abbildung 3-35: Beispiel Heuristics Miner: OP-Vorbereitung, V2, 1000 Instanzen, davon 18 individuell geändert (alles erlaubt, entspricht ca. 2 % *Noise*)

3.3.5 Auswirkungen komplexerer Beispielprozesse

In diesem Kapitel soll gezeigt werden, welche Auswirkungen verschieden komplexe Ausgangsprozesse auf das *Process Mining* Ergebnis haben. Als Basis (einfachster Prozess) wird der Prozess ‚OP-Vorbereitung' verwendet, der bereits eingeführt wurde. Im Folgenden werden noch drei weitere Prozesse vorgestellt, und wie die bekannten Algorithmen damit fertig werden. Der erste Prozess ‚*Template5, V1*' ist ein Prozess mit zwei ineinander verschachtelten UND-Verzweigungen und zwei Synchronisations-Kanten zwischen den parallelen Pfaden. Abbildung 3-36 zeigt den Prozess nachdem er im *Demonstrator* erstellt wurde. Die beiden Synchronisations-Kanten sind zur Verdeutlichung fett dargestellt.

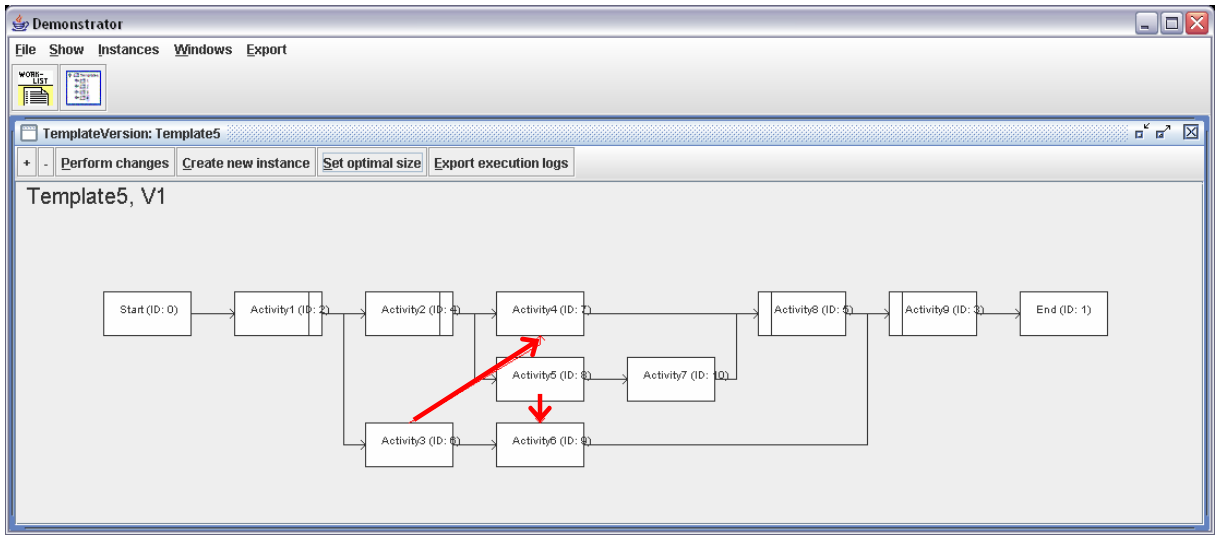


Abbildung 3-36: Beispielprozess: Template5, V1 (im Demonstrator)

Weil beim Prozess ‚*OP-Vorbereitung*' 50 Instanzen (ohne *Noise*) ausreichend waren, um den Prozess zu erzeugen, werden auch hier 50 Instanzen verwendet, um die Unterschiede zwischen den Algorithmen herauszufinden. Wie man in Abbildung 3-37 sieht, reichen dem *Alpha-Algorithmus* bei diesem Prozess die 50 zufälligen Instanzen nicht, um den Prozess komplett wieder herzustellen. Speziell die genaue Lage der Synchronisations-Kanten (im Graph zur Verdeutlichung markiert) lässt sich nicht ermitteln, weil noch einige zusätzliche falsche Kanten im Graph enthalten sind. Die Grundstruktur des Graphen lässt sich aber bereits erkennen. Weitere Versuche mit 100, 250 und 500 zugrunde liegenden Instanzen zeigen, dass der *Alpha-Algorithmus* erst bei 500 Instanzen die Informationen bekommt, um die Synchronisations-Kanten als solche zu erfassen, wenn man bei Petrinetzen überhaupt von Synchronisations-Kanten reden kann.

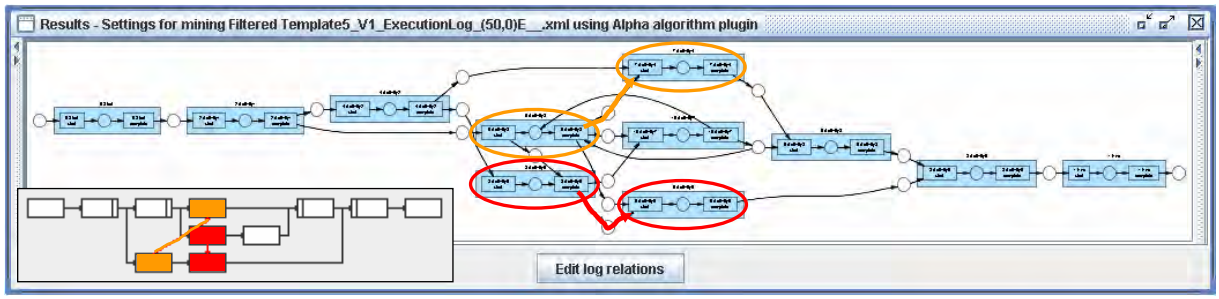


Abbildung 3-37: Beispiel Alpha-Algorithmus: Template5, V1, 50 Instanzen, keine Noise

Auch beim *Multi-Phase-Mining* reichen die 50 Instanzen nicht aus, um den Graph korrekt rekonstruieren zu können. Sollen nur einzelne Instanzen betrachtet werden, so ist der Algorithmus durchaus in der Lage, einen sinnvollen Graph zu erzeugen, wie Abbildung 3-38 zeigt. Bei dieser Instanz werden aber die einzelnen Aktivitäten so angeordnet, wie sie hier abgelaufen sind. D.h., dass die Ausführungsreihenfolge zwar für diese eine Instanz gilt, aber nicht für alle anderen genauso gelten muss. Das zeigt sich dann, wenn alle Instanzen zu einem Graph

zusammengefasst werden, denn sogar wenn die Information von 1000 Instanzen zugrunde liegt, kann *Multi-Phase-Mining* den ursprünglichen Prozess nicht wiederherstellen. Es scheint, als ob der Algorithmus große Probleme mit den Synchronisations-Kanten hat, denn die Schritte mit ein- oder ausgehenden Synchronisations-Kanten stechen durch falsche Platzierung besonders hervor. Wie Abbildung 3-39 zeigt, unterscheidet sich das Ergebnis bei 50 (ausgewählten) Instanzen deutlich vom Ausgangsprozess, aber auch bei 1000 Instanzen kommt das Ergebnis dem Ausgangsprozess nur unwesentlich näher. Die Aktivitäten mit ein- oder ausgehenden Synchronisations-Kanten sind im Graph markiert, dazu zeigt eine gestrichelte Linie jeweils die Lage der ursprünglichen Synchronisations-Kanten, da diese Beziehungen im Ergebnisgraph nicht mehr existieren.

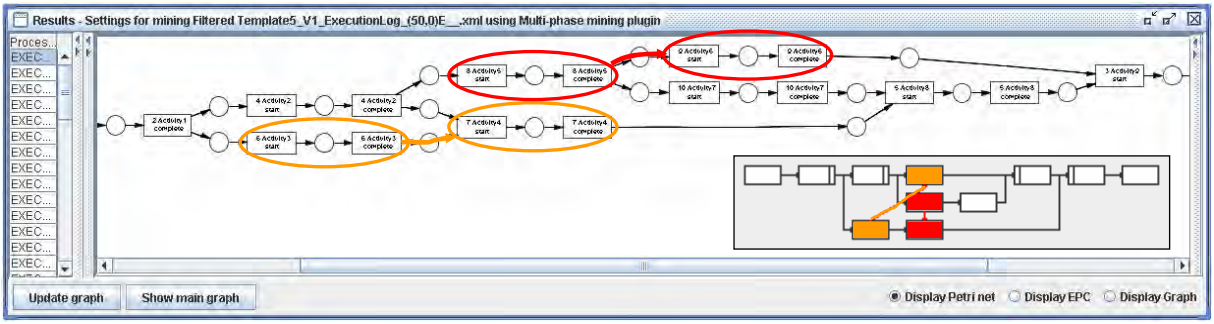


Abbildung 3-38: Beispiel Multi-Phase-Mining: Template5, V1, 50 Instanzen (davon Instanz 0 ausgewählt), keine Noise

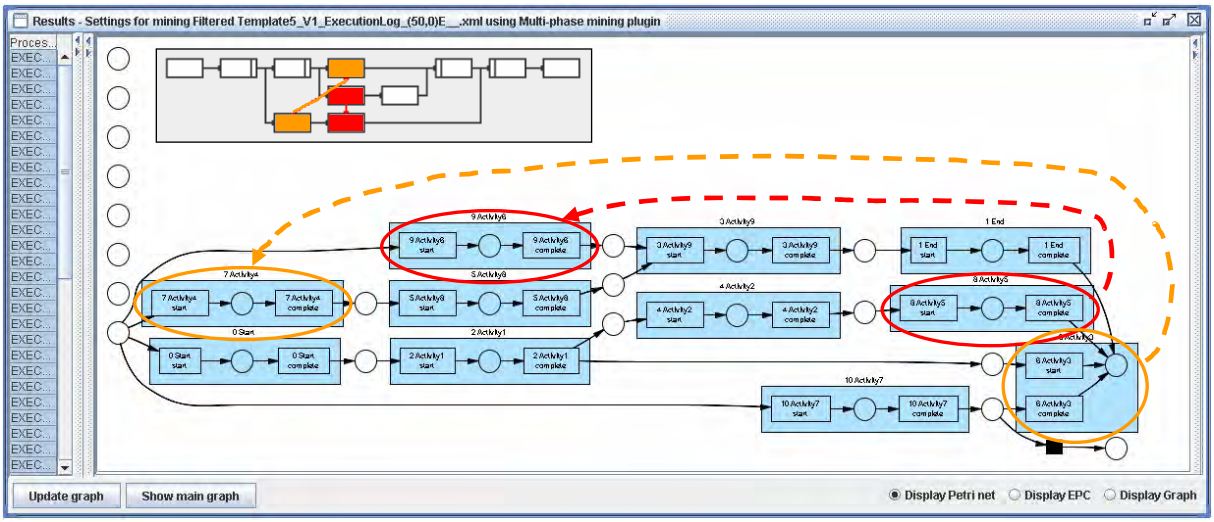


Abbildung 3-39: Beispiel Multi-Phase Mining: Template5, V1, 50 Instanzen (davon alle ausgewählt), keine Noise

Mit den 50 Instanzen gut zurecht kommt der *Tsinghua-Alpha-Algorithmus*, wie Abbildung 3-40 zeigt. Ihm reichen sogar 25 Instanzen um den ursprünglichen Prozess zu rekonstruieren. Die gezielte Betrachtung von Start- und Endereignis eines Schrittes scheint hilfreich zu sein, bei der Bestimmung der Beziehungen zwischen den einzelnen Aktivitäten. Auch die Synchronisations-Kanten können relativ gut als solche erkannt werden, da sie zwischen den ein-

zelnen Pfaden der UND-Verzweigungen verlaufen. Ebenfalls erkannt wird die Abhängigkeit zwischen Aktivität3 und Aktivität6 (im Bild der obere Pfad, durch eine gestrichelte Linie markiert). Mit dieser Abhängigkeit haben die anderen Algorithmen (vgl. auch Abbildung 3-39) oft Probleme, da Aktivität6 durch die eingehende Synchronisations-Kante nicht unbedingt unmittelbar nach Aktivität3 im Log auftaucht.

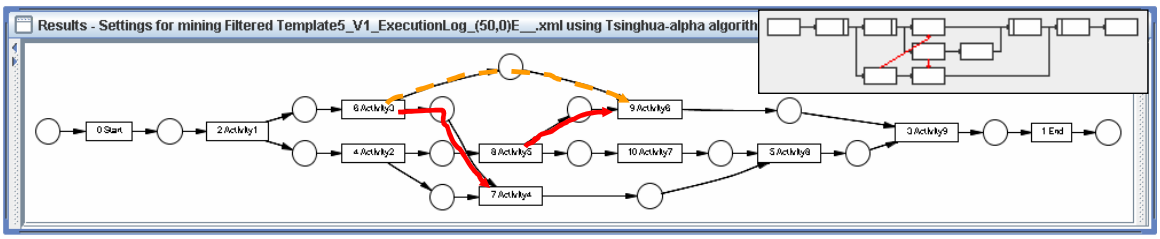


Abbildung 3-40: Beispiel Tsinghua-Alpha-Algorithmus: Template5, V1, 50 Instanzen, keine Noise

Auch der *Heuristics Miner* erkennt die Abhängigkeit von Aktivität3 und Aktivität6 bei 50 Instanzen noch nicht, wie Abbildung 3-41 zeigt (Aktivität3 ist links, Aktivität6 rechts eingekreist). Eine Veränderung der *Mining* Parameter bringt hier keinen Erfolg, erst die Erhöhung des zugrunde liegenden Datensatzes auf 250 Instanzen ermöglicht es dem *Heuristics Miner* den ursprünglichen Prozess wiederherzustellen.

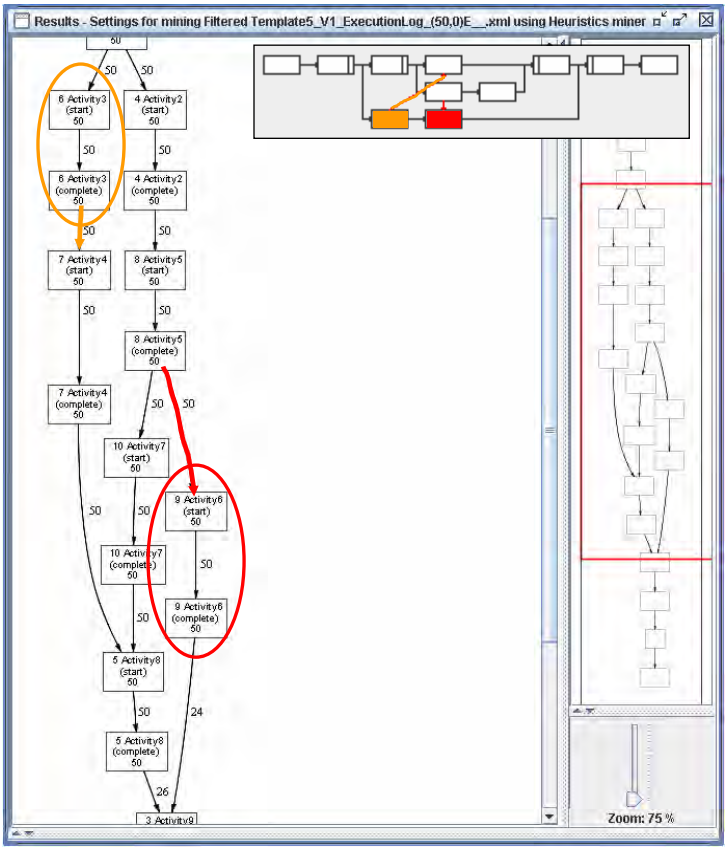


Abbildung 3-41: Beispiel Heuristics Miner: Template5, V1, 50 Instanzen, keine Noise

Der etwas komplexere Prozess ‚*Template5, V1*' zeigt recht gut, wie die einzelnen Algorithmen mit den Log-Daten zurechtkommen. Während der *Tsinghua-Alpha-Algorithmus* scheinbar mühelos mit dem Prozess klar kommt, haben die anderen Algorithmen schon mehr Probleme. Erst wenn bedeutend mehr Daten vorhanden sind, können auch der *Heuristics Miner* (250 Instanzen) und der *Alpha-Algorithmus* (500 Instanzen) den ursprünglichen Prozess wieder herstellen. Dem *Multi-Phase-Mining* reichen selbst 1000 Instanzen nicht, um alle Instanzen zu einem korrekten Prozess zusammenzufassen, während die Visualisierung einzelner Prozessinstanzen relativ gut geht.

Als nächstes soll der Prozess ‚*Template8, V1*' betrachtet werden. Dieser Prozess zeichnet sich durch eine UND-Verzweigung mit acht parallelen Pfaden aus. Für diesen Prozess ergeben sich mehr als 40000 (8!) verschiedene mögliche Ausführungsreihenfolgen, welche naturgemäß nicht alle in einem Log mit 50 Instanzen vorkommen können. Deshalb ist für die Algorithmen in diesem Fall die Erkennung der Parallelität das oberste Ziel. Abbildung 3-42 zeigt den Prozess, nachdem er im *Demonstrator* erstellt wurde.

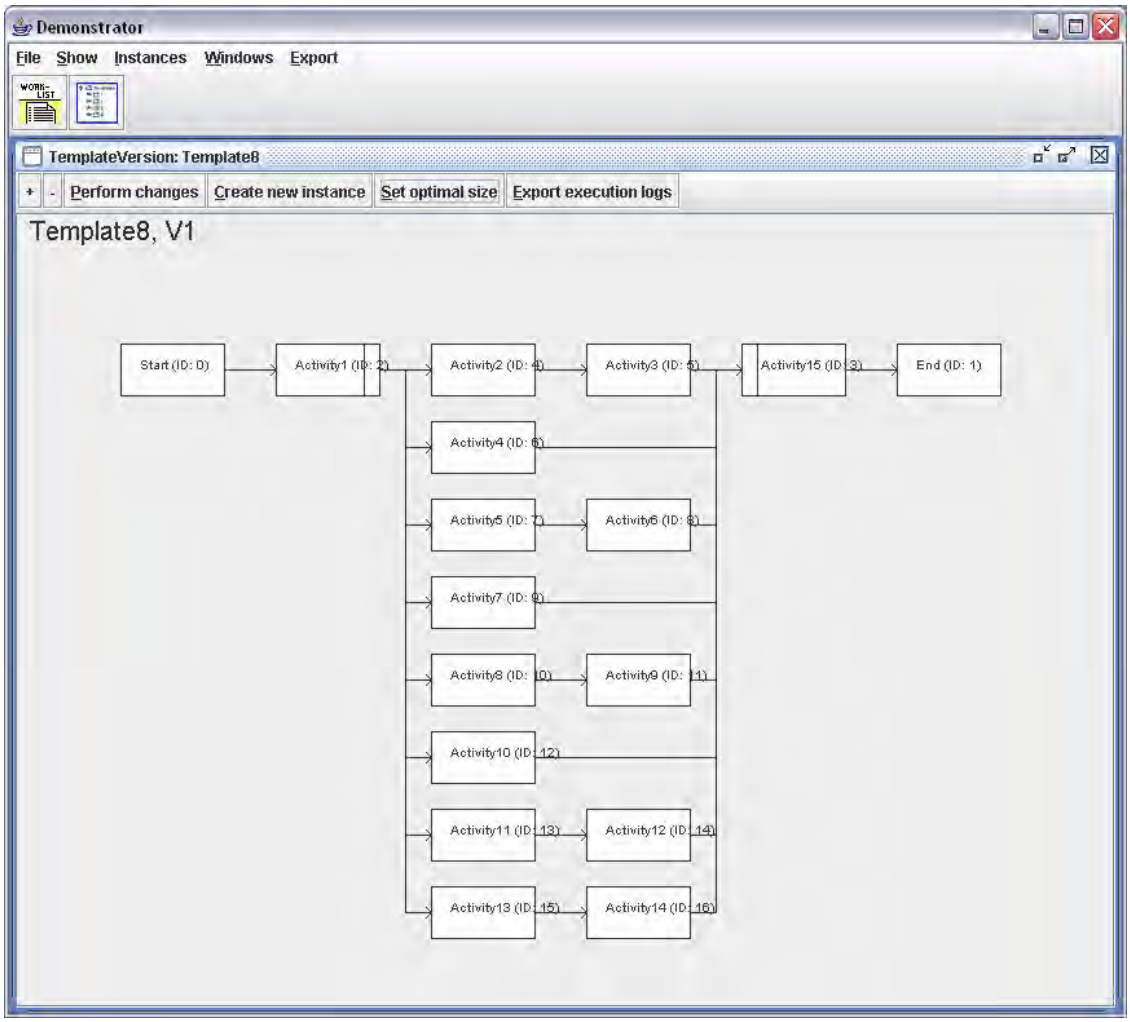


Abbildung 3-42: Beispielprozess: Template8, V1 (im Demonstrator)

Auch bei diesem Prozess reichen dem *Alpha-Algorithmus* die 50 Instanzen nicht, um den Prozess komplett wiederherstellen zu können. Wie Abbildung 3-43 ausschnittsweise zeigt, werden nur Start- und Endknoten sowie Split- und Join-Knoten sicher erkannt, die Ausführungsreihenfolge der anderen Schritte lässt sich nicht feststellen, obwohl der Algorithmus die Parallelität grundsätzlich erkannt hat. Erst bei 500 Instanzen wird die Parallelität vollständig erkannt und nur noch korrekte Kanten angezeigt.

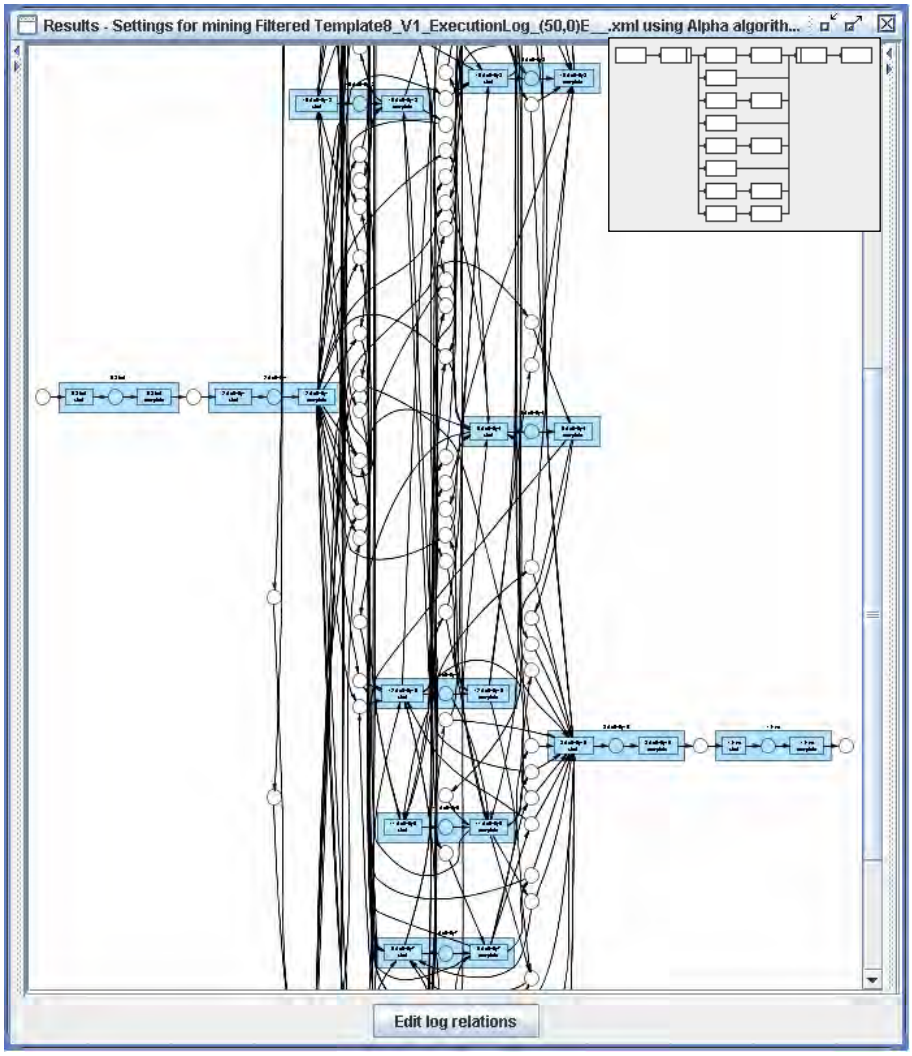


Abbildung 3-43: Beispiel Alpha-Algorithmus: Template8, V1, 50 Instanzen, keine Noise

Beim *Multi-Phase-Mining* werden ebenfalls 500 Instanzen benötigt, um die Parallelität zwischen den einzelnen Aktivitäten korrekt zu erkennen, wobei sich bei 50 Instanzen im Gegensatz zum *Alpha-Algorithmus* noch nicht einmal die Grundstruktur erkennen lässt.

Der *Tsinghua-Alpha-Algorithmus* kann schon bei 25 Instanzen den ursprünglichen Prozess korrekt rekonstruieren. Hier zeigt sich wieder, dass die explizite Erkennung der Parallelität deutliche Vorteile gegenüber den anderen Algorithmen bringt, was die Anzahl der benötigten Instanzen betrifft. Abbildung 3-44 zeigt den Graph bei 50 Instanzen, um die Vergleichbarkeit zu gewährleisten. Im Graph markiert sind die Aktivitäten, die allein auf einem Pfad sind (‚*Ac-*

tivity4', ‚*Activity7*' und ‚*Activity10*'). Diese werden von anderen Algorithmen erst bei (deutlich) mehr Instanzen als parallel erkannt (vgl. auch Abbildung 3-45).

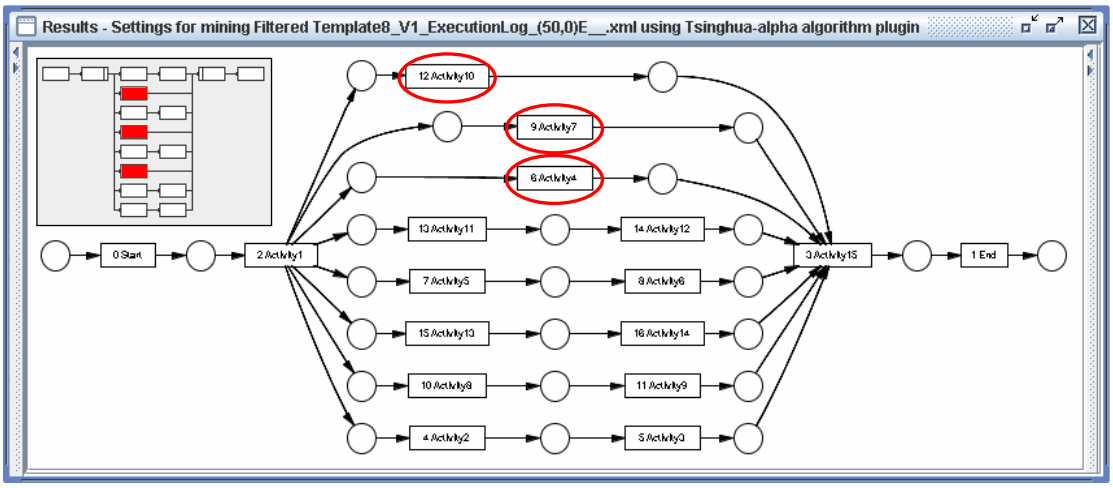


Abbildung 3-44: Beispiel Tsinghua-Alpha-Algorithmus: Template8, V1, 50 Instanzen, keine Noise

Abbildung 3-45 zeigt das Ergebnis des *Heuristics Miner*, wie zu erkennen reichen ihm 50 Instanzen nicht, den ursprünglichen Prozess wieder herzustellen. Speziell die Parallelität der Aktivitäten, die allein einen Pfad bilden (‚*Activity4*', ‚*Activity7*' und ‚*Activity10*', im Graph markiert), wird noch nicht korrekt erkannt. Der Algorithmus nimmt bei 50 Instanzen noch fälschlicherweise an, dass diese Aktivitäten voneinander abhängen. Durch die Veränderung der Parameter kann das Ergebnis noch etwas verbessert werden, doch den korrekten Prozess erhält man erst, wenn man die Anzahl der Instanzen auf ca. 100 erhöht. Man kann aber durch eine Betrachtung der annotierten Häufigkeitszahlen erkennen, welche Beziehungen seltener sind. Denkt man sich diese weg, kommt man dem tatsächlichen Prozess wieder ein Stück näher. So kann man erkennen, dass die Häufigkeitszahlen zwischen den Aktivitäten ‚*Activity4*', ‚*Activity7*' und ‚*Activity10*' mit ‚*8*' und ‚*15*' deutlich niedriger sind als die Häufigkeitszahlen zwischen den beiden Ereignissen der Aktivitäten (‚*50*' bzw. ‚*32*').

Auch bei diesem Prozess zeigen sich Unterschiede zwischen den einzelnen Algorithmen. Wieder erkennt der *Tsinghua-Alpha-Algorithmus* den zugrunde liegenden Prozess mit den wenigsten Informationen. Man muss dazu sagen, dass eines der Hauptmerkmale des Algorithmus die explizite Erkennung von Parallelität ist, was sich natürlich bei diesem Prozess vorteilhaft auswirkt. Noch relativ gut erkennt der *Heuristics Miner* den Prozess, wobei schon etwas mehr Informationen nötig sind, um den Prozess korrekt zu erkennen. Deutlich mehr Log-Daten benötigen der *Alpha-Algorithmus* und *Multi-Phase-Mining* um den Prozess wieder herstellen zu können, weil die vielen verschiedenen Ausführungsreihenfolgen bei diesem Prozess bei den Algorithmen erstmal für Verwirrung sorgen.

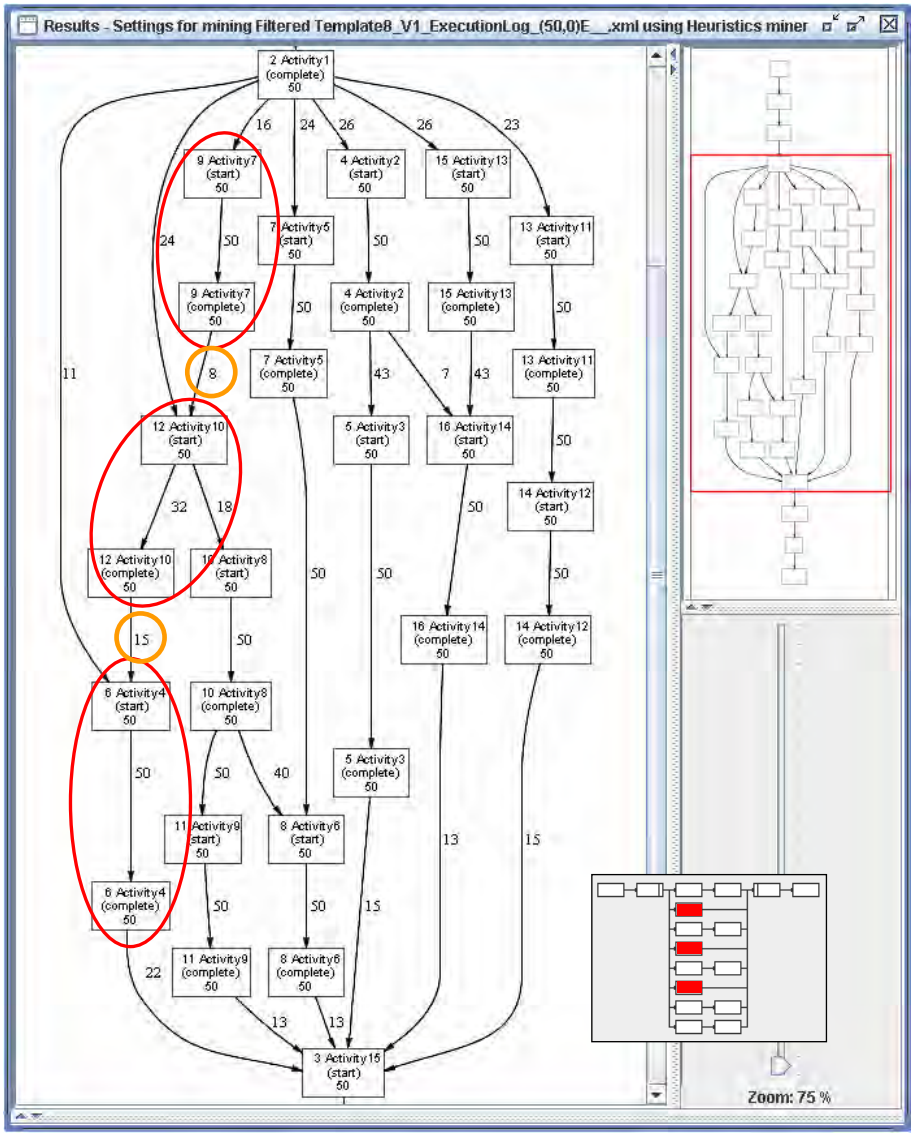


Abbildung 3-45: Beispiel Heuristics Miner: Template8, V1, 50 Instanzen, keine Noise

Zuletzt soll der Prozess ‚*Template10, V1*' betrachtet werden. Dieser Prozess hat zuerst eine UND-Verzweigung mit zwei Pfaden und darin geschachtelt jeweils eine ODER-Verzweigung. Abbildung 3-46 zeigt den Prozess, nachdem er im *Demonstrator* erstellt wurde. Die ODER-Verzweigung erschwert die Erkennung der umschließenden Parallelität, da nicht immer dieselben Schritte ausgeführt werden.

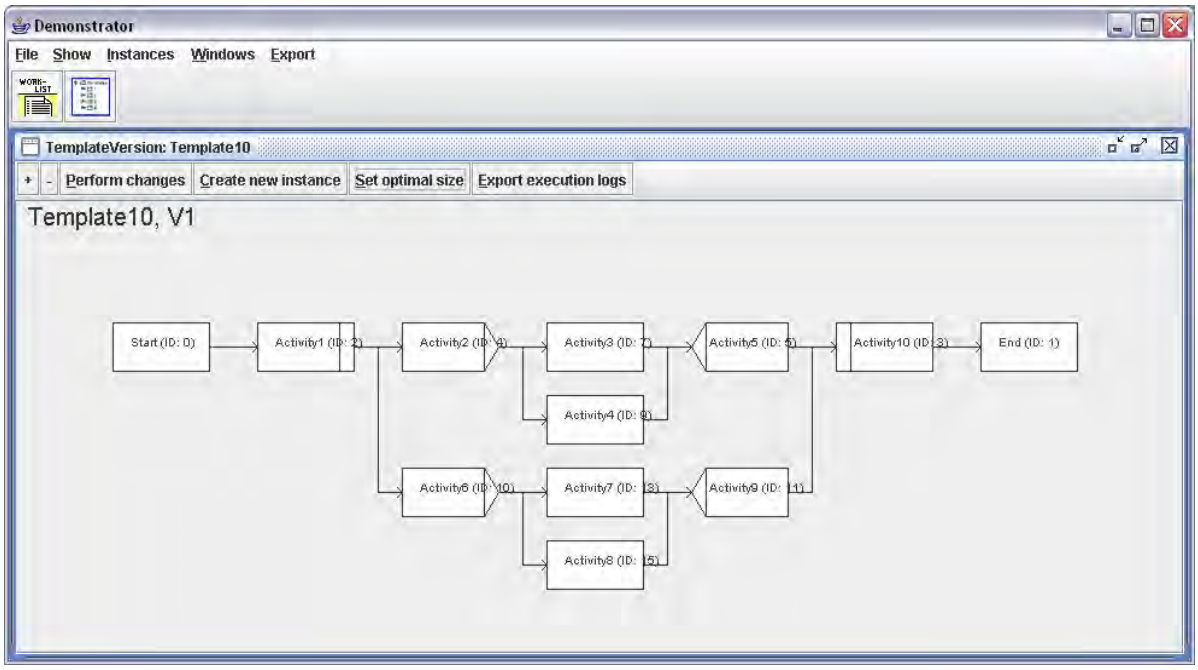


Abbildung 3-46: Beispielprozess: Template10, V1 (im Demonstrator)

Wie zu erwarten hat der *Alpha-Algorithmus* Probleme den Prozess bei 50 Instanzen korrekt zu rekonstruieren. Wie man in Abbildung 3-47 sieht, kann man eine gewisse Parallelität erkennen, doch die genauen Abhängigkeiten zwischen den einzelnen Schritten lassen sich nicht ermitteln. Bei diesem Prozess benötigt der Algorithmus 250 Instanzen um alle Beziehungen korrekt zu erfassen.

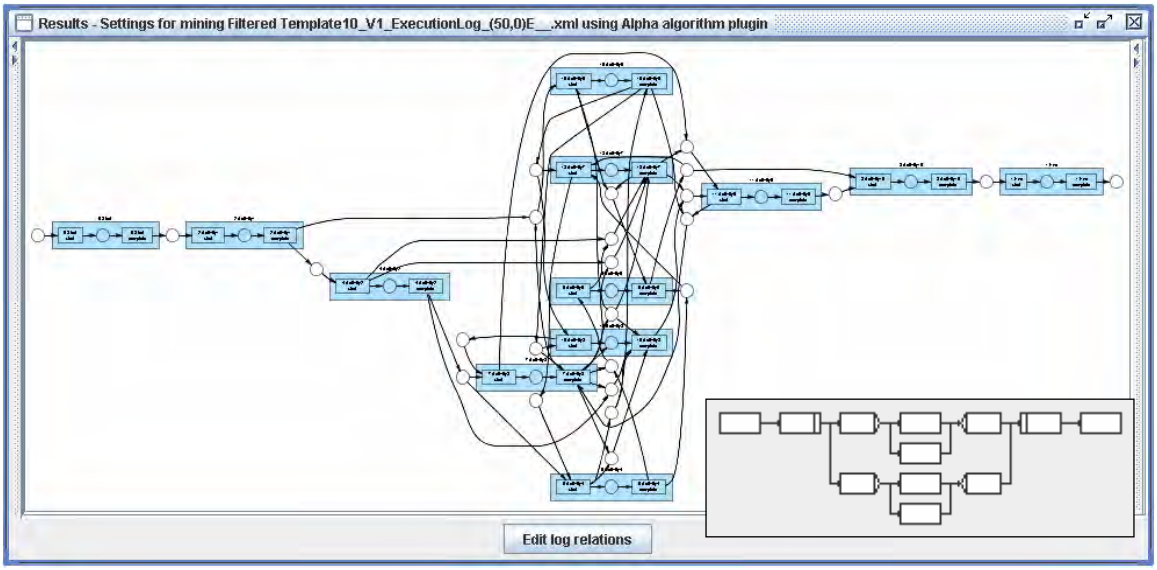


Abbildung 3-47: Beispiel Alpha-Algorithmus: Template10, V1, 50 Instanzen, keine Noise

Auch *Multi-Phase-Mining* benötigt 250 Instanzen um den Prozess komplett wieder herstellen zu können. Und im Gegensatz zum *Alpha-Algorithmus* kann bei 50 Instanzen die Grundstruk-

tur des Prozesses noch nicht erkannt werden, da sich die einzelnen Instanzen in ihrer Ausführungsreihenfolge zu stark unterscheiden. Dies sieht man wenn man sich einzelne Instanzen anzeigen lässt.

Der *Tsinghua-Alpha-Algorithmus* als Spezialist für parallele Beziehungen erzeugt bei 50 Instanzen bereits einen fehlerlosen Graphen, wie man in Abbildung 3-48 sehen kann.

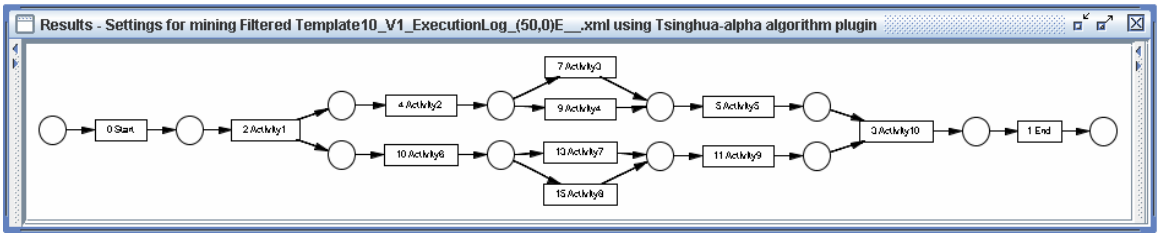


Abbildung 3-48: Beispiel Tsinghua-Alpha-Algorithmus: Template10, V1, 50 Instanzen, keine Noise

Der *Heuristics Miner* kann bereits bei 25 Instanzen diesen Prozess korrekt rekonstruieren, Abbildung 3-49 zeigt den Graph bei 50 Instanzen.

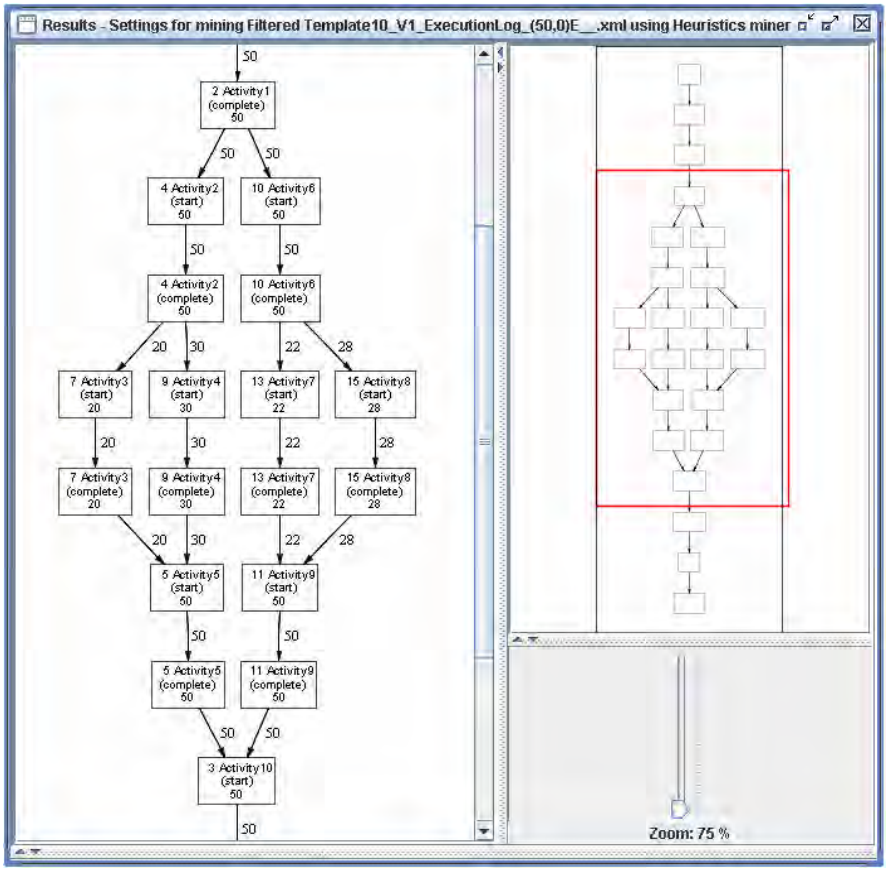


Abbildung 3-49: Beispiel Heuristics Miner: Template10, V1, 50 Instanzen, keine Noise

Der *Heuristics Miner* zeigt den Ergebnisgraph als heuristisches Netz an. In solch einem Netz wird graphisch nicht zwischen einer UND-Verzweigung und einer ODER-Verzweigung unterschieden, wie man in Abbildung 3-49 sehen kann. Nur die Häufigkeitszahlen auf den Kanten geben Auskunft darüber, ob es sich um eine UND- oder eine ODER-Verzweigung handelt. Bei einer UND-Verzweigung haben alle Kanten die gleiche Häufigkeit, bei einer ODER-Verzweigung haben die Kanten in der Summe zusammen die gleiche Häufigkeit wie der vorhergehende Knoten. Im Beispiel kommt zuerst eine UND-Verzweigung (je 50 Vorkommen) und danach je eine ODER-Verzweigung (20 & 30 Vorkommen bzw. 22 & 28 Vorkommen).

Bei diesem Prozess zeigen sich ebenfalls die Unterschiede zwischen den einzelnen Algorithmen. Wieder haben *Tsinghua-Alpha-Algorithmus* und *Heuristics Miner* einen deutlichen Vorteil gegenüber *Alpha-Algorithmus* und *Multi-Phase-Mining*, was die Anzahl der benötigten Instanzen betrifft. Hier muss erwähnt werden, dass alle Beispiel-Log-Daten ohne *Noise* waren, da hier einzig und allein auf die Komplexität der Prozesse geachtet wurde. Wie sich die einzelnen Algorithmen bei *Noise* verhalten wurde ja weiter oben schon erläutert.

Neben diesen drei Prozessen wurden noch weitere Prozess-Logs mit den Algorithmen rekonstruiert, jedoch waren bei den anderen Prozessen kaum Unterschiede zwischen den Algorithmen zu erkennen[12]. Interessant an dieser Stelle ist vielleicht, dass ein zu ‚*Template10, V1*' analoger Prozess (*Template9, V1*), der zuerst eine ODER-Verzweigung und dann verschachtelt je eine UND-Verzweigung hat von allen Algorithmen ohne Probleme mit nur 25-50 Instanzen erkannt werden konnte. Das lässt darauf schließen, dass sich die Komplexität an der Parallelität der Prozesse festmachen lässt. Je mehr parallele Pfade es in einem Prozess gibt, desto schwieriger wird es für die Algorithmen diese Pfade so zu erkennen, wie sie im ursprünglichen Prozess modelliert wurden.

3.4 Fazit

In diesem Kapitel werden die Eigenschaften der einzelnen Algorithmen noch mal kurz dargelegt. Es gibt mindestens drei Kriterien, die den *Process Mining* Prozess beeinflussen:

- Anzahl Instanzen: Die Anzahl der für das *Mining* zur Verfügung stehenden Instanzen hat einen maßgeblichen Anteil an der Qualität des Ergebnis-Graphen. Je mehr Daten vorhanden sind, desto leichter ist es für die Algorithmen den ursprünglichen Prozess wieder herzustellen. Deshalb ist ein Algorithmus umso besser, je weniger Daten er für die Gewinnung benötigt. In den Beispieldaten wurden zwischen 10 und 1000 Instanzen verwendet, um die Unterschiede zwischen den Algorithmen herauszuheben.
- Menge *Noise*: Je unvollständiger oder fehlerhafter die Log-Daten sind, desto schwieriger ist es für die Algorithmen den zugrunde liegenden Prozess zu erkennen. Für Algorithmen, die *Noise* behandeln können liegt das Hauptproblem in der Entscheidung, ob eine bestimmte Aufzeichnung im Log ein Fehler oder eine seltene Option ist. Je nachdem muss die Aufzeichnung dann berücksichtigt oder vernachlässigt werden, um den ursprünglichen Prozess korrekt rekonstruieren zu können. In den Beispielen gibt es unterschiedliche Arten von *Noise*. *Einfache Noise* zeichnet sich durch einzelne fehlende Ereignisse in den

[12] Alle verwendeten Prozesse sind im Anhang aufgeführt.

Log-Daten aus, während etwas realistischere *Noise* durch Einfügen, Verschieben und/oder Löschen von kompletten Prozessschritten erstellt wird. Die Menge an *Noise* kann beliebig variiert werden, verwendet wurden 5 %, 10 %, 15 % und 20 % (*einfache*) *Noise*.

- Prozess-Komplexität: Je mehr (konkurrierende) Schritte ein Prozess hat, umso schwieriger ist es für die Algorithmen, den Prozess wieder herzustellen. Ein Algorithmus ist gut, wenn er mit einer bestimmten Anzahl Instanzen einen komplexen Prozess erkennen kann, während ein anderer Algorithmus bei der gleichen Anzahl Instanzen noch Probleme hat. In den Beispielen wird Komplexität durch Synchronisations-Kanten und Parallelität erreicht. ODER-Verzweigungen haben scheinbar keinen größeren Einfluss auf die Komplexität.

Neben diesen drei Kriterien gibt es noch weitere, die aber im Rahmen der Diplomarbeit nicht gezielt verändert/untersucht wurden, z.B. die Unbalanciertheit bei Entscheidungen. So müssen während des Ablaufs eines Prozesses mehrere Entscheidungen getroffen werden (ODER-Verzweigung: welchen Pfad ausführen? UND-Verzweigung: welche Aktivität zuerst ausführen?). Je nachdem wie die Wahrscheinlichkeiten unter den Möglichkeiten verteilt sind, kann hier eine Unbalance vorliegen, also z.B. ein Pfad (viel) häufiger gewählt werden als ein anderer. Bei realen Prozessen kann die Unbalanciertheit durchaus einen größeren Einfluss auf die Prozessgewinnung haben, beim *Demonstrator* werden die Entscheidungen jedoch per Zufallsfunktion getroffen, so dass die einzelnen Möglichkeiten grundsätzlich gleichwertig und gleichwahrscheinlich sind.

Die einzelnen Kriterien hängen zusätzlich voneinander ab, so kann durch eine Erhöhung der Anzahl der Instanzen (also durch mehr Log-Daten) *Noise* besser rausgefiltert werden und komplexere Prozesse können einfacher rekonstruiert werden. Deshalb wurde für die Bewertung der Algorithmen bezüglich *Noise* und Komplexität die Anzahl der Instanzen bei den einzelnen Algorithmen konstant gehalten, bzw. als Maßzahl für die Bewertung verwendet. Tabelle 2 zeigt eine Übersicht über die Algorithmen und deren Eigenschaften bezüglich der vorgestellten Kriterien.

Tabelle 2: Übersicht Algorithmen

	Anzahl Instanzen	Menge Noise	Komplexität Prozess
Alpha-Algorithmus	O	O	-
Multi-Phase Mining	-	-	-
Tsinghua-Alpha	++	--	++
Heuristics Miner	+	++	+

Der *Alpha-Algorithmus* ist ein theoretisch fundierter Ansatz, für den genau gesagt werden kann, welche Klasse von Prozessen er rekonstruieren kann und welche nicht. Dazu müssen für eine korrekte Funktionsweise bestimmte Voraussetzungen erfüllt sein, d.h. die Log-Daten müssen u.a. vollständig und fehlerfrei sein. Sind diese Voraussetzungen erfüllt, kann der Algorithmus bei einem einfachen Prozess wie ‚*OP*-Vorbereitung' mit einer im Vergleich zu den anderen Algorithmen durchschnittlichen Anzahl Instanzen (25-50) den Prozess wieder herstellen. Sind die Log-Daten nicht fehlerfrei, also mit *Noise* versetzt, so kann bestenfalls die

Grundstruktur eines Prozesses erkannt werden, der Prozess kann aber nicht mehr korrekt rekonstruiert werden. Je mehr und unterschiedlicher die *Noise* in den Log-Daten ist, umso schlechter wird der Algorithmus damit fertig. Da in der Realität aber häufig geänderte und/oder fehlerhafte Log-Daten vorliegen, ist der Algorithmus für den Einsatz unter realen Bedingungen nur eingeschränkt geeignet. Dass die Log-Daten vollständig sein sollten, zeigt sich wenn die Prozesse komplexer werden, da hier schon deutlich mehr Instanzen benötigt werden, um ein korrektes Ergebnis zu liefern (250-500). Das liegt grundsätzlich daran, dass alle möglichen Ausführungsreihenfolgen auch im Log sein sollten, was in der Realität nicht immer möglich ist (z.B. bei 8 parallelen Schritten gibt es über 40000 verschiedene Ausführungsreihenfolgen). Der *Alpha-Algorithmus* hat seine Schwächen also bei fehlerhaften und unvollständigen Log-Daten, hier sind andere Algorithmen deutlich besser. Dafür ist genau spezifiziert und bewiesen, welche Art von Prozessen rekonstruiert werden können.

Das *Multi Phase Mining* ist grundsätzlich dazu gedacht, einzelne Prozessinstanzen zu visualisieren. Dazu wird ein Instanzgraph, basierend auf der Information des kompletten Datensatzes erstellt. Der Algorithmus hat im Gegensatz zum *Alpha-Algorithmus* weniger Voraussetzungen, so muss nur gegeben sein, dass die einzelnen Ereignisse im Log in einer kausalen Beziehung zueinander stehen, die Log-Daten müssen aber nicht vollständig sein. Als erster Algorithmus konnte *Multi Phase Mining* eine Ereignis-Prozess-Kette (EPK) als Ergebnis liefern, was für die Weiterverarbeitung im kommerziellen Tool ARIS PPM von Vorteil ist. Wenn nur eine individuelle Instanz betrachtet wird, spielt sowohl die Anzahl der Instanzen im Log, als auch die Menge an *Noise* und die Komplexität des Prozesses eine untergeordnete Rolle. Schon mit relativ wenigen Daten kann für die Instanz ein korrekter Prozess angezeigt werden, auch wenn die Log-Daten fehlerhaft sind oder der ursprüngliche Prozess komplexerer Natur ist. Das liegt auch daran, dass fehlerhafte Informationen besser herausgefiltert werden können als z.B. beim *Alpha-Algorithmus*. Sollen aber die einzelnen Instanzen zu einem Gesamtmodell zusammengefasst werden, offenbaren sich die Nachteile vom *Multi Phase Mining*. Denn jetzt zeigt sich, dass der Algorithmus mehr Instanzen benötigt als die anderen Algorithmen, um den Prozess korrekt zu rekonstruieren, was auch nur möglich ist, wenn die Log-Daten nicht fehlerhaft sind. Denn dann lässt sich nicht einmal mehr eine Grundstruktur erkennen. Ähnlich ist es bei komplexeren Prozessen, hier werden auch viele Instanzen (250-500) benötigt, um ein korrektes Ergebnis zu liefern, ansonsten ist schon das Erkennen einer Grundstruktur schwierig.

Der *Tsinghua-Alpha-Algorithmus* ist eine Variante des *Alpha-Algorithmus*, der die Ereignistypen *start* und *complete* für die explizite Erkennung paralleler Abhängigkeiten verwendet. Der Algorithmus kommt am besten mit einer geringen Anzahl Instanzen und komplexen Prozessen zurecht, weil hier die gute Erkennung paralleler Schritte besonders hilfreich ist. Ein weiterer Vorteil ist die graphische Zusammenfassung der einzelnen Ereignisse einer Aktivität zu einem Knoten, denn so lässt sich das Ergebnis besonders gut mit dem Ausgangsprozess vergleichen, weil auch da für eine Aktivität meist nur ein Knoten angezeigt wird. Der Algorithmus liefert also bereits bei wenigen Log-Daten ein gutes Ergebnis, auch bei komplexeren Prozessen. Eine große Schwäche zeigt er allerdings bei *einfacher Noise*, denn dadurch, dass er konsistente Daten (zu jedem Start-Ereignis muss ein End-Ereignis im Log sein und umgekehrt) erwartet, und die in diesem Fall nicht gegeben sind, kann der Algorithmus nicht korrekt funktionieren. Das Ergebnis kann in solch einem Fall nicht gebraucht werden, da es keinerlei

Aussagen über den zugrunde liegenden Prozess macht. Bei *Noise* aus geänderten Instanzen sieht das Ergebnis wieder besser aus, da hier die Log-Daten meist konsistent sind, also komplette Aktivitäten gelöscht, verschoben oder eingefügt wurden. Hier verhält sich der *Tsinghua-Alpha-Algorithmus* dann ähnlich wie der *Alpha-Algorithmus* und *Multi Phase Mining*, einzelne einfache Änderungen werden erkannt, bei mehreren kombinierten Änderungen wird es schwieriger und es lässt sich oft nur noch die Grundstruktur des Prozesses erkennen.

Der *Heuristics Miner* ist speziell daraufhin entwickelt worden, mit fehlerhaften und unvollständigen Log-Daten zurecht zu kommen. Anhand von Maßzahlen wird entschieden, in welcher Beziehung zwei Aktivitäten zueinander stehen. Der Algorithmus kommt bei einfachen Prozessen mit relativ wenigen Instanzen aus (25-50), um den Prozess korrekt zu rekonstruieren. Ihm reicht dieselbe Anzahl Instanzen aber auch aus, wenn die Log-Daten fehlerhaft sind, also *Noise* beinhalten. *Einfache Noise* ist für den Algorithmus gar kein Problem, da hier nur einzelne Ereignisse im Log fehlen, was bei der Berechnung von Maßzahlen nicht weiter von Bedeutung ist. Bei *Noise* aus geänderten Instanzen ist der Algorithmus am besten von allen vorgestellten Algorithmen, auch wenn nicht mehr alle Prozesse korrekt wieder hergestellt werden können (selbst wenn die Parameter für das *Mining* verändert werden). Das liegt hauptsächlich am Einfügen neuer Aktivitäten bei einzelnen Instanzen, da das nicht unbedingt als Fehler zu betrachten ist, sondern mehr als Ausnahmefall im Prozess. Eingefügte Schritte nehmen also dem Ergebnis die Korrektheit, sie sollten aber auch nicht unterschlagen werden, da sie ja seltene Optionen im Prozess darstellen. Sind solche Einfügungen öfter als ca. 5 % im Log vorhanden, werden sie deutlich als optionale Schritte wahrgenommen und auch die Einbindung in den Prozess wird besser, so ist eine gewisse Grundstruktur im Graphen meist zu erkennen. Bei komplexeren Prozessen ist der *Heuristics Miner* durch die Berechnung von Maßzahlen besser als der *Alpha-Algorithmus* und *Multi Phase Mining*, es reichen ca. 100-250 Instanzen aus, um den Prozess korrekt rekonstruieren zu können. Das Ergebnis wird als heuristisches Netz angezeigt, dazu wird bei den einzelnen Kanten und Knoten die Häufigkeit im Log annotiert. Zwischen einer UND- und einer ODER-Verzweigung wird bei einem heuristischen Netz nicht unterschieden, um den Typ einer Verzweigung zu erkennen müssen die Häufigkeitszahlen betrachtet werden.

Im Kontext dieser Arbeit wurden Schleifenkonstrukte in Prozessen nicht betrachtet. Sie spielen zwar in realen Prozessen eine Rolle, allerdings können die für einige Algorithmen (z.B. *Alpha-Algorithmus* in seiner ursprünglichen Form) kritischen kurzen Schleifen (ein- und zweielementige Schleifen) beim *ADEPT*-Metamodell durch die Blockstrukturierung erst gar nicht vorkommen, da jede Schleife eine Start- und eine End-Aktivität hat und mindestens eine Aktivität im Schleifenkörper vorkommt. Somit enthalten Log-Daten die auf *ADEPT*-Prozessen basieren grundsätzlich genügend Informationen um Schleifen korrekt zu erkennen. Weil der *Demonstrator* nicht das komplette *ADEPT*-Metamodell unterstützt, können mit ihm keine Schleifen modelliert werden und deshalb das Verhalten der Algorithmen bei Schleifen auch nicht getestet werden. Denkbar ist ein ähnliches Verhalten wie bei fehlerhaften Log-Daten, d.h. wenn die Schleifen oft genug ausgeführt wurden und somit häufig im Log auftauchen wird es für die Algorithmen grundsätzlich kein Problem sein die Schleifen korrekt zu erkennen. Werden Schleifen nur selten ausgeführt und erscheinen dementsprechend weniger häufig in den Log-Daten, könnte es sein, dass die Algorithmen Probleme mit der Erkennung hätten und Graphen mit fehlerhaften Beziehungen als Ergebnis zurückliefern.

3.5 Weitere Algorithmen und Möglichkeiten in ProM

ProM bietet neben den vorgestellten Möglichkeiten noch eine Reihe weiterer Verfahren an, um die gesammelten Log-Daten zu verarbeiten. Diese werden nachfolgend kurz vorgestellt und dargelegt, warum sie im Zusammenhang mit *ADEPT* nicht von Bedeutung sind.

3.5.1 Weitere Algorithmen

3.5.1.1 Genetic Algorithmus

Dieser Algorithmus [MWA04, MWA05, AMW05, *ProM*-Hilfe] nutzt genetische Algorithmen um Prozessmodelle zu gewinnen. Genetische Algorithmen sind lernfähige Suchmethoden, die versuchen die optimale Lösung für ein gegebenes Problem zu finden, basierend auf den genetischen Prozessen der Evolution innerhalb einer Bevölkerung. Der *Genetic Algorithmus* kann per Definition mit komplexen Konstrukten und *Noise* in den Log-Daten umgehen. Abbildung 3-50 zeigt den Ergebnisgraph bei 50 Instanzen des Beispielprozesses ‚*OP-Vorbereitung, V1*' und 5 *% einfacher Noise*.

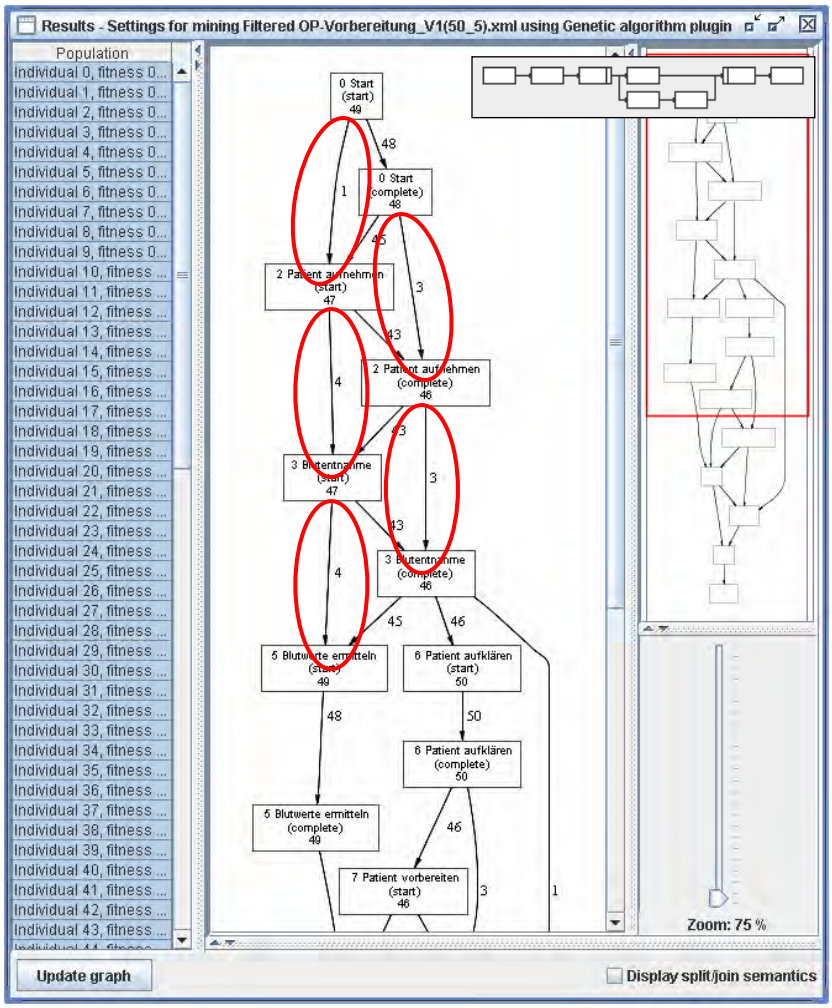


Abbildung 3-50: Beispiel Genetic Algorithmus: OP-Vorbereitung, V1, 50 Instanzen, 5 *% einfache Noise*

Das Ergebnis wird als heuristisches Netz (siehe Kapitel 3.3.4) angezeigt, d.h. u.a. werden die Häufigkeitszahlen der einzelnen Knoten und Kanten annotiert. Wie man dem Graph entnehmen kann, werden die fehlerhaft im Log aufgezeichneten Beziehungen (≙ Kanten mit niedrigen Häufigkeitszahlen) zwischen den einzelnen Ereignissen nicht ausgeblendet, so wie das beim *Heuristics Miner* der Fall ist (vgl. Abbildung 3-30). Man erhält also ein Ergebnis, dass dem ursprünglichen Prozess (oben rechts im Bild) nicht entspricht und auch die Grundstruktur ist schon schwieriger zu erkennen, wenn auch noch vorhanden. Dazu kommt noch, dass der *Genetic Algorithmus* nicht besonders performant ist. Für die Erzeugung des Beispielgraphen (bei 50 Instanzen und Standardparametern) benötigt der *Genetic Algorithmus* 20 Minuten[13], während der *Heuristics Miner* beim selben Datensatz nur 1-2 Sekunden für die Erzeugung eines Graphen benötigt. Der unübersichtliche Ergebnisprozess und die schlechte Performanz machen den *Genetic Algorithmus* dadurch uninteressant für das *Mining* der mit dem *Demonstrator* erstellten Log-Daten.

3.5.1.2 Case Data Extraction

Das *Case Data Extraction* Plug-in für *ProM* betrachtet die Daten der einzelnen Instanzen in den Log-Daten. Es konvertiert die Daten in eine kommaseparierte (*comma separeted values / csv*) Datei. Solch eine Datei kann z.B. in Microsoft Excel importiert werden. Dies ist hauptsächlich dazu gedacht, die Daten, die ein Prozess mitprotokolliert hat, von den Graphinformationen zu trennen und gegebenenfalls anderweitig weiterzuverarbeiten. Eventuell kann man dieses Plug-in in Zukunft zusammen mit anderen Algorithmen verwenden, um noch mehr Informationen über Prozesse zu gewinnen.

3.5.1.3 Process Instance Inspector

Der *Process Instance Inspector* zeigt für einzelne Instanzen lediglich die serielle Aneinanderreihung der jeweiligen Ereignisse. Zu den einzelnen Ereignissen werden die im Log vorhandenen Zusatzinformationen (z.B. gemessene Blutwerte) angezeigt, wie Abbildung 3-51 zeigt. Eine Bearbeitung der Instanzen oder irgendeine Form von *Mining* findet nicht statt, auch eine Zusammenfassung der Instanzen ist nicht möglich. Denkbar wäre dieses Plug-in zukünftig im Zusammenhang mit anderen Algorithmen zu verwenden, um die Daten einzelner Instanzen zu vergleichen. Dadurch könnten z.B. Gründe für bestimmte Änderungen an einzelnen Instanzen gefunden werden.

[13] Auf einem 2,0 GHz Intel Centrino Notebook mit 1024 MB Hauptspeicher und Windows XP Professional.

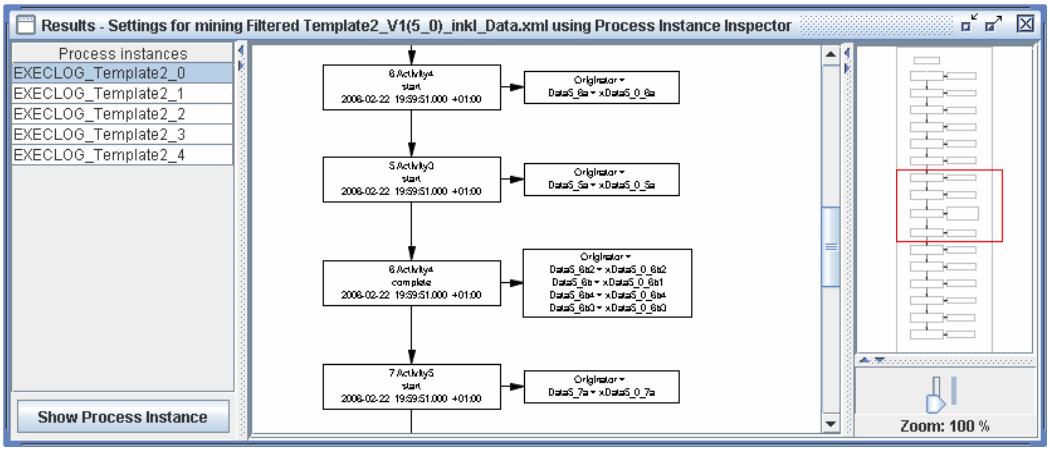


Abbildung 3-51: Beispiel Process Instance Inspector: Template2, V1, inkl. Data, 5 Instanzen (davon Instanz 0 ausgewählt), keine Noise

3.5.1.4 Social Network Miner

Der *Social Network Miner* [AaSo04] dient dazu, Rollen und andere Organisationsgebilde aus den Ereignis-Logs abzuleiten, also die Mitarbeit und Zusammenarbeit der einzelnen Bearbeiter zu erfassen. Einerseits kann die Beziehung zwischen Personen (oder Gruppen von Personen) und dem Prozess betrachtet werden, z.B. welcher Bearbeiter für welche Aktivität zuständig ist. Andererseits können die Beziehungen der Personen untereinander untersucht werden, um z.B. zu erkennen, welche Personen zusammen arbeiten und welche nicht.

3.5.1.5 Duplicate Tasks GA

Das *Duplicate Tasks GA* Plug-in ist eine Version des *Genetic Algorithmus* die für doppelte Schritte geeignet ist. Doppelte Schritte treten auf, wenn mehre Schritte im Prozessmodell denselben Bezeichner haben. Außer in der Erkennung von doppelten Schritten unterscheidet sich dieses Plug-in nicht vom *Genetic Algorithmus* (siehe Kapitel 3.5.1.1), es ist insbesondere genauso wenig performant.

3.5.1.6 DWS Mining

Das *Disjunctive Workflow Schema (DWS) Mining* Plug-in ist in der Lage eine Gruppe von Workflowmodellen zu erkennen, die verschiedene Untergruppen der gegebenen Log-Daten darstellen, und diese in einen durchsuchbaren Baum einzuordnen. Das heisst, sämtliche Instanzen einer Log-Datei werden in disjunkte Klassen eingeteilt, die in einer Baumstruktur dargestellt werden. Dieser Algorithmus benutzt den *Heuristics Miner*, um immer mehr Instanzen in die einzelnen Klassen abzuspalten, bis schließlich jede Instanz einer Klasse zugeteilt worden ist, und auch nur noch gleichartige Instanzen zusammen sind. Wenn der *Heuristics Miner* bei der gegebenen Anzahl Instanzen den ursprünglichen Prozess erkennen kann, werden die Instanzen alle einer Klasse zugeordnet, da sie ja auch alle vom selben Prozessschema abstammen. Sind die Log-Daten fehlerhaft oder nicht ausreichend, dann werden die einzelnen Instanzen unterschiedlichen Klassen zugeordnet, jedoch kann diese Klassenzuordnung im Zusammenhand mit *ADEPT* und dieser Diplomarbeit nicht weiter verwendet werden, weshalb dieser Algorithmus nicht näher betrachtet wird.

3.5.1.7 Alpha++ Algorithmus

Der *Alpha++ Algorithmus* [WWS06] ist eine Version des *Alpha-Algorithmus*, die speziell dazu weiterentwickelt wurde, implizite Abhängigkeiten zu erkennen. Implizite Abhängigkeiten sind indirekte Abhängigkeiten, die im Log nicht vorkommen können, weil sie das Ausführungsverhalten eines Prozesses nicht verändern. Abbildung 3-52 zeigt z.B. eine implizite Abhängigkeit in einem Petrinetz. Transition *B* hängt implizit (über die Stelle P_1) von Transition *A* ab, d.h. *B* kann nur ausgeführt werden, wenn zuvor *A* ausgeführt wurde. Da aber zwischen *A* und *B* noch mindestens ein Schritt *C*[14] ausgeführt wird, kann *B* im Log nie direkt auf *A* folgen. Die implizite Abhängigkeit kann also nicht erkannt werden.

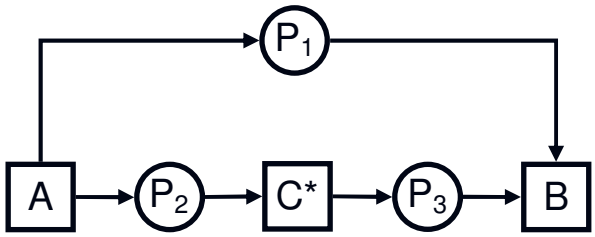


Abbildung 3-52: Implizite Abhängigkeit in einem Petrinetz [WWS06]

Im Zusammenhang mit impliziten Abhängigkeiten gibt es so genannte *Non-Free-Choice*-Konstrukte (siehe Abbildung 3-53). Auch solche Konstrukte enthalten implizite Abhängigkeiten. Bei Stelle P_1 entscheidet sich, ob T_1 oder T_2 ausgeführt wird. Je nachdem was gewählt wurde, wird später T_4 bzw. T_5 ausgeführt, d.h. für die Stelle P_5, dass sie nicht mehr die freie Wahl ($\triangleq$ *non-free-choice*) hat, welche Transition schalten soll, weil diese Entscheidung schon früher getroffen wurde (bei P_1). Die (impliziten) Beziehungen die zwischen T_1 und T_4 bzw. T_2 und T_5 können nicht erkannt werden, da im Log immer T_3 dazwischen ausgeführt wird, sie also nie direkt aufeinander folgen können.

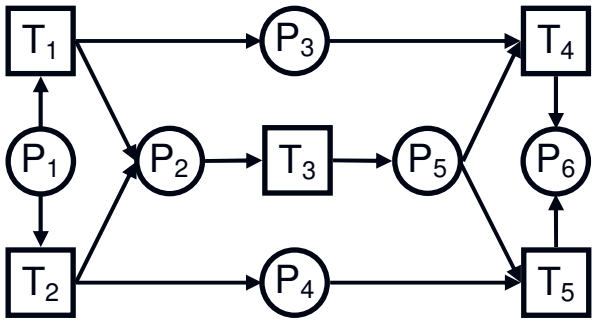


Abbildung 3-53: Beispiel eines *Non-free-choice*-Konstrukts [WWS06]

Der *Alpha++ Algorithmus* kann implizite Abhängigkeiten und *Non-Free-Choice*-Konstrukte erkennen, weil in *ADEPT*-Prozessen aber keine impliziten Abhängigkeiten vorkommen kön-

[14] Es können auch mehrere Schritte bzw. ein Subnetz zwischen *A* und *B* ausgeführt werden, an der impliziten Abhängigkeit und deren Erkennbarkeit ändert das nichts.

nen (u.a. durch die Blockstruktur) und sich die Ergebnisse deshalb nicht vom normalen *Alpha-Algorithmus* entscheiden, ist eine genaue Betrachtung dieses Ansatzes uninteressant.

3.5.1.8 Workflow Patterns Miner

Der *Workflow Patterns Miner*[15] [GBG05] erkennt Workflow Pattern in einem Prozess. Als Eingabe wird ein fehlerfreier Log-Datensatz benötigt, um eine korrekte Erkennung der Patterns zu ermöglichen. Ansonsten kann es sein, dass viele seltene Pattern wie z.B. ‚1-out-of-3' vom Algorithmus verwendet werden müssen, um den Prozess zu beschreiben, obwohl solche Pattern im ursprünglichen Prozess gar nicht verwendet wurden. Da die vom *Demonstrator* verwendeten Patterns auch so zu erkennen sind, und Workflow Pattern nicht Thema dieser Arbeit sind, ist dieses Plug-in nicht näher von Bedeutung.

3.5.2 Weitere Möglichkeiten in ProM

In *ProM* besteht (seit Version 3) auch bei anderen Algorithmen als dem *Multi-Phase Mining* die Möglichkeit, ein Petrinetz als eine Ereignis-Prozess-Kette darzustellen und umgekehrt. Auch ein heuristisches Netz kann als Petrinetz oder als EPK dargestellt werden. Dazu muss während ein bestimmter Prozess angezeigt wird der Menüeintrag *Conversion* angeklickt und die entsprechende Option ausgewählt werden. Abbildung 3-54 zeigt ein heuristisches Netz (links), sowie die Konvertierungen in eine EPK (rechts) und in ein Petrinetz (unten). Der ursprüngliche Prozess ist oben rechts im Bild eingeblendet. Darüber hinaus besteht auch die Möglichkeit, ein Petrinetz als ein YAWL-Modell anzuzeigen, wobei der Hauptunterschied darin besteht, dass die Stellen (*places*) aus dem Petrinetz entfernt werden.

Neben dem Gewinnen der Prozesse aus den Log-Daten gibt es noch die Möglichkeit die gewonnenen oder eingelesenen Prozesse zu analysieren, z.B. Petrinetzanalyse, Fitnessberechnung bei heuristischen Netzen. Oder die Prozesse in anderer Form zu exportieren (z.B. DOT-Datei, gruppierte XML-Datei, ARIS Graphformat) und sie auch wieder zu importieren.

Man kann *ProM* auch mit eigenen Plug-ins erweitern (siehe auch Kapitel 2.4), so wäre z.B. die Konvertierung eines Petrinetzes in einen *ADEPT*-Graphen eine denkbare Erweiterung. Die Herausforderung dabei ist das Erkennen einer Blockstruktur in einem Petrinetz bzw. die Erkennung von Synchronisations-Kanten, denn diese verbinden ja verschiedene Blöcke.

[15] Näheres zu Workflow Pattern unter www.workflowpatterns.com

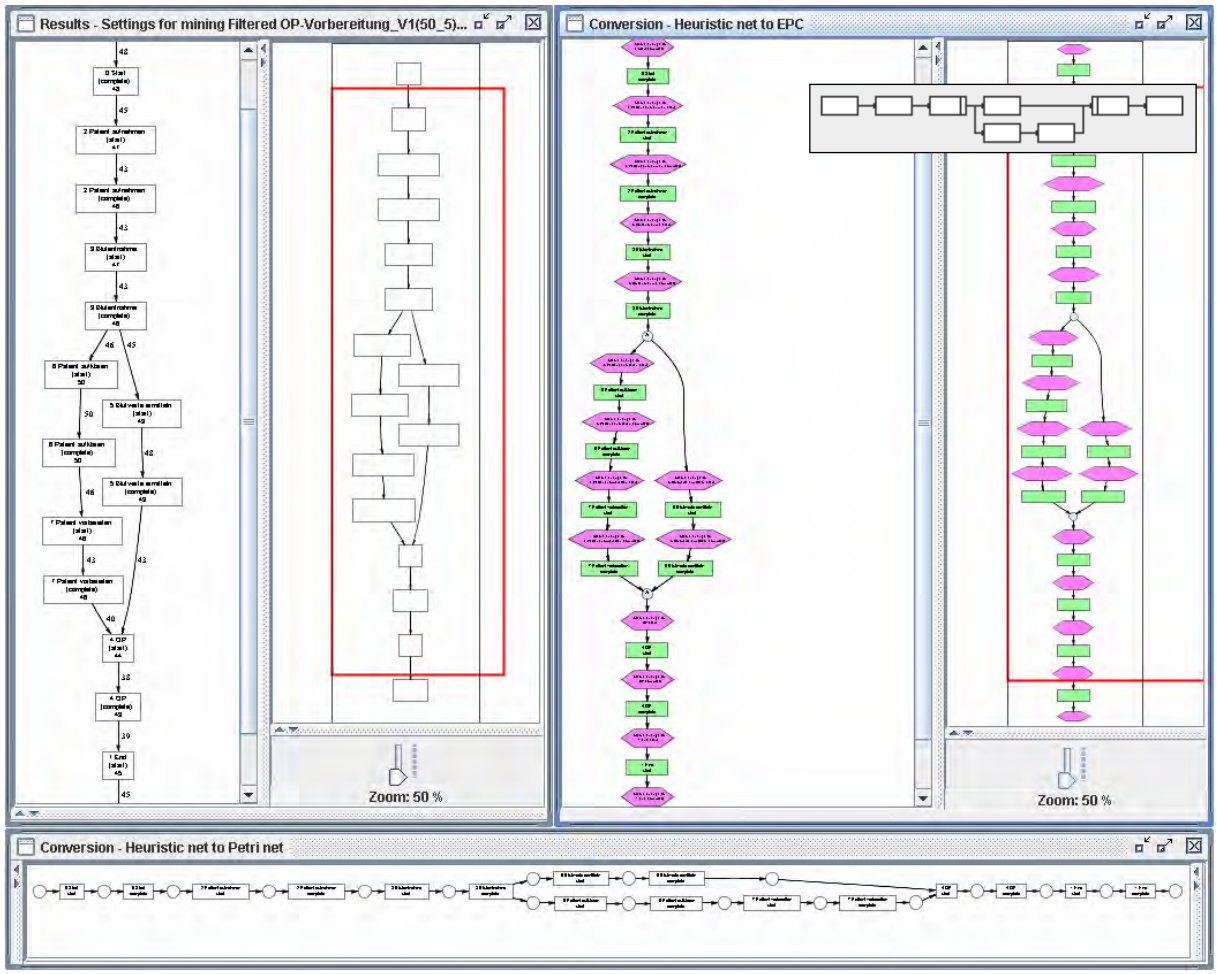


Abbildung 3-54: Konvertierung von Prozessgraphen in *ProM*

4 Rahmenwerk zum Mining von Änderungs-Logs

Nachdem nun ausführlich betrachtet wurde, worauf sich *Process Mining* Techniken bisher konzentrieren (nämlich *Mining* von Ausführungs-Logs), sollen im Folgenden kurz die Eigenschaften adaptiver Prozess-Management-Systeme (PMS) dargelegt werden bevor dann ein Rahmenwerk für die Integration von adaptiven PMS und *Process Mining* vorgestellt wird.

4.1 Adaptives Prozess-Management-System

Adaptives Prozessmanagement erweitert die Flexibilität prozessorientierter Informationssysteme durch die Ermöglichung (dynamischer) Änderungen verschiedener Prozessaspekte (z.B. Kontroll- und Datenfluss). Solche Prozessänderungen sind auf zwei Ebenen möglich, auf Vorlagen- und Instanzebene [RRD04a, RRD04b]. Instanzspezifische Ad-Hoc-Änderungen (z.B. Einfügen oder Löschen eines Prozessschritts) erlauben z.B. die Anpassung einzelner Prozessinstanzen an außergewöhnliche oder geänderte Situationen [ReDa98]. Änderungen auf Vorlagenebene ermöglichen die Anpassung an sich ändernde Rahmenbedingungen (z.B. neue Gesetze). Besonders bei langlaufenden Prozessen ist es deshalb wichtig, dass die Änderungen an einem Schema auch auf bereits laufende Instanzen propagiert werden können [RRD04c]. Das muss auf eine korrekte und konsistente Art und Weise geschehen und entsprechende Migrationsmethoden müssen eine Vielzahl gleichzeitig laufender Prozessinstanzen effizient bewältigen können. Dazu muss es möglich sein, die Änderungen sowohl auf unveränderte als auch auf (individuell) geänderte Instanzen zu propagieren. Abbildung 4-1 zeigt wie solch eine Vorlagen-Änderung (von Schema S auf Schema S') auf (teilweise) veränderte Instanzen (I1, I2) propagiert wird.

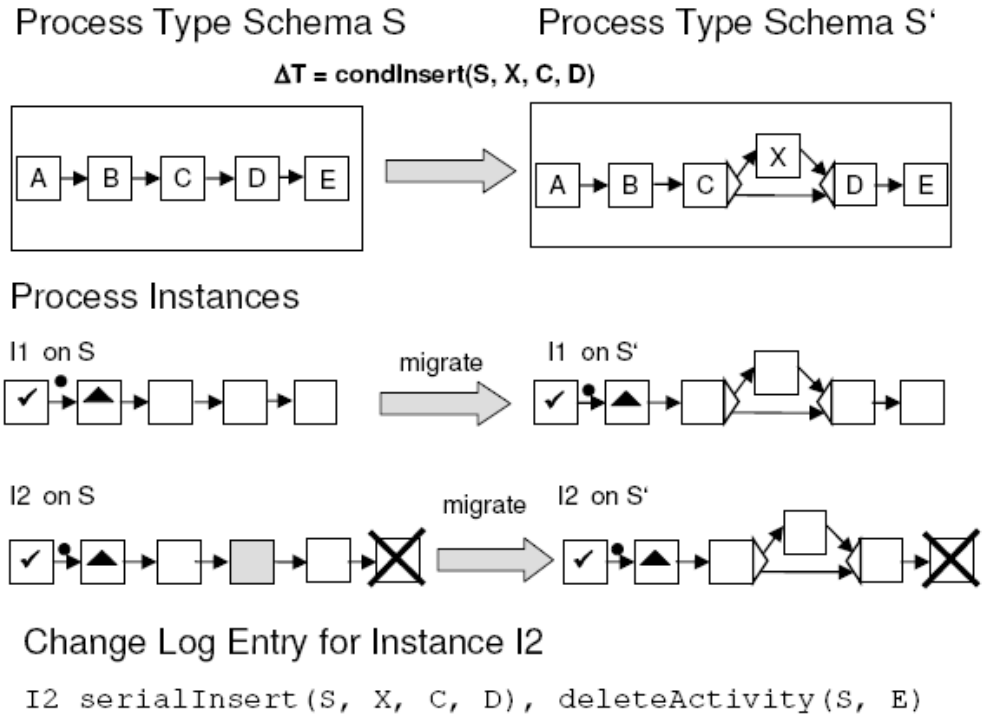


Abbildung 4-1: Adaptives Prozess-Management [WRRW05]

Alle (zur Laufzeit) stattfindenden Änderungen werden typischerweise in Änderungs-Logs protokolliert (siehe [RRJK06]), die Informationen über den Typ der angewandten Änderung und deren Kontext bereithalten. Diese Log-Information tragen sowohl zur Analyse potentieller Konflikte zwischen Prozesstyp- und Prozessinstanzänderungen als auch zur Dokumentation der Instanzhistorie bei.

Adaptive Prozess-Management-Systeme (PMS) wie der *ADEPT Demonstrator* bieten sowohl auf Prozesstyp- als auch auf Prozessinstanzebene die gewünschte Flexibilität. Bisher sind adaptive PMS allerdings nicht der Frage nachgegangen, was man aus den zusätzlichen Informationen in Änderungs-Logs lernen kann und wie man daraus optimierte Prozessmodelle ableiten kann. *Process Mining* Techniken dagegen bieten aussichtsreiche Perspektiven um von Änderungen zu lernen, konzentrieren sich bisher aber auf die Analyse von reinen Ausführungs-Logs (siehe [AWM04], Kapitel 3). Deshalb wird im Folgenden ein Rahmenwerk für die Integration beider Technologien vorgestellt, das zur kompletten Unterstützung des Prozesslebenszyklus beiträgt.

4.2 Rahmenwerk

Sowohl *Process Mining* als auch adaptives Prozess-Management behandeln grundlegende Probleme die momentanen in der Praxis von *Business Process Management* Implementierungen weit verbreitet sind. Diese Probleme entstammen der Tatsache, dass Entwicklung, Ausführung und Analyse eines Geschäftsprozesses gemeinhin als unabhängige Phasen betrachtet (und implementiert) werden. Obwohl beide Technologien für sich gesehen nützlich sind, können sie ihr volles Potential erst erreichen, wenn sie eng zusammenarbeiten. Während *Process Mining* konkrete und verlässliche Informationen darüber liefern kann, wie Prozessmodelle geändert werden müssen, bieten adaptive PMS die Werkzeuge, um diese Änderungen sicher und in geeigneter Weise zu realisieren.

Weiterhin ist denkbar, *Process Mining* Techniken zu entwickeln, die in adaptiven PMS als Feedback Kreislauf integriert sind. Im Anschluss daran müssten die adaptiven PMS mit Funktionalität für die Auswertung dieser Feedbackinformationen ausgestattet werden. Das Rahmenwerk in Abbildung 4-2 illustriert, wie solch eine Integration ausschauen könnte.

Adaptive PMS (oben im Bild) arbeiten auf vordefinierten Prozessmodellen. Die Weiterentwicklung dieser Modelle über einen längeren Zeitraum erzeugt eine Reihe von Prozessänderungen, d.h. führt zu mehreren Prozessvarianten. Wie in jedem prozessorientierten Informationssystem werden Ausführungs-Logs erstellt, die den Ablauf der ausgeführten Aktivitäten für jede Instanz festhalten. Zusätzlich protokollieren adaptive PMS den Ablauf der Änderungsoperationen die für jede Instanz auf das Prozessmodell angewandt wurden, erzeugen somit eine Reihe von Änderungs-Logs. Die *Process Mining* Techniken die in solch ein System in Form eines Feedback Kreislaufs eingebaut werden, gehören einer der drei folgenden Kategorien an:

- Änderungs-Analyse: Diese Techniken nutzen die Information in Änderungs-Logs neben den ursprünglichen Prozessmodellen und ihrer Varianten. Ihr Ziel ist die Bestimmung gemeinsamer und häufiger Varianten für jedes Prozessmodell, welche die ursprünglichen Modelle ersetzen könnten.

- Integrierte Analyse: Diese Analyse nutzt Änderungs- und Ausführungs-Logs zusammen, z.B. für die Bestimmung von Gründen für eine Änderung aus den Kontextdaten der Ausführungs-Logs.
- Ausführungs-Analyse: Ausschließlich auf der Untersuchung von Ausführungs-Logs basierend können Techniken in dieser Kategorie Teile von Prozessmodellen lokalisieren, die geändert werden müssen. Wenn z.B. ein Pfad einer Verzweigung nie genutzt wird kann das zugrunde liegende Modell durch das Entfernen dieses Teils vereinfacht werden.

Diese Beispiele zeigen nur Richtungen auf, in die die Entwicklung passender *Process Mining* Techniken gehen könnte.

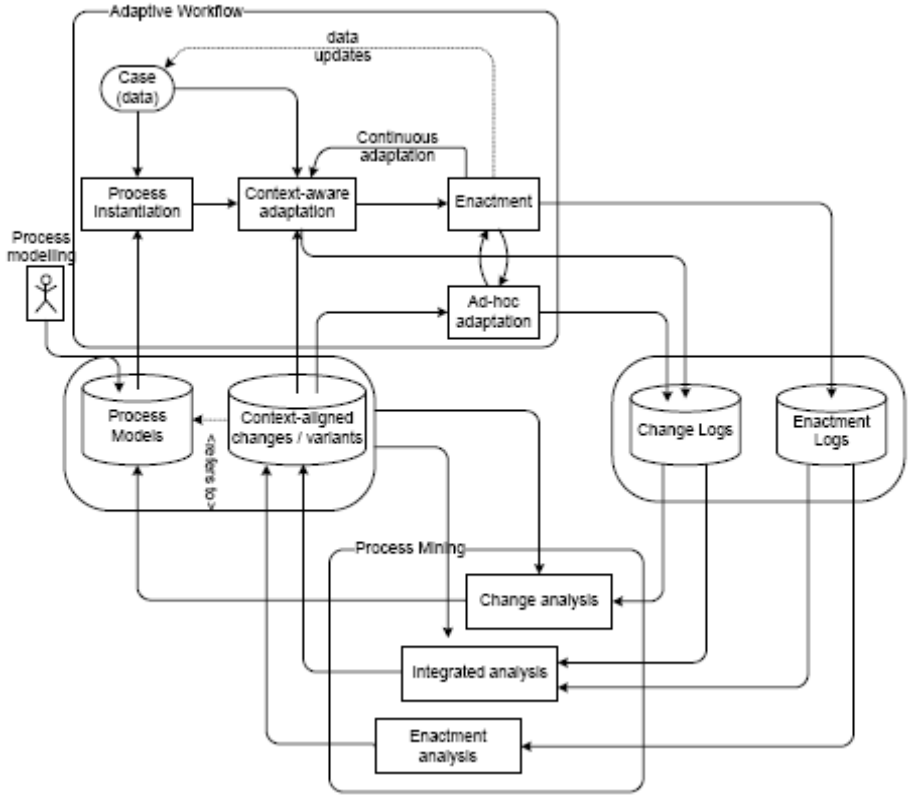


Abbildung 4-2: Integration von *Process Mining* und adaptivem Prozess-Management [GRRA06]

Wenn solch ein Feedback Kreislauf konsistent entworfen und realisiert wird, dann ist das resultierende System in der Lage dem Anwender Unterstützung und autonome Administration in noch nie da gewesener Art und Weise zu bieten. Außerdem liefert eine enge Integration von adaptiven PMS und *Process Mining* Technologien eine mächtige Grundlage, auf der eine neue Generation wirklich intelligenter und zunehmend autonomer prozessorientierter Informationssysteme aufgebaut werden kann.

5 Grundlagen für das Mining von Änderungs-Logs

Für ein besseres Grundverständnis bzgl. der Integration von *ADEPT* und *ProM* wurden in Kapitel 3 ausführlich die Erzeugung und die Auswertung von Ausführungs-Logs untersucht. Nun soll die Grundlage für eine völlig neue Art von *Mining* Verfahren geschaffen werden, nämlich dem *Mining* von Änderungs-Logs. Ziel hierbei ist es, aus bereits durchgeführten Änderungen zu lernen und somit den Benutzer zu unterstützen.
Die erste Herausforderung für das *Mining* von Änderungs-Logs liegt in der Erstellung von Log-Daten, denn um bestehende *Mining* Techniken untersuchen und neue Techniken entwickeln zu können wird eine größere Menge möglichst realitätsnaher Log-Daten benötigt. Bei adaptiven Prozess-Management-Systemen werden zusätzlich zu den Ereignissen während der Ausführung auch die Änderungen protokolliert, die stattgefunden haben. Weil diese Änderungen von Benutzern durchgeführt werden (z.B. wenn sich die Rahmenbedingungen eines Prozesses geändert haben) ist es sehr aufwändig, auf diese Weise eine größere Menge an Log-Daten zu erstellen. Deshalb muss die Erzeugung automatisiert werden. Dazu müssen alle Interaktionen mit dem Anwender und alle Benutzereingaben entfallen und trotzdem sinnvolle, möglichst realitätsnahe Log-Daten entstehen.

Eine Änderungsoperation wird durch ihren Typ, das Subjekt der Änderung (die geänderte Aktivität) und dem syntaktischen Kontext (Vorgänger, Nachfolger) charakterisiert. Somit besteht die zweite Herausforderung darin, diese Informationen für das *Mining* in geeigneter Art und Weise aufzubereiten, da *ProM* kein Format für Änderungs-Logs vorsieht, sondern nur für die Speicherung von Ausführungs-Logs (*MXML*).

Zunächst werden in diesem Kapitel die Ideen für Erstellung von Änderungs-Logs betrachtet, danach die technischen Grundlagen, d.h. die Erstellung und die Transformierung von Änderungs-Logs, bevor in Kapitel 6 die Konzepte zum *Mining* von Änderungs-Logs (*Change Mining*) vorgestellt werden.

5.1 Konzept für die Erzeugung von realitätsnahen Änderungs-Log-Daten

Für die Erzeugung der Log-Daten soll der Benutzer ähnlich wie bei der Erzeugung von Ausführungs-Logs nur wenige Parameter vorgeben. So soll er angeben können, wie viele Instanzen erzeugt werden sollen und wie hoch die Wahrscheinlichkeit für eine Änderung ist. Daneben soll noch bestimmt werden können, welche Art von Änderungen stattfinden soll, also Einfügen, Löschen, Verschieben oder Kombinationen davon. Sämtliche Entscheidungen, die während der Ausführung getroffen werden müssen, werden vom System übernommen. Auch für die Erstellung von Änderungs-Logs müssen die Aktivitäten der einzelnen Instanzen ausgeführt werden. Das geschieht analog zur automatischen Erzeugung von Ausführungs-Logs (siehe Kapitel 3.1). Neben der Ausführung müssen die Instanzen geändert werden, um so Änderungs-Logs zu erhalten. Für den Beispielprozess ‚*OP-Vorbereitung*' wird eine Reihe von Änderungen fest vorgegeben (siehe auch Kapitel 5.2.1), aus denen für jede Instanz bestimmte Änderungen ausgewählt werden, je nachdem wie wahrscheinlich die Änderungen sind und welche Art von Änderungen gewünscht ist. Um auch Änderungs-Logs für beliebige

Prozesse erstellen zu können, müssen einige Entscheidungen per Zufall getroffen werden, weil der Prozess nicht von vornherein bekannt ist. Eine Zufallsfunktion muss entscheiden, an welche Stelle im Prozess eine neue Aktivität eingefügt, welche Aktivität gelöscht oder welche Aktivität verschoben und an welche Stelle sie verschoben werden soll. Auch für Änderungen an beliebigen Prozessen gibt es eine Reihe fest vorgegebener logischer Änderungen (siehe Kapitel 5.2.2), aber der Zufall entscheidet bei den einzelnen Instanzen, welche Aktivitäten von den Änderungen betroffen sind. Wie beim Beispielprozess ‚*OP-Vorbereitung*' entscheidet der Benutzer, wie wahrscheinlich die Änderungen sind und welche Art von Änderungen erlaubt sind.

Die Erstellung von Änderungs-Logs ist eine Herausforderung, eine weitere ist es, die Informationen auf eine Art und Weise zu speichern, so dass sie für das *Mining* verwendet werden können. Für das Format von Änderungs-Logs gibt es zwei grundsätzliche Möglichkeiten: Einmal die Erweiterung von *ProM*, um Änderungs-Logs importieren zu können, oder ein Mapping von Änderungs-Logs auf Ausführungs-Logs. Der Vorteil eines Mappings ist der, dass auch die bestehenden *Mining* Verfahren für das *Mining* von Änderungs-Logs verwendet werden können, da sie in einer Form vorliegen, den die Algorithmen verstehen. Weil auch Änderungen einem Prozess und einer Instanz zugeordnet werden können, passt das Format für Ausführungs-Logs (*MXML*) auch für die Speicherung von Änderungs-Logs. Anstelle der Ausführungsreihenfolge werden die Änderungen chronologisch gespeichert. Doch wie sieht ein „gutes" Mapping aus? Da eine Änderungsoperation durch den Typ der Änderung, das Subjekt der Änderung (die geänderte Aktivität) und dem syntaktischen Kontext (Vorgänger, Nachfolger) der Änderung beschrieben wird, werden diese Informationen für die Auswertung benötigt. Weil das *MXML*-Format aber ursprünglich für Ausführungs-Logs gedacht war, gibt es diese Felder nicht. Diese Informationen können jedoch in den optionalen *Data* Elementen (siehe Kapitel 2.3.3) gespeichert werden. Die bestehenden Algorithmen können diese zusätzlichen Daten in den *Data* Elementen nicht nutzen, da sie nicht wissen, dass diese vorhanden sind. Deshalb wird in Kapitel 6.2 ein neuer Algorithmus vorgestellt, der diese Informationen nutzen kann.

5.2 Das Erstellen von Änderungs-Logs

Die Änderungs-Logs werden im *Demonstrator* analog zu den Ausführungs-Logs erstellt. Dazu muss entweder zuerst ein neues Template erstellt werden oder ein bereits vorhandenes Template in den *Demonstrator* geladen werden. Wenn man ein Template angezeigt bekommt, gibt es die Option „*Export change logs*", wie Abbildung 5-1 zeigt.

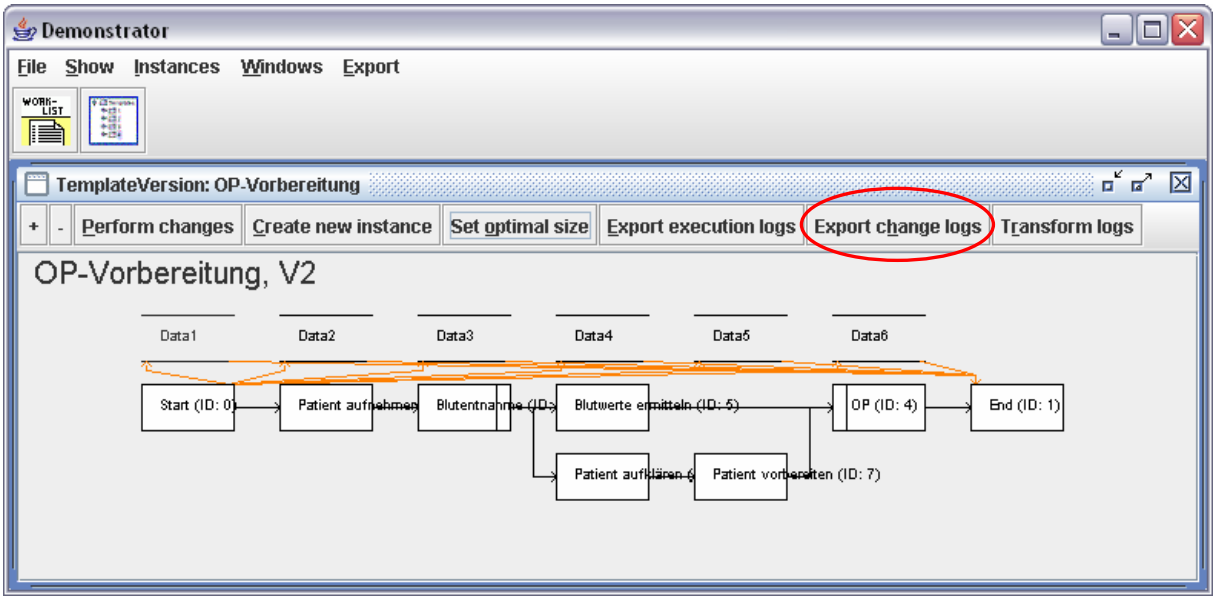


Abbildung 5-1: *Demonstrator* mit ausgewähltem Prozess ‚OP-Vorbereitung, V2'

Klickt man diese Schaltfläche an, erscheint ein Hilfsdialog, in dem angezeigt wird, welches Template ausgewählt ist (siehe Abbildung 5-2). Des Weiteren muss der Benutzer nun angeben, wie viele Instanzen erzeugt werden sollen (im Beispiel 25) und welche Änderungen vorgenommen werden sollen. Der Benutzer kann dazu entscheiden, ob das Einfügen, das Löschen oder das Verschieben von Aktivitäten oder Kombinationen davon erlaubt sind (im Beispiel sind alle Änderungen erlaubt) und wie wahrscheinlich die Änderungen sein sollen (im Beispiel 25 %). Mittels einer Checkbox kann der Benutzer schließlich angeben, ob die erzeugten Log-Dateien sofort in das *MXML*-Format umgewandelt werden sollen (Häkchen gesetzt), oder ob die Transformierung erst später stattfinden soll (Häkchen entfernt).

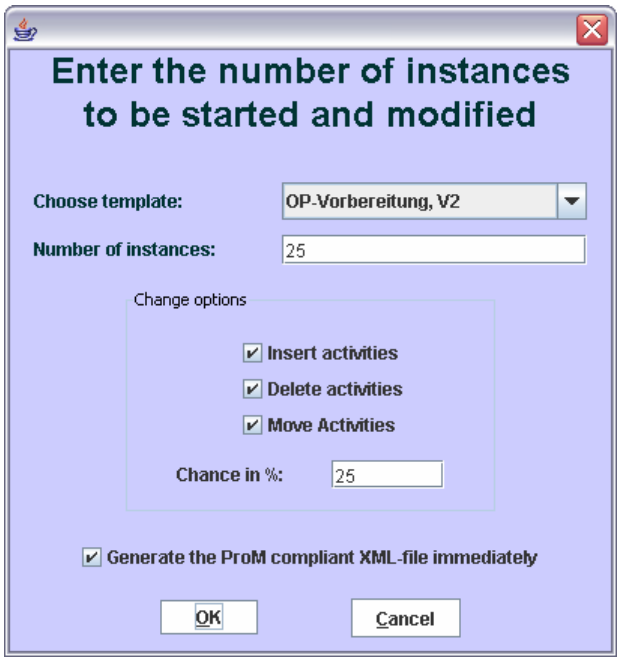


Abbildung 5-2: Hilfsdialog für die Erzeugung von Änderungs-Logs

Wenn der Benutzer schon weiß, welches bestehende Template er für die Log-Erzeugung verwenden will, kann er auch direkt über den Menüeintrag *Export → Export Change Logs* den Hilfsdialog (siehe Abbildung 5-2) aufrufen und dann in der Combobox das entsprechende Template auswählen.

Wenn alle Angaben gemacht sind und der Benutzer „*OK*“ angeklickt hat, werden im Hintergrund die angegebene Anzahl Instanzen erzeugt und ausgeführt und XML-Log-Dateien mit den entsprechenden Änderungen generiert. Analog zur Erstellung von Ausführungs-Logs werden die nötigen Entscheidungen durch eine Zufallsfunktion getroffen. Denn auch für die Erzeugung von Änderungs-Logs muss entschieden werden, welcher Eintrag von der Arbeitsliste als nächstes ausgeführt wird oder welcher Ausführungspfad beschritten werden soll. Die vom *Demonstrator* (als Zwischenergebnis) erzeugten Änderungs-Log-Dateien unterscheiden sich von Ausführungs-Log-Dateien nur durch die zusätzlichen Informationen über eventuelle Änderungen bei den einzelnen Instanzen. Gibt es bei einer Instanz keine Änderungen, so unterscheidet sich die Log-Datei nicht von einer Log-Datei, die während der Erstellung von Ausführungs-Logs erstellt wurde. Im Folgenden wird dargestellt, welche Änderungen es geben kann und wie entschieden wird, ob sie durchgeführt werden oder nicht.

5.2.1 Änderungen bei einem bereits bekannten Prozess (OP-Vorbereitung, V2)

Der bisher verwendete Beispielprozess ‚*OP-Vorbereitung, V1*’ wurde für die Erstellung von Änderungs-Logs um sechs Datencontainer (*Data1-Data6*) erweitert, die als Entscheidungsgrundlage für sechs verschiedene (logische) Änderungsoperationen dienen (siehe auch ‚*OP-Vorbereitung, V2*’ in Abbildung 5-1). Die Datencontainer enthalten jeweils einen zufälligen Datenwert in Form einer Prozentzahl (= Zahl zwischen 0 und 100), der darüber entscheidet, ob eine bestimmte Änderung stattfindet oder nicht. Nachfolgend ist aufgelistet, welcher Datenwert für welche Änderung steht:

- *Data1*: Löschen von ‚*Patient aufnehmen*', in 50 % der Fälle wird zusätzlich zur Löschung die Aktivität ‚*Patient entlassen*' eingefügt. Entspricht der Situation, dass ein Patient schon vorher aufgenommen wurde (z.B. im Rahmen einer anderen Operation). Das Löschen von Aktivitäten muss erlaubt sein, ggf. das Einfügen auch. Abbildung 5-3 illustriert diese Änderung.

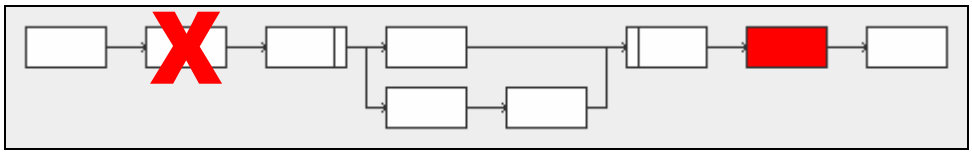


Abbildung 5-3: Änderung 1 am Prozess OP-Vorbereitung:
Löschen von ‚*Patient aufnehmen*', ggf. Einfügen von ‚*Patient entlassen*'

- *Data2*: Einfügen der Aktivitäten ‚*Erweiterte Bluttests*' (zwischen ‚*Blutwerte ermitteln*' und ‚*OP*') und ‚*Urintest*' (zwischen ‚*Blutentnahme*' und ‚*OP*'; parallel zu ‚*Blutwerte ermitteln*' und ‚*Patient aufklären*'). Entspricht einer Reihe von zusätzlichen Untersuchungen, die teilweise gleichzeitig stattfinden können, teilweise nacheinander ablaufen. Das Einfügen von Aktivitäten muss erlaubt sein. Abbildung 5-4 zeigt die Auswirkungen auf den Prozess.

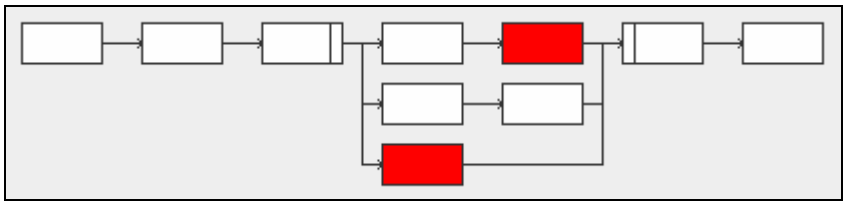


Abbildung 5-4: Änderung 2 am Prozess OP-Vorbereitung:
Einfügen von ‚*Erweiterte Bluttests*' und ‚*Urintest*'

- *Data3*: Einfügen der Aktivität ‚*Erweiterte Bluttests*' (falls nicht bereits durch Änderung 2 eingefügt) zwischen ‚*Blutwerte ermitteln*' und ‚*OP*' und anschließendes Löschen derselben. Entspricht der Situation, dass angeordnete Tests doch nicht mehr durchgeführt werden müssen. Das Einfügen und Löschen von Aktivitäten muss erlaubt sein. Abbildung 5-5 zeigt diese Änderung.

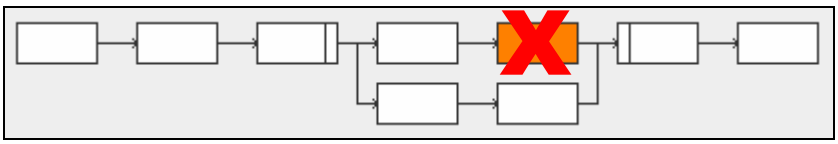


Abbildung 5-5: Änderung 3 am Prozess OP-Vorbereitung:
Einfügen und Löschen von ‚*Erweiterte Bluttests*'

- *Data4*: Verschieben der Aktivität ‚*Patient aufklären*' an die Stelle zwischen ‚*Patient vorbereiten*' und ‚*OP*'. Entspricht der Situation, dass der aufklärende Arzt kurzfristig verhindert ist und deshalb die Vorbereitung des Patienten vorgezogen wird. Das Verschieben von Aktivitäten muss erlaubt sein. Abbildung 5-6 verdeutlicht die Auswirkung auf den Prozess.

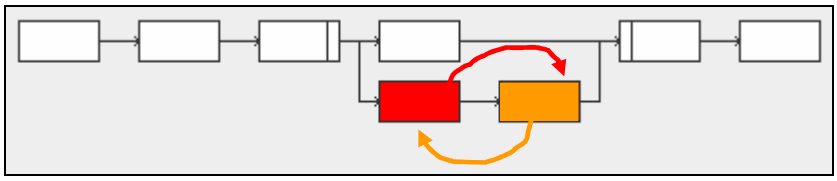


Abbildung 5-6: Änderung 4 am Prozess OP-Vorbereitung:
Verschieben von ‚*Patient aufklären*' (nach hinten)

- *Data5*: Einfügen der Aktivitäten ‚*Urintest*' (falls nicht bereits eingefügt) und ‚*Speicheltest*' zwischen ‚*Blutentnahme*' und ‚*OP*' (parallel zu ‚*Blutwerte ermitteln*' und ‚*Patient aufklären*'). Entspricht einer Reihe von zusätzlichen Untersuchungen, die zeitgleich zu anderen (bisherigen) ablaufen können. Das Einfügen von Aktivitäten muss erlaubt sein. Abbildung 5-7 illustriert die Änderung.

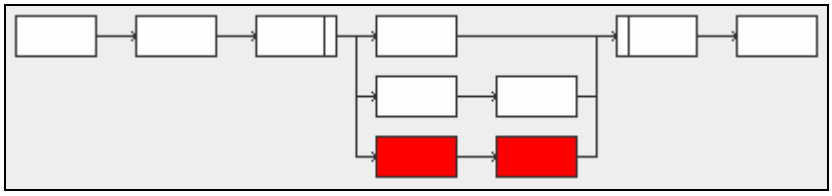


Abbildung 5-7: Änderung 5 am Prozess OP-Vorbereitung:
Einfügen von ‚*Urintest*' und ‚*Speicheltest*'

- *Data6*: Verschieben der Aktivität ‚*Patient aufklären*' an die Stelle vor ‚*Blutentnahme*'. Entspricht der Situation, dass der aufklärende Arzt kurzfristig früher Zeit für die Beratung hat, die eigentlich später eingeplant ist. Das Verschieben von Aktivitäten muss erlaubt sein. Abbildung 5-8 zeigt die Auswirkung auf den Prozess.

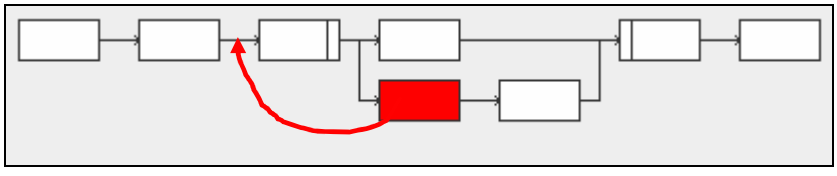


Abbildung 5-8: Änderung 6 am Prozess OP-Vorbereitung:
Verschieben von ‚*Patient aufklären*' (nach vorn)

Alle diese Änderungen können auch in Kombination miteinander stattfinden, je nachdem, welche Werte in den einzelnen Datencontainern enthalten sind. Die Änderungen finden alle

statt, nachdem die Start-Aktivität beendet wurde. Für das Löschen von Aktivität ‚*Patient aufnehmen*' ist das zwingend nötig, da laufende und beendete Aktivitäten nicht (mehr) geändert werden können. Die anderen Änderungen könnten auch später ablaufen, jedoch macht es für die Erzeugung von Änderungs-Logs keinen großen Unterschied, wann genau eine Änderung stattgefunden hat, da der zeitliche Ablauf keine solche Rolle spielt wie bei Ausführungs-Logs. In realen Prozessen würden die Änderungen sicher kurzfristiger ablaufen, da z.B. erst bei der Ermittlung der Blutwerte entschieden werden kann, ob auch erweiterte Bluttests stattfinden müssen, was aber an dieser Stelle nicht relevant ist.

Die Erzeugung der Log-Dateien läuft nun folgendermaßen ab. Zuerst wird die gewünschte Anzahl Instanzen erzeugt, danach werden die einzelnen Instanzen ausgeführt. Jeweils beim Beenden der Start-Aktivität werden die Datenwerte in die entsprechenden Datencontainer geschrieben. Gleich danach werden diese Werte wieder entnommen und mit dem vom Benutzer im Hilfsdialog (vgl. Abbildung 5-2) angegebenen Wahrscheinlichkeitswert verglichen. Je nachdem welche Änderungen erlaubt sind (Einfügen, Löschen, Verschieben), werden die Änderungen durchgeführt, falls der Zufallswert im Datencontainer niedriger ist, als der gegebene Wahrscheinlichkeitswert. Nachdem eine Instanz geändert wurde (oder auch nicht, bei ungünstigen Zufallszahlen oder zu niedrigem Wahrscheinlichkeitswert) werden die restlichen Aktivitäten ausgeführt. Wenn alle Instanzen abgearbeitet wurden, wird für jede Instanz eine XML-Log-Datei gespeichert.

5.2.2 Änderungen bei beliebigen Prozessen

Da bei beliebigen Prozessen nicht davon ausgegangen werden kann, dass sie Datencontainer als Entscheidungsgrundlage für die Änderungen bereithalten, müssen erst einmal Datencontainer in die Prozesse eingefügt werden. Dies geschieht, wenn sich die Start-Aktivität im Zustand ‚*running*' befindet, denn dann können mit dem Beenden der Start-Aktivität die Datenwerte geschrieben werden, genauso wie es beim Prozess ‚*OP-Vorbereitung, V2*' abläuft. Auch bei beliebigen Prozessen werden sechs Datencontainer eingefügt. Die Entscheidung für oder gegen eine Änderung läuft analog zum Prozess ‚*OP-Vorbereitung, V2*' ab, also anhand des Vergleichs Datenwerte und Wahrscheinlichkeitszahl, sowie den grundsätzlichen Angaben über erlaubte Änderungen. Nachfolgend ist wieder aufgelistet, für welche (logische) Änderung die einzelnen Datenwerte stehen:

- *Data1*: Einfügen einer neuen Aktivität ‚*New1*' an einer beliebigen Stelle im Prozess. Dazu wird per Zufallsfunktion eine Aktivität bestimmt, in deren Umgebung die neue Aktivität eingefügt werden soll. Dann wird der Vorgänger oder der Nachfolger der gewählten Aktivität bestimmt (wieder durch eine Zufallsfunktion). Somit ergibt sich der Zielbereich für die neu einzufügende Aktivität aus einer zufälligen Aktivität und einer weiteren Aktivität in deren Umgebung. Das Einfügen von Aktivitäten muss erlaubt sein. Abbildung 5-9 illustriert die möglichen Zielbereiche bei einem einfachen Prozess (der Übersichtlichkeit halber wird weiterhin der Prozess ‚*OP-Vorbereitung*' verwendet).

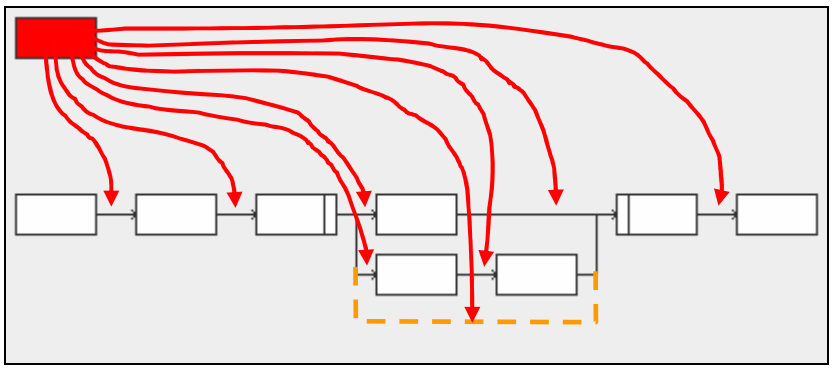


Abbildung 5-9: Änderung 1 an einem beliebigen Prozess:
Einfügen von ‚*New1*'

- *Data2*: Einfügen einer neuen Aktivität ‚*New2*' an einer beliebigen (per Zufallsfunktion bestimmten) Stelle im Prozess und einer weiteren Aktivität ‚*New3*' in der Umgebung von ‚*New2*'. Das Einfügen von Aktivitäten muss erlaubt sein. Abbildung 5-10 zeigt wieder die möglichen Zielbereiche für die beiden neuen Aktivitäten, die sowohl in der Reihenfolge ‚*New2*' ‚*New3*' als auch in der Reihenfolge ‚*New3*' ‚*New2*' eingefügt werden können.

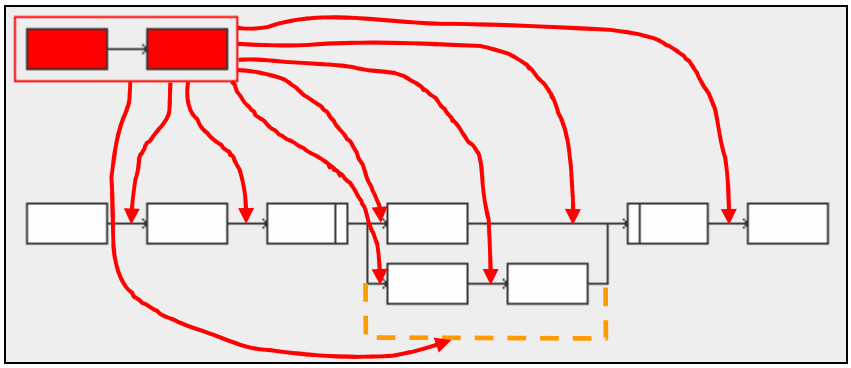


Abbildung 5-10: Änderung 2 an einem beliebigen Prozess:
Einfügen von ‚*New2*' und ‚*New3*'

- *Data3*: Löschen einer beliebigen Aktivität, wobei Start- und End-Knoten sowie Split- und Join-Knoten nicht gelöscht werden können. Das Löschen von Aktivitäten muss erlaubt sein. Abbildung 5-11 verdeutlicht, welche Aktivitäten bei diesem Prozess gelöscht werden dürfen.

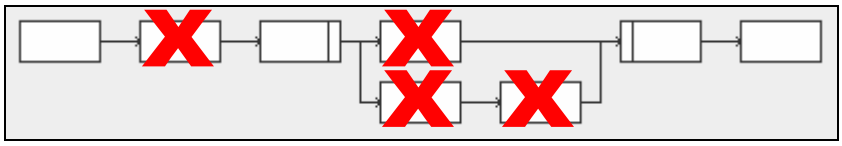


Abbildung 5-11: Änderung 3 an einem beliebigen Prozess:
Löschen einer beliebigen Aktivität

- *Data4*: Einfügen einer neuen Aktivität ‚*New4*' und Löschen derselben. Das Einfügen und Löschen von Aktivitäten muss erlaubt sein. Abbildung 5-12 zeigt die möglichen Zielbereiche für das Einfügen, wobei die Aktivität nach dem Einfügen wieder gelöscht wird, so dass der Prozess selbst wieder den gleichen Zustand wie vor der Änderung erhält.

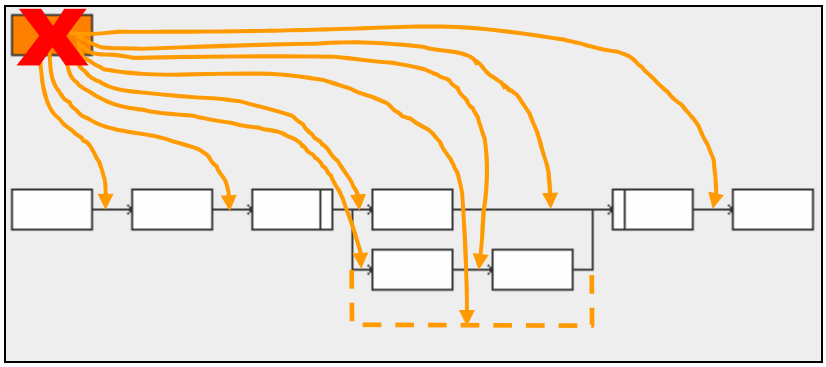


Abbildung 5-12: Änderung 4 an einem beliebigen Prozess:
Einfügen und Löschen von ‚*New4*'

- *Data5*: Verschieben einer beliebigen Aktivität, wobei Start- und End-Knoten sowie Split- und Join-Knoten nicht verschoben werden können. Hier muss bei der Bestimmung des Zielbereichs beachtet werden, dass der zu verschiebende Knoten nicht ein Knoten des Zielbereichs ist. Deshalb muss ein Prozess zum Verschieben von Aktivitäten auch mindestens vier Aktivitäten haben. Das Verschieben von Aktivitäten muss erlaubt sein. In Abbildung 5-13 sind die Aktivitäten markiert, die verschoben werden können. Für die zweite Aktivität (dunkel markiert) sind zusätzlich die möglichen Zielbereiche angegeben.

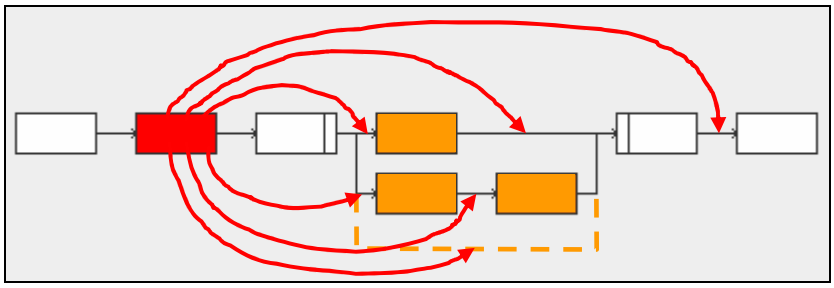


Abbildung 5-13: Änderung 5 an einem beliebigen Prozess:
Verschieben einer beliebigen Aktivität

- *Data6*: Einfügen einer neuen Aktivität ‚*New5*' und Verschieben derselben. In 25 % der Fälle wird die neue Aktivität danach auch wieder gelöscht. Das Einfügen und Verschieben von Aktivitäten muss erlaubt sein, ggf. auch das Löschen. Abbildung 5-14 zeigt wieder die möglichen Zielbereiche. Nachdem die Aktivität an einer

Stelle eingefügt wurde kann sie an einen der restlichen Zielbereiche verschoben werden.

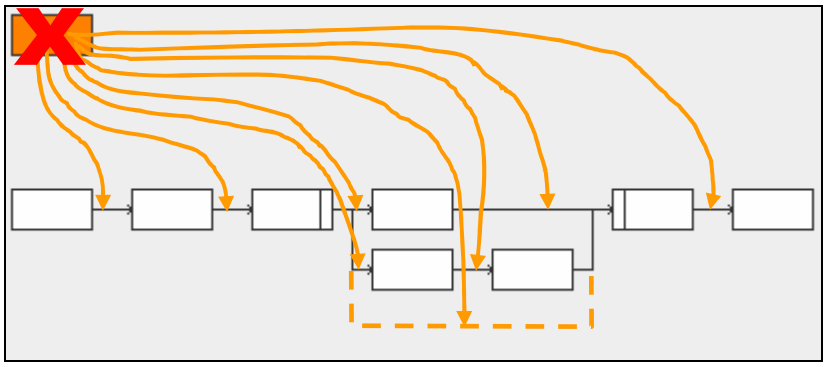


Abbildung 5-14: Änderung 6 an einem beliebigen Prozess:
Einfügen und Verschieben von ‚*New5*', ggf. Löschen

Auch diese Änderungen können kombiniert auftreten, je nachdem welche Werte in den einzelnen Datencontainern enthalten sind. So kann z.B. die Aktivität ‚*New1*' (Änderung 1) auch gelöscht (Änderung 3) oder verschoben (Änderung 5) werden. Außer den Änderungen selbst unterscheidet sich der Ablauf nicht von dem beim Prozess ‚*OP-Vorbereitung, V2*'. Nach den Änderungen werden hier ebenfalls die restlichen Schritte abgearbeitet und schließlich für jede Instanz eine XML-Log-Datei gespeichert.
Anders als per Zufallsfunktion kann nicht entschieden werden, an welche Stelle eine Aktivität eingefügt werden soll oder welche Aktivität verschoben oder gelöscht werden soll, da ja der Algorithmus für die Erzeugung von Änderungs-Logs bei jedem beliebigen Prozess anwendbar sein soll. Der (einfache) Prozess ‚*OP-Vorbereitung*' wurde als Beispielprozess für die Abbildungen nur gewählt, damit diese nicht zu unübersichtlich werden. Stattdessen hätte auch jeder andere Prozess als Grundlage für die Änderungen verwendet werden können.

5.2.3 XML-Log-Datei als Zwischenergebnis

Der *Demonstrator* erzeugt für jede einzelne Instanz eine XML-Log-Datei in der u.a. alle Änderungen und die Ausführungsreihenfolge der einzelnen Aktivitäten enthalten sind. Für die Erzeugung von Änderungs-Logs wird nicht wie bei Ausführungs-Logs die Ausführungsreihenfolge benötigt, sondern die Änderungen, weshalb die interessanten Daten an anderer Stelle im Log zu finden sind (vgl. Kapitel 3.2). Nachfolgend ist der relevante Ausschnitt aus einer der Instanz-Log-Dateien vom Prozess ‚*OP-Vorbereitung, V2*' aufgeführt, wobei sich die Struktur von Log-Dateien anderer Prozesse nicht davon unterscheidet.

```
<ChangeOperations>
  <changes.DeleteActivityOperation
       type="changes.DeleteActivityOperation"
       description="Delete activity Patient aufnehmen"
       activityID="2"
       activityName="Patient aufnehmen"
       predID="0"
       predName="Start"
```

```
        succID="3"
        succName="Blutentnahme">
      <!--  ...  -->
    </changes.DeleteActivityOperation>
    <!--  ...  -->
    <changes.InsertActivityOperation
        type="changes.InsertActivityOperation"
        description="Insert activity Erweiterte Bluttests between
                                              Blutwerte ermitteln and OP"
        activityName="Erweiterte Bluttests"
        predActivityName="Blutwerte ermitteln"
        succActivityName="OP"
        activityID="-2"
        succActivityID="4"
        predActivityID="5"
        succIsHiddenActivity="false"/>
    <!--  ...  -->
    <changes.MoveActivitiesChangeOperation
        type="changes.MoveActivitiesChangeOperation"
        description="Move the activities between Patient aufklären and
          Patient aufklären to the new location (Patient vorbereiten, OP)"
        firstActivityID="6"
        lastActivityID="6"
        destPredActivityID="7"
        destSuccActivityID="4"
        newSuccOfPredOfFirst="7"
        predOfFirstActivityID="3"
        succOfLastActivityID="7"
        predOfFirstActivityName="Blutentnahme"
        succOfLastActivityName="Patient vorbereiten"
        firstActivityName="Patient aufklären"
        lastActivityName="Patient aufklären"
        destPredActivityName="Patient vorbereiten"
        destSuccActivityName="OP"/>
</ChangeOperations>
```

Für jeden Eintrag einer Änderungsoperation gibt es die Attribute *type* (den Änderungstyp) und *description* (die textuelle Beschreibung). Es gibt drei Typen, die für das Erzeugen von Änderungs-Logs interessant sind, das sind *changes.DeleteActivityOperation*, *changes.InsertActivityOperation* und *changes.MoveActivitiesChangeOperation*. Für jeden Typ gibt es zusätzlich bestimmte Attribute, die in die Log-Daten geschrieben werden.
Für einen gelöschten Knoten wird seine ID (*activityID*), seine Bezeichnung (*activityName*), die ID und die Bezeichnung des Vorgängerknotens (*predID, predName*) und die des Nachfolgerknotens (*succID, succName*) angegeben. Auch beim Einfügen werden ID (*activityID*) und Bezeichnung (*activityName*) des Knotens angeführt, genauso wie ID und Bezeichnung von Vorgänger (*predActivityID, predActivityName*) und Nachfolger (*succActivityID, succActivityName*). Darüber hinaus wird noch angegeben, ob der nachfolgende Knoten ein versteckter Knoten ist (*succIsHiddenActivity*). Diese Information dient allerdings nur dem *Demonstrator* für die Visualisierung der Graphen und braucht deshalb nicht weiter beachtet werden.
Weil im *Demonstrator* nicht nur einzelne Knoten verschoben werden können, sondern auch Gruppen von Aktivitäten, gibt es bei verschobenen Knoten folgende Attribute zusätzlich: ID

und Name des ersten (*firstActivityID, firstActivityName*) und des letzten Knotens (*lastActivityID, lastActivityName*) der zu verschiebenden Gruppe, ID und Name vom Vorgänger des ersten Knotens (*predOfFirstActivitiyID, predOfFirstActivityName*), ID und Name vom Nachfolger des letzten Knotens (*succOfLastActivityID, succOfLastActivityName*), ID und Name von Vorgänger (*destPredActivityID, destPredActivityName*) und Nachfolger (*destSuccActivityID, destSuccActivityName*) im Zielbereich der Verschiebung und den neuen Nachfolger vom ursprünglichen Vorgänger des ersten Knotens (*newSuccOfPredOfFirst*).

Tabelle 3 listet die für die einzelnen Änderungsoperationen vorhandenen Attribute noch einmal auf. (Dunkel) Markiert sind die Attribute, die für das *Mining* von Änderungs-Logs besonders interessant sind, also Typ der Änderung, Subjekt der Änderung und der syntaktische Kontext.

Tabelle 3: Änderungs-Log-Daten, die vom Demonstrator geliefert werden

	InsertActivity-Operation	DeleteActivity-Operation	MoveActivities-ChangeOperation
type	X	X	X
description	X	X	X
activityID	X	X	
activityName	X	X	
predActivityID/predID	X	X	
predActivityName/predName	X	X	
succActivityID/succID	X	X	
succActivityName/succName	X	X	
newHiddenActivityID		X	X
succIsHiddenActivity	X		
firstActivityID			X
lastActivityID			X
destPredActivityID			X
destSuccActivityID			X
newSuccOfPredOfFirst			X
firstActivityName			X
lastActivityName			X
destPredActivityName			X
destSuccActivityName			X

Es gibt beim *Demonstrator* zusätzlich noch Änderungsoperationen für Blöcke (AND, XOR), Kanten (Synchronisation, XOR) und Datenfelder, die aber für das *Mining* von Änderungs-Logs erstmal uninteressant sind.

5.3 Das Transformieren von Änderungs-Logs

Nachdem nun von allen Instanzen Log-Dateien erzeugt wurden (im Beispiel wurden 25 Instanz-Dateien erstellt), sollen auch diese in eine *MXML*-Datei (siehe Kapitel 2.3.3) umgewandelt werden, um nachfolgend in *ProM* ausgewertet werden zu können. Wie bereits beschrieben werden Änderungsoperationen durch den Typ der Änderung, das Subjekt der Änderung (die geänderte Aktivität) und dem syntaktischen Kontext (Vorgänger, Nachfolger) der Änderung beschrieben. Weil diese Eigenschaften nicht dem Ablauf von Prozess-Ereignissen entsprechen, müssen bei Änderungs-Logs auch die optionalen *Data* Felder benutzt werden, um die Attribute in den Log-Daten zu speichern. Ansonsten ist die erzeugte XML-Datei aufgrund der zugrunde liegenden Schema-Datei natürlich ähnlich wie das Ergebnis bei Ausführungs-Logs.
Eine *MXML*-Datei enthält mindestens einen Prozess, der wiederum mehrere Prozessinstanzen haben kann. Für jede Instanz gibt es ein *Data* Feld, in dem angegeben wird, um welchen Typ von Log es sich handelt, also ob es sich um einen Ausführungs-Log (*MXML.EnactmentLog*) oder um einen Änderungs-Log (*MXML.ChangeLog*) handelt. Danach kommt für jede Änderung ein so genanntes *AuditTrailEntry*, entsprechend den Ereignissen bei Ausführungs-Logs. In diesem *AuditTrailEntry* gibt es dann ein *Data* Element mit fünf Attributen, darin werden die Eigenschaften einer Änderungsoperation gespeichert: Der Nachfolger der geänderten Aktivität (*CHANGE.postset*), der Vorgänger der geänderten Aktivität (*CHANGE.preset*), eine textuelle Beschreibung oder Begründung der Änderung (*CHANGE.rationale*), der Name der geänderten Aktivität (*CHANGE.subject*) und der Typ der Änderung (*CHANGE.type*). Drei Änderungs-Typen sind erlaubt, *INSERT*, *DELETE* und *MOVE*, bei den anderen Attributen kann beliebiger Text stehen. Außer den *Data* Attributen besteht ein *AuditTrailEntry* noch aus *WorkflowModelElement*, *EventType* und *Originator*. *WorkflowModelElement* und *EventType* sind Pflichtangaben und sind eigentlich für die Erfassung von Ausführungs-Logs gedacht. Ein *WorkflowModelElement* beschreibt das Element der Prozessbeschreibung, zu dem das aufgezeichnete Ereignis gehört (z.B. der Name einer Aktivität). Weil die Änderungen aber keinem vorgegebenen Prozessmodell folgen, gibt es solch eine Information nicht. Damit die *MXML*-Änderungs-Logs trotzdem mit herkömmlichen Process *Mining* Algorithmen kompatibel sind wird eine Verknüpfung von Änderungstyp und Änderungssubjekt (= Name der geänderten Aktivität) für das *WorkflowModelElement* verwendet. Der Ereignis-Typ ist immer *complete*, da Änderungen als atomare Operationen betrachtet werden können. Der Bearbeiter einer Änderung ist immer unbekannt (*unknown*), weil der *Demonstrator* diese Information nicht liefern kann. Im Folgenden ist ein Ausschnitt aus der erzeugten, *ProM*-konformen XML-Datei aufgeführt.

```
<ProcessInstance id="CHANGELOG_OP-Vorbereitung_5">
  <Data>
    <Attribute name="LogType">MXML.ChangeLog</Attribute>
  </Data>
  <AuditTrailEntry>
    <Data>
      <Attribute name="CHANGE.postset">Blutentnahme</Attribute>
      <Attribute name="CHANGE.preset">Start</Attribute>
      <Attribute name="CHANGE.rationale">Delete activity Patient
                                                aufnehmen</Attribute>
```

```
        <Attribute name="CHANGE.subject">Patient aufnehmen</Attribute>
        <Attribute name="CHANGE.type">DELETE</Attribute>
      </Data>
      <WorkflowModelElement>DELETE.Patient_aufnehmen
                                                  </WorkflowModelElement>
      <EventType>complete</EventType>
      <Originator>unknown</Originator>
    </AuditTrailEntry>
    <!--  ...  -->
    <AuditTrailEntry>
      <Data>
        <Attribute name="CHANGE.postset">OP</Attribute>
        <Attribute name="CHANGE.preset">Blutwerte_ermitteln</Attribute>
        <Attribute name="CHANGE.rationale">Insert activity Erweiterte
                Bluttests between Blutwerte ermitteln and OP</Attribute>
        <Attribute name="CHANGE.subject">Erweiterte Bluttests</Attribute>
        <Attribute name="CHANGE.type">INSERT</Attribute>
      </Data>
      <WorkflowModelElement>INSERT.Erweiterte_Bluttests
                                                  </WorkflowModelElement>
      <EventType>complete</EventType>
      <Originator>unknown</Originator>
    </AuditTrailEntry>
    <!--  ...  -->
    <AuditTrailEntry>
      <Data>
        <Attribute name="CHANGE.postset">OP</Attribute>
        <Attribute name="CHANGE.preset">Patient_vorbereiten</Attribute>
        <Attribute name="CHANGE.rationale">Move the activities between
                Patient aufklären and Patient aufklären to the new
                location (Patient vorbereiten, OP)</Attribute>
        <Attribute name="CHANGE.subject">Patient aufklären</Attribute>
        <Attribute name="CHANGE.type">MOVE</Attribute>
      </Data>
      <WorkflowModelElement>MOVE.Patient_aufklären</WorkflowModelElement>
      <EventType>complete</EventType>
      <Originator>unknown</Originator>
    </AuditTrailEntry>
  </ProcessInstance>
```

Wie man sieht, enthält die *ProM*-konforme XML-Datei weniger Informationen als die einzelnen Instanz-Logs zusammen. So wird z.B. die ID einer Aktivität für das *Mining* von Änderungs-Logs nicht benötigt und deshalb auch nicht übernommen. Und die Instanz-Log-Daten enthalten auch Informationen, die nur für den *Demonstrator* selbst von Bedeutung sind, wie z.B. *succIsHiddenActivity* oder *newSuccOfPredOfFirst*. Aus Sicht des *Demonstrators* wäre auch das Einfügen und Einbinden von Datencontainern eine Änderungsoperation, doch solch eine Änderung ist für das *Mining* nicht interessant und wird deshalb auch nicht in die *MXML*-Datei übernommen.

Die Transformierung kann wie bei den Ausführungs-Logs auf zwei Arten gestartet werden. Einmal können die erstellten Instanzen gleich nach der Erstellung automatisch transformiert werden, dazu muss im Hilfsdialog zur Log-Erzeugung (siehe Abbildung 5-2) das Häkchen bei der automatischen Generierung gesetzt werden. Ist das Häkchen gesetzt, werden die Instanzen

erzeugt und sofort im Anschluss werden die erzeugten Instanzen für die Transformierung verwendet. Bei der automatischen Transformierung werden auch nur die soeben erzeugten Instanzen umgewandelt. Sollen zusätzlich auch andere Instanzen mit umgewandelt werden oder die Instanzen nur zu einem späteren Zeitpunkt transformiert werden, dann kann dazu der Hilfsdialog für die Log-Transformierung verwendet werden (siehe Abbildung 5-15). In diesem Dialog kann angegeben werden, ob nur automatisch generierte Änderungs-Logs verwendet werden sollen, oder auch automatisch erzeugte Ausführungs-Logs (ohne Änderungen) oder ob sogar manuell erstellte Log-Daten für die Transformierung verwendet werden sollen. Enthält eine Instanz keine Änderungen, so wird sie zwar in die Log-Daten aufgenommen, aber ohne Änderungen entfallen die Einträge zum *AuditTrailEntry*. Solch eine ‚*leere*' Instanz liefert den *Mining* Algorithmen deshalb keine Daten, weshalb es wenig Sinn hat, Ausführungs-Log-Dateien mit zu transformieren. Mit dem Hilfsdialog für die Log-Transformierung kann die Transformierung von Ausführungs-Logs und Änderungs-Logs auch gleichzeitig gestartet werden, falls dazu eine Veranlassung besteht. Schließlich kann dann noch der Zielordner für die Ausgabedatei angegeben werden. Dieser Dialog kann entweder über den entsprechenden Menüeintrag *Export → Transform XML-logs* oder über die Schaltfläche „*Transform logs*", die bei der Anzeige eines Templates vorhanden ist, gestartet werden.

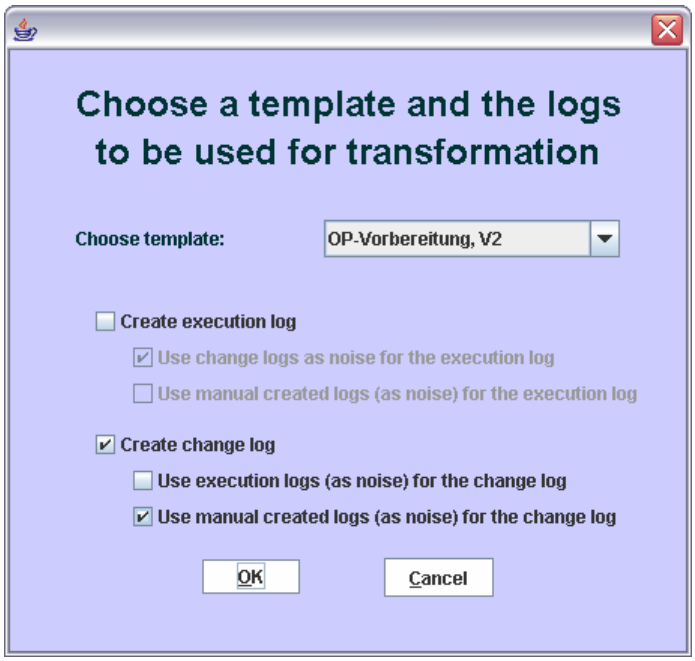


Abbildung 5-15: Hilfsdialog für die Log-Transformierung

Für die Transformierung selbst werden dann alle Instanzen verwendet, die zu der ausgewählten Prozessvorlage (Template) gehören, und die den entsprechenden Gruppen angehören, also entweder Änderungs-Logs, Ausführungs-Logs oder manuelle Log-Daten. Es können nur Instanzen verwendet werden, die auf der Festplatte gespeichert sind, da die benötigten Informationen direkt aus den XML-Dateien der einzelnen Instanzen entnommen werden. Wurden also z.B. in einem vorhergehenden Schritt die Änderungs-Logs für 25 Instanzen erzeugt, werden

bei der Transformierung derselben 25 XML-Dateien eingelesen und die entsprechenden Informationen in eine einzelne *MXML*-Datei übernommen.

Die Umwandlung an sich erfolgt durch zu Hilfenahme der *MXMLib* Bibliothek, die von den Entwicklern von *ProM* (C.W. Günther) bereits im Rahmen dieser Arbeit für die Erzeugung von Änderungs-Logs angepasst wurde. Diese Bibliothek ermöglicht eine relativ einfache Umwandlung der eigenen Log-Dateien in *MXML*. Der große Vorteil dieser Bibliothek ist die einfache Wartbarkeit, da auf eine Änderung des Formats nur der Austausch der Bibliothek erfolgen muss. Speziell bei Änderungs-Logs könnte in Zukunft eine Änderung der Schema-Datei bevorstehen, da diese ja bisher mit Hilfe der *Data* Elemente in das für Ausführungs-Logs entwickelte Schema eingepasst werden müssen. Es ist aber auch denkbar, dass für zukünftige *Mining* Algorithmen noch mehr Daten zur Verfügung gestellt werden müssen (z.B. für die Bestimmung von Gründen für Änderungen) und deshalb noch weitere Attribute in das *Data* Feld aufzunehmen sind.

6 Mining von Änderungs-Logs – Vergleich und Bewertung

Nachdem in Kapitel 5 gezeigt wurde, wie Änderungs-Logs automatisch erzeugt werden und wie sie in ein für das *Mining* brauchbares Format (*MXML*) gebracht werden, geht es in diesem Abschnitt darum, mit *Mining* Algorithmen Informationen aus diesen Log-Daten zu extrahieren. Ziel beim *Mining* von Änderungs-Logs ist die Unterstützung des Anwenders in der Prozessoptimierung. Taucht z.B. eine bestimmte Änderung immer wieder in den Log-Daten auf, dann kann der Prozessentwickler darauf aufmerksam gemacht werden, so dass diese Änderung vielleicht in das Prozessmodell einfließt und nicht jede Instanz einzeln geändert werden muss.

Für das *Mining* von Änderungs-Logs gibt es grundsätzlich zwei Möglichkeiten: Erstens die Verwendung bestehender Verfahren (siehe Kapitel 6.1), zweitens die Entwicklung eines speziellen *Change Mining Algorithmus* (siehe Kapitel 6.2), da in Abschnitt 6.1 gezeigt wird, dass die Anwendung bestehender Techniken zwar Ergebnisse liefert, diese aber durch die Hinzunahme zusätzlicher Informationen (also den Daten aus den *Data* Elementen, z.B. Vorgänger, Nachfolger) noch entscheidend verbessert werden können. In Abschnitt 6.3 werden die Ergebnisse der verschiedenen Algorithmen schließlich verglichen und bewertet.

6.1 Anwendung bestehender Mining-Algorithmen auf Änderungs-Logs

In diesem Abschnitt wird zunächst die Anwendung bereits bestehender Verfahren (also Algorithmen zum *Mining* von Ausführungs-Logs, siehe Abschnitt 3) betrachtet. Weil auch die Änderungs-Logs in eine *MXML*-Log-Datei transformiert werden, ist es einfach, diese Log-Daten als Eingabe für die bisherigen Algorithmen zu verwenden, um zu sehen, welche Schlüsse sich daraus ziehen lassen. Für die Untersuchung der Algorithmen werden sowohl Änderungs-Logs verwendet, die auf dem Prozess ‚*OP-Vorbereitung*' basieren, als auch Log-Daten, die auf dem Prozess ‚*Template5*'[16] basieren. Bei beliebigen Prozessen werden ja wie bereits gezeigt andere Änderungsoperationen angewendet, weil die Struktur des zu ändernden Prozesses nicht bekannt ist.

Um die Qualitäten der *Mining* Algorithmen bewerten zu können wurden 25 Instanzen vom Prozess ‚*OP-Vorbereitung, V2*' erzeugt, wobei in 25 % der Fälle (alle) Änderungen erlaubt waren. Tatsächlich wurden 22 Instanzen geändert, wobei sich die Anzahl der Änderungen bei den einzelnen Instanzen unterscheidet. Vom Prozess ‚*Template5*' wurden ebenfalls 25 Instanzen erzeugt, wobei wieder in 25 % der Fälle (alle) Änderungen erlaubt waren. Bei diesem Prozess wurden effektiv 21 Instanzen geändert, wobei sich auch hier die Anzahl der Änderungen wiederum unterscheidet. Ein für Änderungs-Logs guter Algorithmus sollte die Abhängigkeiten zwischen den einzelnen Änderungen erkennen. Nachfolgend ist aufgeführt, ob und welche Erkenntnisse sich aus dem *Mining* von Änderungs-Logs ziehen lassen.

[16] Für diese Beispiele hätte auch jeder andere beliebige Prozess verwendet werden können.

6.1.1 Alpha-Algorithmus

Werden die Änderungs-Logs als Eingabe für den *Alpha-Algorithmus* [AWM04, MDAW04] verwendet, so ist das Ergebnis ein Graph, der die sequentielle Ausführung der Änderungen wiedergibt, wie Abbildung 6-1 zeigt. Es wird also angezeigt, welche Änderungsoperationen aufeinander folgen und welche Abhängigkeiten zwischen ihnen bestehen könnten, wobei der Algorithmus davon ausgeht, dass die einzelnen Änderungen von der jeweils zuvor ausgeführten abhängen.

Man sieht z.B. dass auf das Einfügen von ‚*Erweiterte Bluttests*' sowohl das Löschen derselben Aktivität folgen kann wie auch das Einfügen einer weiteren Aktivität ‚*Urintest*'. Dies entspricht den implementierten Änderungsoperationen (siehe auch Kapitel 5.2.1), eine logische Änderung ist das Einfügen und das darauf folgende Löschen (Änderung3), während eine andere logische Änderung das Einfügen der beiden Aktivitäten (Änderung2) ist. Es besteht also zwischen den drei Änderungsoperationen eine gewisse Abhängigkeit. Zwischen dem Löschen von ‚*Patient aufnehmen*' und dem Einfügen von ‚*Patient entlassen*' besteht auch eine gewisse Abhängigkeit, denn die zweite Aktivität kann nur eingefügt werden, wenn die erste gelöscht wurde (Änderung1). Und die Aktivität ‚*Speicheltest*' kann auch nur zusammen mit ‚*Urintest*' eingefügt werden (Änderung5).

Der *Alpha-Algorithmus* ist also durchaus in der Lage Abhängigkeiten zwischen Änderungsoperationen zu erkennen, wobei es deutlich schwieriger ist die richtigen Abhängigkeiten zu erkennen, wenn man die zugrunde liegenden Änderungen nicht kennt.

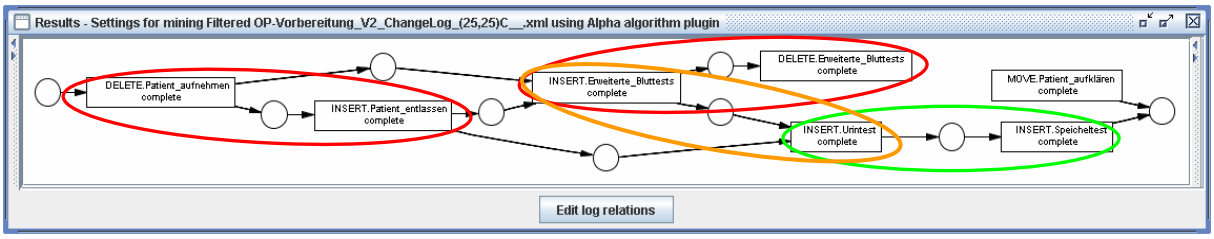


Abbildung 6-1: Beispiel Alpha-Algorithmus: OP-Vorbereitung, V2, 25 Instanzen, davon 22 individuell geändert (alles erlaubt)

Bei beliebigen Prozessen ist es durch die höhere Anzahl einzelner Änderungen (speziell Löschen und Verschieben) schwieriger, Abhängigkeiten zwischen den Operationen zu erkennen, wie Abbildung 6-2 illustriert. Ohne die Kenntnis der zugrunde liegenden Änderungsoperationen lässt sich nicht mehr bestimmen, welche Änderungen voneinander tatsächlich abhängen und welche nur in den einzelnen Instanzen zusammen vorgekommen sind. Der Algorithmus stößt also bei mehr als einer handvoll Änderungen schnell an seine Grenzen, was die Erzeugung hilfreicher Änderungsgraphen betrifft und ist deshalb für das *Mining* von Änderungs-Logs weniger geeignet.

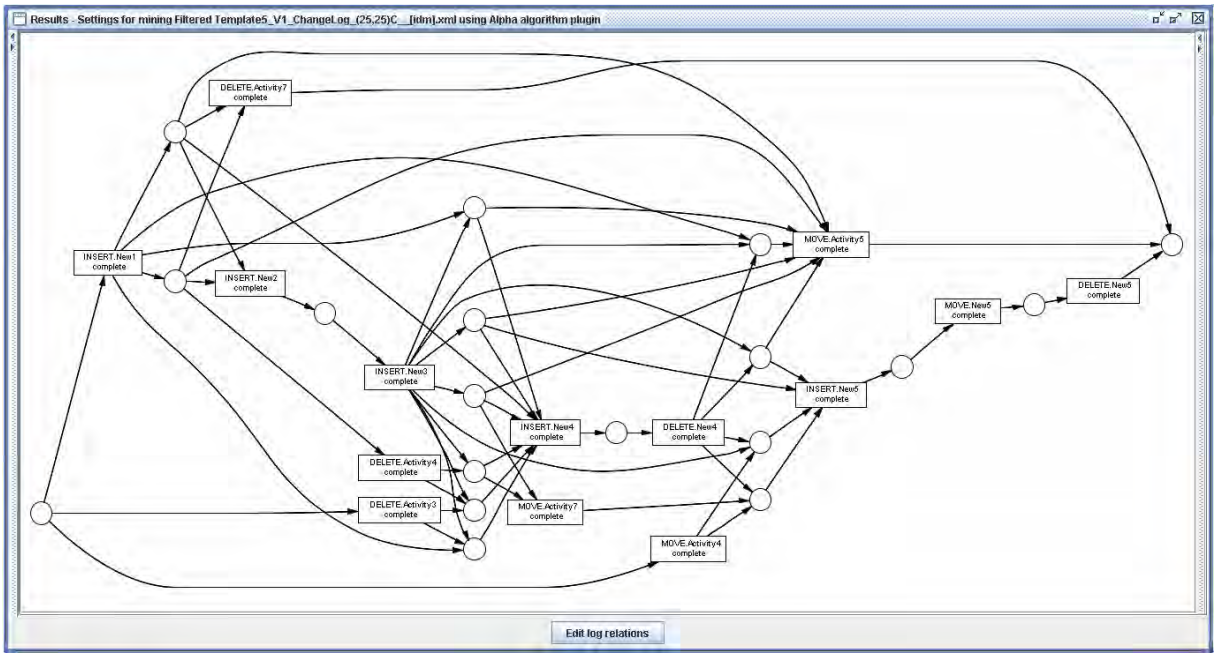


Abbildung 6-2: Beispiel Alpha-Algorithmus, Template5, V1, 25 Instanzen, davon 21 individuell geändert (alles erlaubt)

6.1.2 Multi-Phase Mining

Das *Multi-Phase Mining* [DoAa04] kann verwendet werden, um sich die Änderungen der einzelnen Instanzen anzeigen zu lassen. Abbildung 6-3 zeigt z.B. die Änderungen, die an Instanz *Nr.5* durchgeführt wurden. Ähnlich wie beim *Alpha-Algorithmus* wird die zeitliche Abfolge der Änderungen durch eine sequentielle Anordnung derselben wiedergegeben. Dass das Löschen von ‚*Erweiterte Bluttests*' (Änderung3) und das Einfügen von ‚*Urintest*' (Änderung2) parallel auftreten liegt vermutlich daran, dass sowohl die eine, wie auch die andere Änderung in den Log-Daten direkt auf das Einfügen von ‚*Erweiterte Bluttests*' folgen kann und der *Multi-Phase Mining Algorithmus* zur Erzeugung der Instanzmodelle die gesamten zugrunde liegenden Log-Daten verwendet. Bei der Instanz selbst haben die Änderungen alle hintereinander stattgefunden, d.h. zuerst wurde der Schritt ‚*Urintest*' eingefügt, dann wurde der Schritt ‚*Erweiterte Bluttests*' gelöscht (und danach wurde der Schritt ‚*Patient aufklären*' verschoben). Der Algorithmus kann also nicht erkennen, welche Änderungen tatsächlich voneinander abhängen und welche nicht.

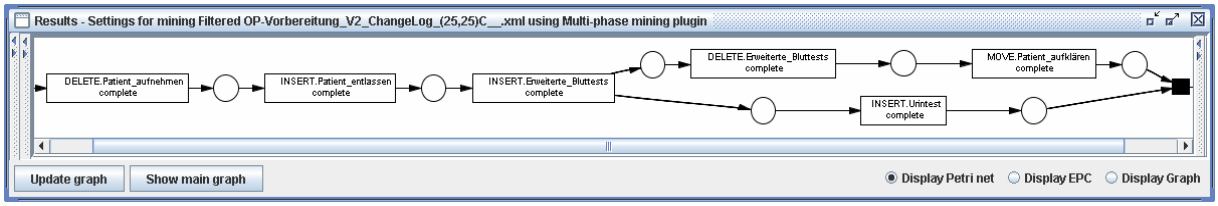


Abbildung 6-3: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 25 Instanzen (davon Instanz 5 ausgewählt, alle Änderungen erlaubt); Ergebnis als Petrinetz

Noch deutlicher zeigt sich das, wenn die Änderungsgraphen der einzelnen Instanzen zu einem Gesamtmodell zusammengefasst werden, wie Abbildung 6-4 (als EPK) zeigt. Wieder sind mehr oder weniger sequentiell die Änderungen aufgeführt, die in den einzelnen Instanzen stattgefunden haben. Gut zu erkennen ist eigentlich nur, dass nicht in jeder Instanz alle Änderungen stattgefunden haben.

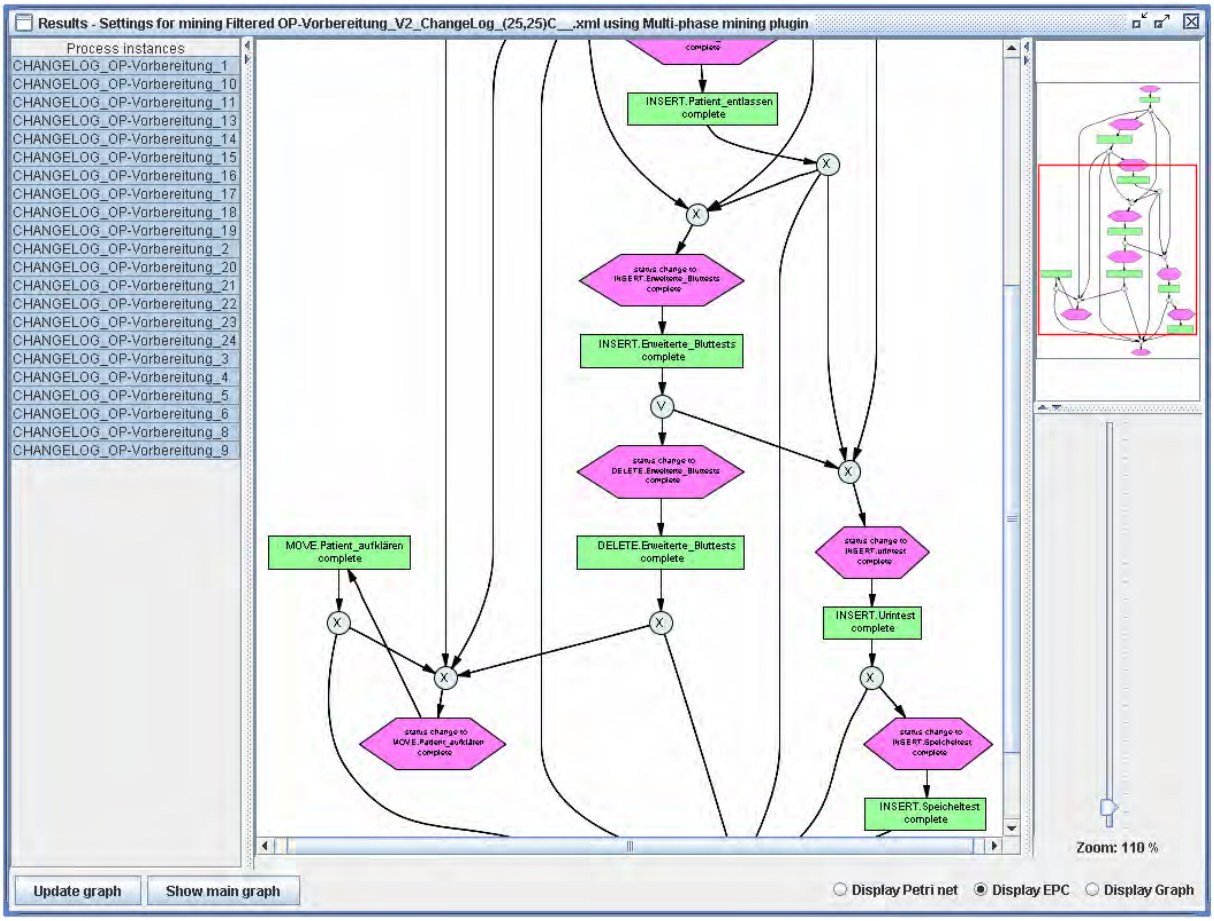


Abbildung 6-4: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 25 Instanzen (alle ausgewählt), davon 22 individuell geändert (alles erlaubt); Ergebnis als EPK

Lässt man sich das Ergebnis statt als Ereignis-Prozess-Kette als Workflow Graph (siehe Abbildung 6-5) anzeigen, so werden wie bei einem heuristischen Netz die Häufigkeiten der einzelnen Knoten und Kanten annotiert. Diese Zahlen können verwendet werden, um festzustellen, welche Änderungen tatsächlich voneinander abhängen könnten. Die Änderung ‚*Erweiterte Bluttests*' Löschen erscheint z.B. zehnmal im Log, und zehnmal ist das Einfügen derselben Aktivität der direkte Vorgänger in den Log-Daten, so dass mit hoher Wahrscheinlichkeit gesagt werden kann, dass die zweite Änderung von der ersten abhängt, was offensichtlich ist, da eine Aktivität nur gelöscht werden kann, nachdem sie eingefügt wurde. Ähnlich aussagekräftig sind die Häufigkeitswerte beim Einfügen von ‚*Erweiterte Bluttests*' (10) und ‚*Urintest*' (12), sowie beim Einfügen von ‚*Urintest*' (4) und ‚*Speicheltest*' (4). Diese Änderungen entsprechen ebenfalls jeweils einer logischen Änderung im *Demonstrator* (vgl. Kapitel 5.2.1).

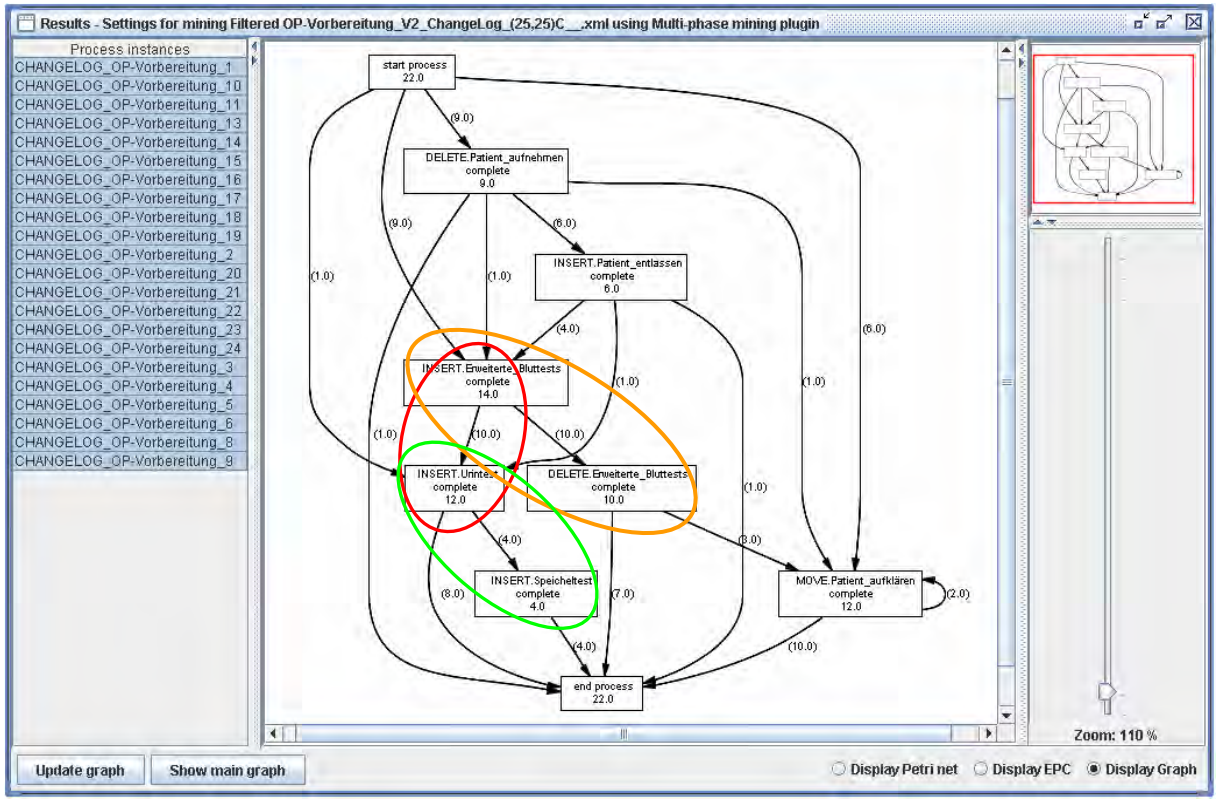


Abbildung 6-5: Beispiel Multi-Phase Mining: OP-Vorbereitung, V2, 25 Instanzen (alle ausgewählt), davon 22 individuell geändert (alles erlaubt); Ergebnis als Workflow Graph

Werden Änderungs-Log-Daten von beliebigen Prozessen verwendet, so zeigt sich ein ähnliches Ergebnis wie beim *Alpha-Algorithmus*, nämlich dass es mit zunehmender Anzahl einzelner Änderungen (wieder speziell Lösch- und Verschiebe-Operationen) für den Algorithmus schwierig ist, irgendwelche Abhängigkeiten zwischen den Änderungen zu erkennen. Anhand der Häufigkeitszahlen im Workflow Graphen kann der Betrachter selbst die Änderungen suchen, die tatsächlich voneinander abhängig sind, aber eine bessere Unterstützung gibt es vom *Multi-Phase Mining* nicht.

6.1.3 Tsinghua-Alpha-Algorithmus

Wie Abbildung 6-6 zeigt, ist der *Tsinghua-Alpha-Algorithmus* [WWAW04] nicht für das *Mining* von Änderungs-Logs geeignet. Das liegt daran, dass der Algorithmus zur korrekten Funktionsweise die beiden Ereignisse *start* und *complete* im Log benötigt, bei Änderungsoperationen aber nur ein Ereignis *complete* protokolliert wird, da die Änderungsoperationen atomar zu betrachten sind. Der Änderungsgraph, der als Ergebnis geliefert wird, lässt keinerlei Rückschlüsse zu, welche Änderungen voneinander abhängig sind und welche nicht, egal welcher Prozess zugrunde liegt.

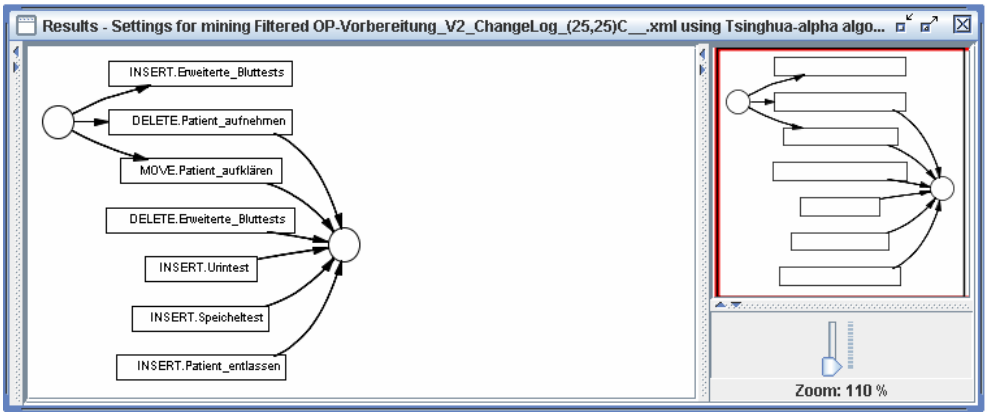


Abbildung 6-6: Beispiel Tsinghua-Alpha-Algorithmus: OP-Vorbereitung, V2, 25 Instanzen, davon 22 individuell geändert (alles erlaubt)

6.1.4 Heuristics Miner

Der *Heuristics Miner* [WeAa01, WeAa03] ist von den herkömmlichen Algorithmen noch am besten für das *Mining* von Änderungs-Logs geeignet, da auch hier anhand der annotierten Häufigkeitszahlen entschieden werden kann, wie sicher eine Abhängigkeit zwischen zwei Änderungen ist. So sind die Beziehungen der tatsächlich abhängigen Änderungen deutlich häufiger wie solche, die nicht wirklich voneinander abhängen, wie Abbildung 6-7 zeigt.

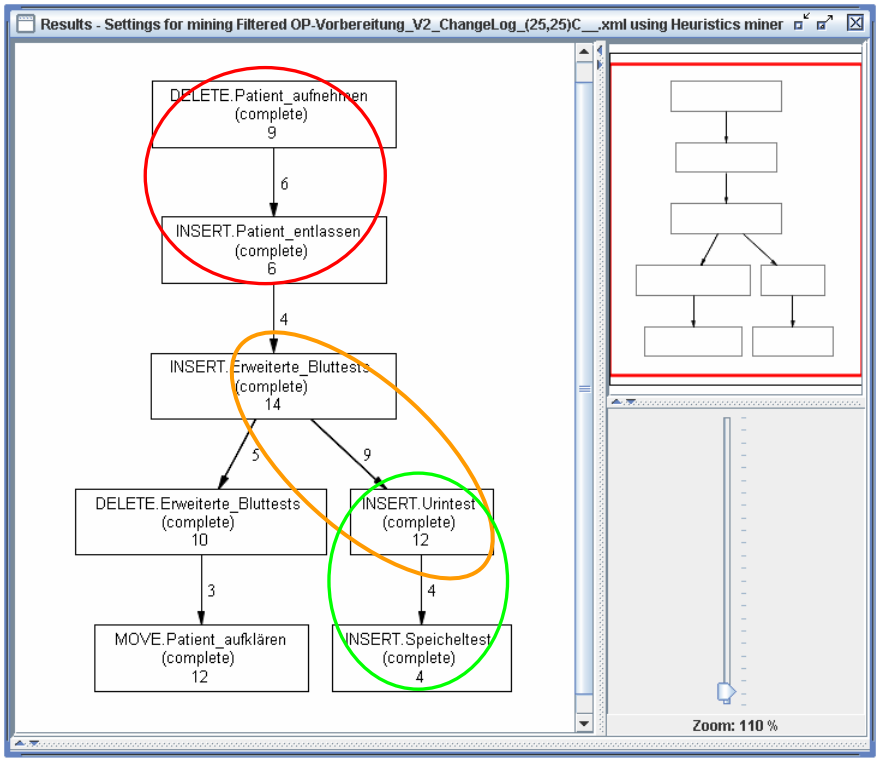


Abbildung 6-7: Beispiel Heuristics Miner: OP-Vorbereitung, V2, 25 Instanzen, davon 22 individuell geändert (alles erlaubt)

Abbildung 6-8 zeigt nun den Änderungsgraph des Prozesses ‚*Template5, V1*'. Anhand der Häufigkeitszahlen kann deutlich erkannt werden, welche Änderungen voneinander abhängen. So hängt das Einfügen und Löschen der Aktivität ‚*New4*' voneinander ab, was daran erkannt werden kann, dass beide Änderungen gleich häufig (7-mal) auftreten. Auch gleich häufig ist das Einfügen der Aktivitäten ‚*New2*' und ‚*New3*' (5-mal) sowie das Einfügen und Verschieben der Aktivität ‚*New5*' (8-mal). In zwei weiteren Fällen wird die Aktivität ‚*New5*' auch wieder gelöscht, aber erst nachdem sie verschoben wurde. Alle anderen Änderungen, die nicht direkt voneinander abhängen sind deutlich seltener in den Log-Daten vertreten und werden auch graphisch nahezu parallel zueinander dargestellt, so dass recht gut erkannt werden kann, welche Änderungen tatsächlich voneinander abhängen. Speziell bei diesem Prozess mit den drei verschiedenen Löschoperationen und den drei verschiedenen Verschiebeoperationen (und allen anderen beliebigen Prozessen mit vielen verschiedenen Änderungen) zeigt sich der Vorteil des *Heuristics Miner* gegenüber den anderen herkömmlichen Algorithmen. Denn die anderen Algorithmen können bei solch einer großen Anzahl verschiedener Änderungen (mit vielen vermeintlichen Abhängigkeiten) bei der Aufzeigung der tatsächlichen Abhängigkeiten keine große Hilfe sein.

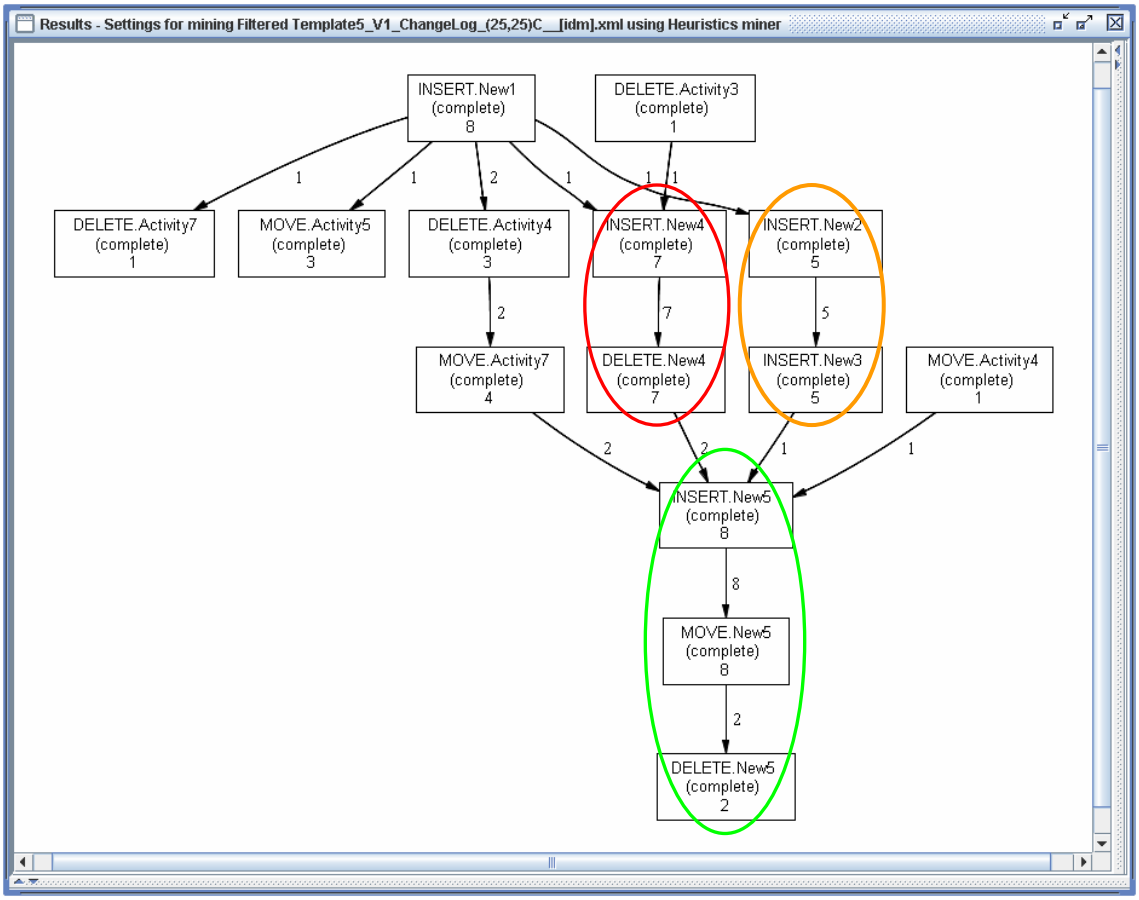


Abbildung 6-8: Beispiel Heuristics Miner: Template5, V1, 25 Instanzen, davon 21 individuell geändert (alles erlaubt)

6.2 Mining von Änderungs-Logs unter Ausnutzung zusätzlicher Information

Wie das *Mining* von Änderungs-Logs mit den bisherigen Algorithmen zeigt, sind die Ergebnisse nur in wenigen Fällen wirklich aussagekräftig. Aus diesem Grund soll im Folgenden ein neuer Algorithmus entwickelt werden, der die zusätzlichen Informationen nutzt, die eine Änderungsoperation mit sich bringt. Dies stellt eine völlig neue Art von *Mining* Algorithmen dar. Zunächst werden in Abschnitt 6.2.1 die Konzepte für den *Change Mining Algorithmus* vorgestellt, bevor der neue Algorithmus dann in Kapitel 6.2.2 evaluiert wird.

6.2.1 Konzept für den Change Mining Algorithmus

Ähnlich wie beim *Mining* von Ausführungs-Logs kann die Reihenfolge, in der die Änderungsoperationen im Log auftreten, als kausale Abhängigkeiten ausgenutzt werden. Wie das *Mining* mit bestehenden Algorithmen (siehe Abschnitt 6.1) aber zeigt, reichen die kausalen Abhängigkeiten nicht aus, um alle Beziehungen zwischen den Änderungen korrekt zu erkennen. Aufgrund der speziellen Art der Einträge (nämlich Änderungsoperationen) können zusätzliche Informationen genutzt und so der Zustand des Änderungsprozesses ermittelt werden. Über den Zustand eines Änderungsprozesses (der durch das zugrunde liegende Prozessschema definiert wird) kann man die Effekte verschiedener Änderungsoperationen vergleichen, um so festzustellen, ob zwei Änderungen in anderer Reihenfolge zum gleichen Ergebnis führen. Ist das der Fall spricht man vom Kommutativität.

Kommutativität ist folgendermaßen definiert[17] (formale Definition in [GRRA06]): Zwei Änderungsoperationen sind kommutativ, wenn sie exakt die gleiche Auswirkung auf ein Prozessschema haben, egal in welcher Reihenfolge sie angewendet werden. Sind zwei Änderungsoperationen nicht kommutativ, werden sie als abhängig betrachtet, d.h. das Ergebnis der zweiten Änderung hängt von der ersten ab.

Es hat z.B. keinen Einfluss auf den Ergebnisprozess, in welcher Reihenfolge zwei Aktivitäten aus dem Prozess gelöscht werden, wohingegen das Einfügen und Löschen einer Aktivität nur in dieser Reihenfolge stattfinden kann (man kann eine Aktivität nicht Löschen, bevor sie nicht eingefügt wurde). Finden die Änderungen in umgekehrter Reihenfolge statt, dann muss die Aktivität zum Löschen schon vorher vorhanden sein und das Ergebnis der Änderungsoperationen unterscheidet sich auch von dem Beispiel mit anderer Reihenfolge (einmal ist die Aktivität nach den Änderungen vorhanden (Löschen → Einfügen), einmal nicht (Einfügen → Löschen)).

Mit dem Konzept der Kommutativität kann die Menge der potenziell abhängigen Log-Einträge auf die tatsächlich abhängigen Einträge reduziert werden. Der Algorithmus geht nun wie folgt vor: Aufgrund der Reihenfolge im Log ergeben sich die möglichen Beziehungen zwischen den Änderungsoperationen, nach der Analyse, ob Kommutativität vorliegt oder nicht, werden nur noch die tatsächlich vorhandenen Abhängigkeiten für die Erzeugung von

[17] Diese Art der Kommutativität ist nur syntaktisch, man könnte auch semantisch abhängige Log-Einträge haben, so dass trotz (syntaktischer) Kommutativität die Reihenfolge der Änderungen eine Rolle für das Ergebnis der Änderungsoperationen spielt. Dieses Thema ist aber nicht mehr im Bereich dieser Arbeit.

Änderungsinstanzen für die einzelnen Änderungs-Logs verwendet. Nun stellt sich die Frage, wie die einzelnen Änderungsinstanzen zusammengeführt werden können, um alle möglichen Änderungen auf Basis eines Prozesses zu einem Ergebnis zu vereinen. Als Lösung bietet sich die Nutzung eines bestehenden Algorithmus an: Der *Multi-Phase Mining Algorithmus* [DoAa04] (siehe Kapitel 3.3.2) erzeugt grundsätzlich auch für jede einzelne Instanz einen Graphen und fasst diese dann zu einem Ergebnis zusammen. Durch die Erweiterung des *Multi-Phase Mining Algorithmus* um das Konzept der Kommutativität entsteht der *Change Mining Algorithmus* [GRRA06], der speziell für das *Mining* von Änderungs-Logs entwickelt wurde (und auch nur Änderungs-Logs als Eingabe versteht). Der *Change Mining Algorithmus* wurde in Kooperation mit der TU Eindhoven entwickelt und kann genauso wie der *Multi-Phase Mining Algorithmus* Petrinetze und Ereignis-Prozess-Ketten (EPK) aus den Abhängigkeitsbeziehungen im Log erzeugen.

Durch die Erweiterung des *Multi-Phase Mining Algorithmus* um das Konzept der Kommutativität verbessert sich die Übersichtlichkeit und die Qualität des Änderungsprozesses deutlich, da nicht jedes Paar von Änderungsoperationen in beiden möglichen Reihenfolgen im Log beobachtet werden muss, um sie als gleichzeitig zu erkennen. Dies ist speziell im Zusammenhang mit Änderungs-Logs von Bedeutung, da Änderungen weniger häufig vorkommen als die tatsächliche Ausführung eines Prozessschemas, d.h. es werden viel mehr Instanzen ohne Änderung protokolliert als solche mit. Weil weniger Log-Daten zur Verfügung stehen müssen diese effektiver genutzt werden, was durch das Konzept der Kommutativität möglich wird. Bei Änderungs-Logs die mit dem *Demonstrator* erzeugt wurden, kann nur anhand von Kommutativität festgestellt werden, welche Änderungen gleichzeitig ablaufen können, da alle Änderungen immer in der gleichen Reihenfolge ablaufen. Wie gut das funktioniert zeigt das nächste Kapitel, in dem der *Change Mining Algorithmus* mit den bestehenden Algorithmen verglichen wird.

6.2.2 Evaluierung des Change Mining Algorithmus

Um die Qualitäten des *Change Mining Algorithmus* bewerten und die Ergebnisse mit den bestehenden Algorithmen (speziell dem *Multi-Phase Mining*) vergleichen zu können, werden auch hier dieselben Änderungs-Logs verwendet, d.h. vom Prozess ‚*OP-Vorbereitung*' Änderungs-Logs mit 25 Instanzen, wobei in 25 % der Fälle (alle) Änderungen erlaubt waren (Tatsächlich wurden 22 Instanzen geändert, wobei sich die Anzahl der Änderungen bei den einzelnen Instanzen unterscheidet).

Abbildung 6-9 zeigt den Änderungsgraph einer einzelnen Instanz (Instanz *Nr.1*) bei der zwei Änderungen stattgefunden haben. Bei dieser Instanz wurde die Aktivität ‚*Erweiterte Bluttests*' eingefügt und danach wieder gelöscht. Es ist offensichtlich, dass diese beiden Änderungen voneinander abhängen, denn nur wenn der Schritt vorhanden ist, kann er auch gelöscht werden. Völlig korrekt hat der Algorithmus erkannt, dass diese beiden Änderungen nicht kommutativ sind, weshalb die beiden Änderungen im Graph auch voneinander abhängen. Faktisch werden diese beiden Änderungen im *Demonstrator* auch in einem logischen Schritt durchgeführt (vgl. Kapitel 5.2.1, Änderung3), falls die Aktivität nicht bereits vorher eingefügt wurde.

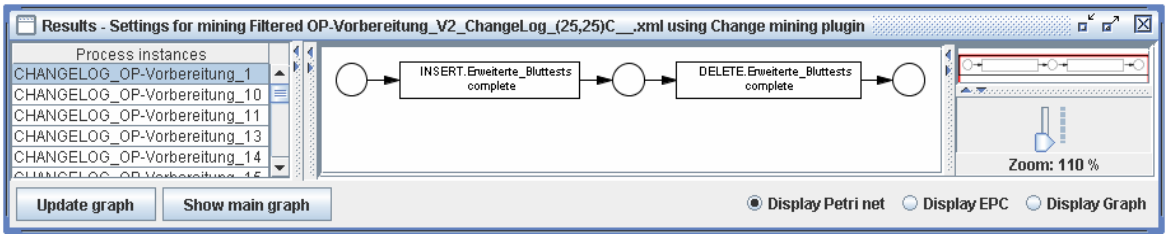


Abbildung 6-9: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (davon Instanz 1 ausgewählt), 25 % Änderungen erlaubt; Ergebnis als Petrinetz

Im nächsten Beispiel ist eine andere Instanz (Instanz *Nr.5*) ausgewählt. Bei dieser Instanz haben mehrere Änderungen stattgefunden, insgesamt wurden sechs Änderungsoperationen aufgezeichnet. Auch bei dieser Instanz haben die beiden abhängigen Änderungen ‚*Erweiterte Bluttests*' Einfügen und Löschen stattgefunden, daneben wurde die Aktivität ‚*Patient aufklären*' verschoben, die Aktivität ‚*Patient aufnehmen*' gelöscht und die beiden Aktivitäten ‚*Urintest*' und ‚*Patient entlassen*' eingefügt. Abbildung 6-10 zeigt den Graph der Änderungen als Petrinetz. Im Vergleich zum Ergebnis beim *Multi-Phase Mining* (Abbildung 6-3) erkennt der Algorithmus aufgrund der Kommutativität deutlich mehr Parallelität zwischen den einzelnen Änderungen, es wird also erkannt, dass die meisten Änderungsoperationen unabhängig voneinander ablaufen (können).

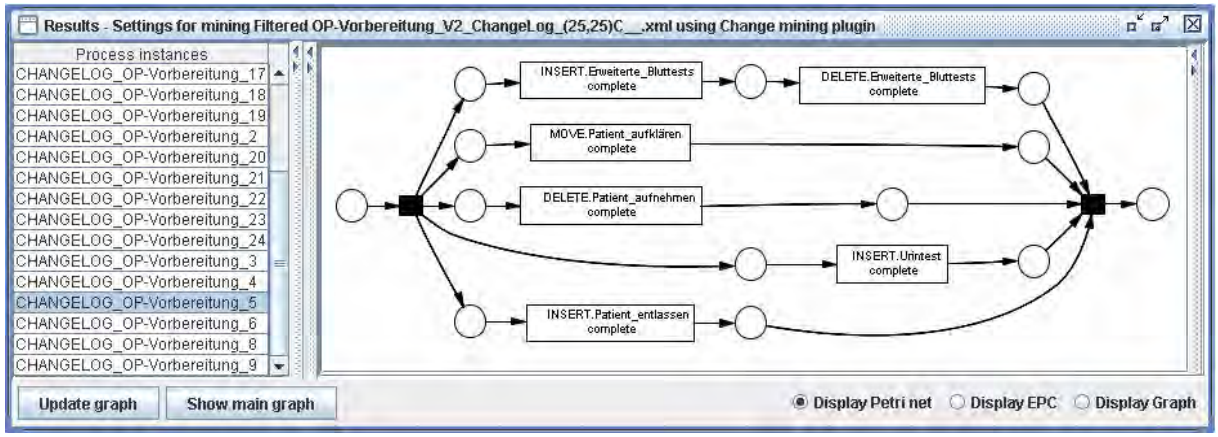


Abbildung 6-10: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (davon Instanz 5 ausgewählt, alle Änderungen erlaubt); Ergebnis als Petrinetz

Bei dieser Instanz zeigt sich aber auch, dass gewünschte Abhängigkeiten zwischen Änderungen nicht erkannt werden können, wenn diese kommutativ sind. So wurde die Aktivität ‚*Urintest*' zusammen mit der Aktivität ‚*Erweiterte Bluttests*' eingefügt (logische Änderung2, vgl. Kapitel 5.2.1), da diese Aktivitäten aber parallel stattfinden macht es keinen Unterschied, in welcher Reihenfolge die Änderungen auf den ursprünglichen Prozess angewandt werden, das Ergebnis bleibt dasselbe. Ebenfalls kann die Aktivität ‚*Patient entlassen*' nur eingefügt werden, wenn die Aktivität ‚*Patient aufnehmen*' gelöscht wurde (logische Änderung1). Da die beiden Aktivitäten aber in unterschiedlichen Bereichen im Prozess stattfinden ist es wiederum egal, in welcher Reihenfolge die Änderungen stattfinden, da es keinen Einfluss auf das Er-

gebnis hat, wann welche Änderung stattfindet. Abbildung 6-11 zeigt erneut den Graph der Änderungen, diesmal als Ereignis-Prozess-Kette. Wie deutlich zu erkennen ist, werden dem Änderungsgraph zusätzliche Ereignisse als Start- und End-Ereignis hinzugefügt, damit alle Funktionen zu einem Graph verbunden werden können.

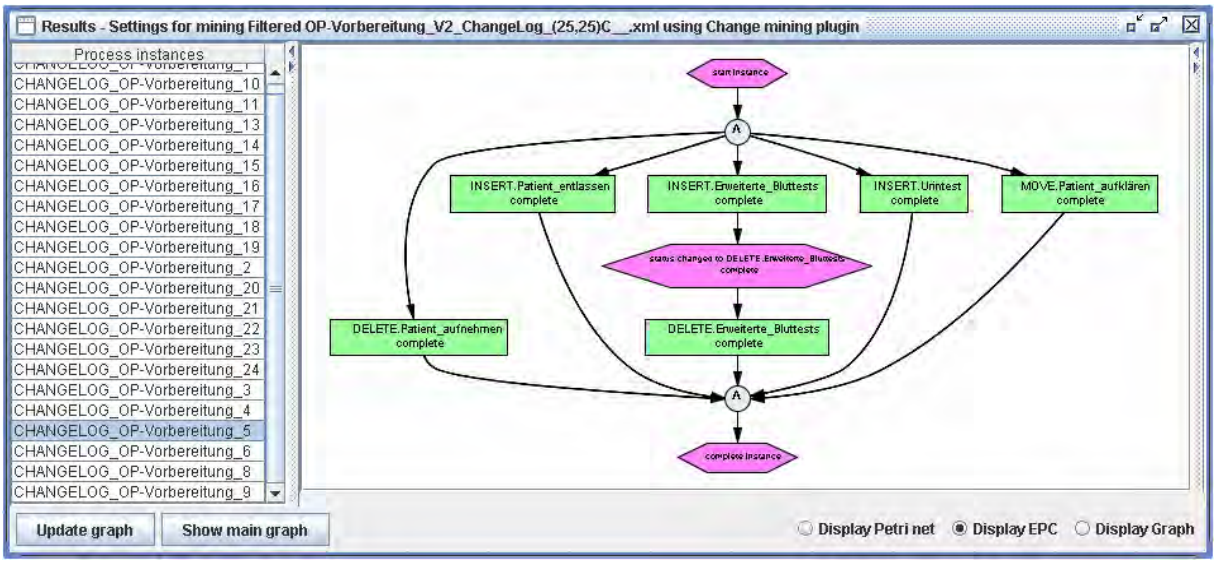


Abbildung 6-11: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (davon Instanz 5 ausgewählt, alle Änderungen erlaubt); Ergebnis als EPK

Abbildung 6-12 zeigt den Änderungsgraph als Workflow Graph, wobei in dieser Ansicht die zusätzlichen Start- und Ende-Ereignisse fehlen und die einzelnen parallelen Änderungen somit auch nicht verbunden sind. Im Gegensatz zum Workflow Graph aller Instanzen (vgl. Abbildung 6-5) werden bei einzelnen Instanzen keine Häufigkeitszahlen annotiert, da ja jeder Knoten (≙ Änderungsoperation) und jede Kante (≙ Beziehung) in einer Instanz nur einmal vorkommen kann.

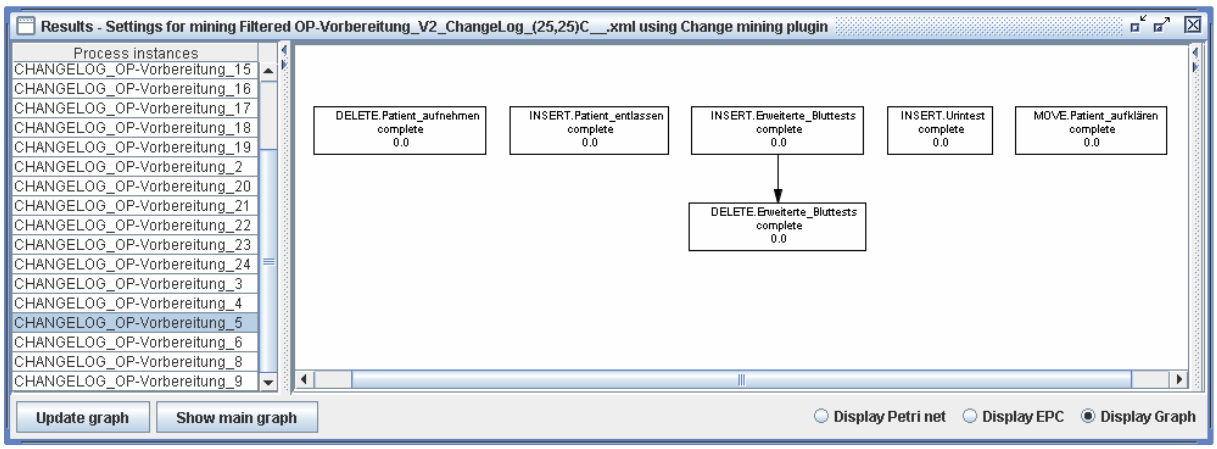


Abbildung 6-12: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (davon Instanz 5 ausgewählt, alle Änderungen erlaubt); Ergebnis als Workflow Graph

Wenn man alle Änderungsgraphen der Instanzen zu einem Gesamtmodell zusammenfasst werden bei ausreichend Log-Daten alle möglichen Änderungen und die Beziehungen untereinander angezeigt. Abbildung 6-13 zeigt das Ergebnis als Petrinetz. Im Vergleich zum Ergebnis einer einzelnen Instanz werden die einzelnen Änderungen nicht mehr mit einer Start- und einer End-Stelle verbunden, da ja nicht immer alle Änderungen stattfinden, man mit Petrinetzen aber nur UND-Verzweigungen und ausschließliche ODER-Verzweigungen darstellen kann. Man muss den Graph also so interpretieren, dass alle einzelnen Pfade ausgeführt werden können oder auch nicht. Beim Beispielprozess ‚*OP-Vorbereitung, V2*' reichen die Änderungen in 25 Instanzen aus, um alle möglichen Änderungen zu erhalten und auch um sie korrekt anzuzeigen.

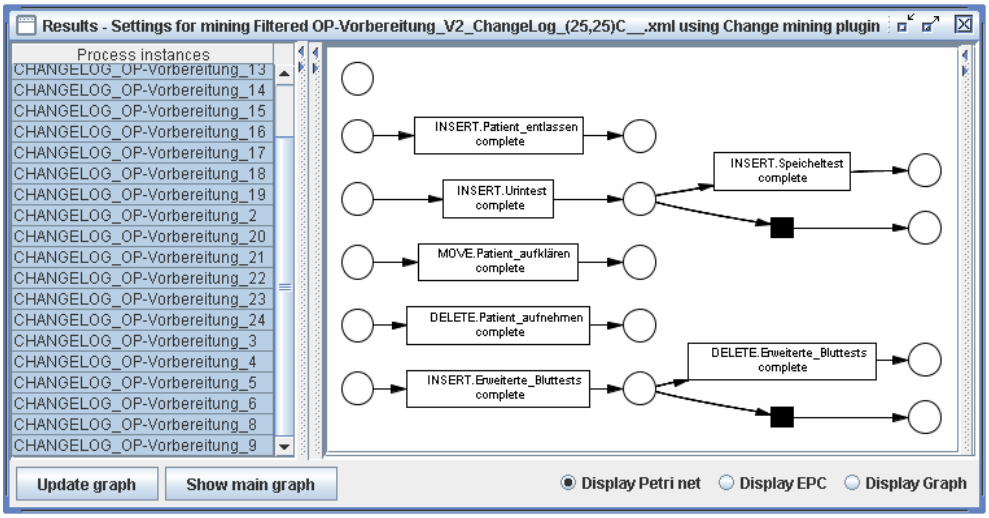


Abbildung 6-13: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (alle ausgewählt), davon 22 individuell geändert (alles erlaubt); Ergebnis als Petrinetz

Abbildung 6-14 zeigt den Änderungsgraph als Ereignis-Prozess-Kette. Weil es bei EPKs ODER-Verzweigungen (Symbol: V) gibt, werden in dieser Ansicht ein Start- und ein End-Ereignis angezeigt. Wie man dem Graph weiterhin entnehmen kann, können die Änderungen ‚*Erweiterte Bluttests*' Löschen und ‚*Speicheltest*' Einfügen optional stattfinden. Graphisch wird das jeweils durch einen Ausweichpfad dargestellt. Für den Gesamtprozess der Änderungen eignet sich die Ansicht als EPK besser, weil schon im Graph deutlich wird, dass die Änderungen alle für sich optional sind, also stattfinden können oder nicht. Dies muss im Gegensatz zur Ansicht als Petrinetz nicht so interpretiert werden, da es bei EPKs explizit ‚echte' ODER-Verzweigungen gibt. Betrachtet man nur eine einzelne Instanz, werden keine ODER-Verzweigungen benötigt, da ja alle beobachteten Änderungen auch bei dieser Instanz stattgefunden haben. Wenn man noch mal Abbildung 6-11 betrachtet, sieht man, dass hier nur eine UND-Verzweigung benötigt wird.

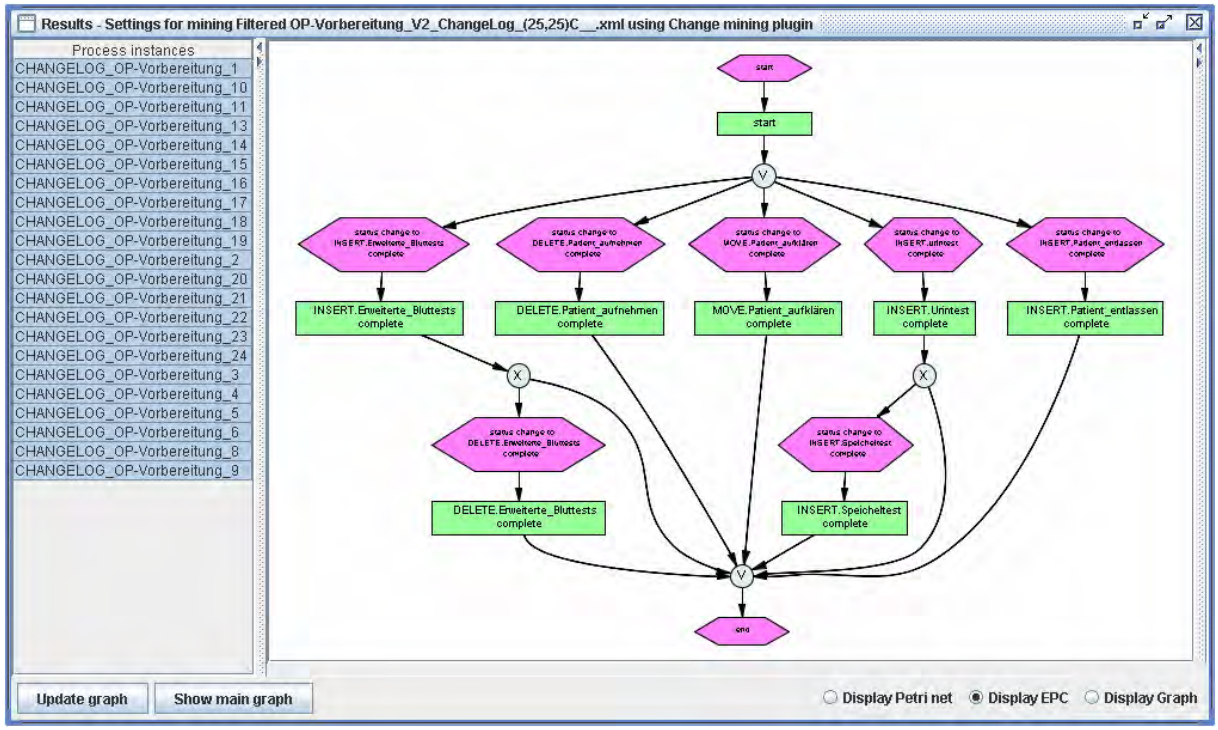


Abbildung 6-14: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (alle ausgewählt), davon 22 individuell geändert (alles erlaubt); Ergebnis als EPK

Bei der Ansicht des Gesamtprozesses als Workflow Graph (siehe Abbildung 6-15) werden im Gegensatz zur Ansicht bei einzelnen Prozessen künstliche Start- und End-Knoten verwendet. Zusätzlich ist bei den einzelnen Knoten und Kanten annotiert, wie oft sie ausgeführt wurden. Es wird also angezeigt, wie oft eine Änderung stattgefunden hat und wie oft bestimmte Optionen gewählt wurden. Zum Beispiel wurde die Aktivität ‚*Urintest*' zwölf mal eingefügt, während die davon abhängige Änderung ‚*Speicheltest*' Einfügen nur viermal vorgekommen ist. Bei dieser Ansicht lassen sich also erste Erkenntnisse ziehen, welche Änderungen wie häufig vorkommen und deshalb vielleicht auch in den Prozess eingebaut werden sollen, so dass nicht jede Instanz extra geändert werden muss. Die Aktivität ‚*Urintest*' wurde z.B. in 48 % (12/25) der Fälle eingefügt, so dass bei diesem Beispiel davon ausgegangen werden kann, dass der ursprüngliche Prozess schlecht modelliert wurde, da dieser Schritt vergessen wurde, obwohl er oft benötigt wird. Bei realen Prozessen sind solch hohe Wahrscheinlichkeiten vermutlich eher selten, dennoch lassen sich aus den Beobachtungen Hinweise für eine mögliche Prozessverbesserung ableiten. Wenn eine Aktivität z.B. besonders häufig gelöscht wurde kann es sein, dass sie nicht mehr benötigt wird und deshalb aus dem Prozess entfernt werden kann.

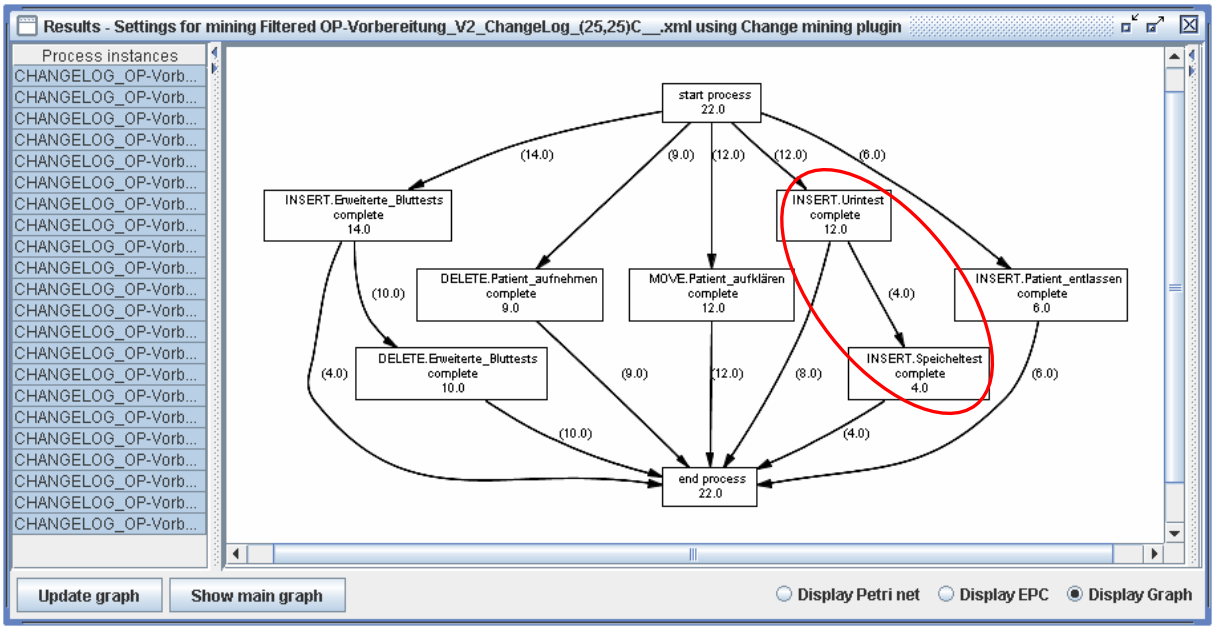


Abbildung 6-15: Beispiel Change Mining: OP-Vorbereitung, V2, 25 Instanzen (alle ausgewählt), davon 22 individuell geändert (alles erlaubt); Ergebnis als Workflow Graph

Neben dem Prozess ‚*OP-Vorbereitung, V2*' kann der *Demonstrator* auch jeden anderen beliebigen Prozess auf Instanzebene ändern und die Änderungen in Log-Daten festhalten. Nachfolgend ist aufgeführt, wie der *Change Mining Algorithmus* mit den Änderungs-Logs weiterer Prozesse zurechtkommt.

Abbildung 6-16 zeigt den Änderungsgraph einer einzelnen Instanz (Instanz 18) des Prozesses ‚*Template5, V1*' als Ereignis-Prozess-Kette. In dieser Instanz haben sieben Änderungen stattgefunden die teilweise voneinander abhängen. Eine neue Aktivität ‚*New1*' wurde unabhängig von anderen Änderungen eingefügt, die Aktivität ‚*Activity4*' wurde unabhängig von anderen Änderungen gelöscht. Die anderen Änderungen hängen jeweils voneinander ab, so wird eine neue Aktivität ‚*New4*' eingefügt und gleich danach wieder gelöscht. Die Aktivität ‚*New5*' wird eingefügt, dann verschoben und danach ebenfalls wieder gelöscht. Es ist offensichtlich, dass Einfügen und Löschen (ggf. auch Verschieben) einer Aktivität voneinander abhängen. Der *Change Mining Algorithmus* erkennt bei dieser einzelnen Instanz die Änderungen und deren Abhängigkeiten korrekt.
Um einiges unübersichtlicher sieht das Ergebnis aus, wenn man den Gesamtprozess der Änderungen betrachtet (siehe Abbildung 6-17). Es wird richtig erkannt, dass das Einfügen von ‚*New3*' vom Einfügen von ‚*New2*' abhängt, ebenfalls wird richtig erkannt, dass das Löschen von ‚*New4*' und ‚*New5*' von den jeweiligen Einfügeoperationen abhängt. Doch der Rest des Graphen ist relativ unübersichtlich. Das liegt daran, dass per Zufallsfunktion entschieden wird, welche Aktivität gelöscht bzw. verschoben werden soll. So kann in jeder Instanz eine andere Aktivität gelöscht bzw. verschoben werden. Tatsächlich wurden dann auch in manchen Instanzen die Aktivitäten ‚*Activity3*', ‚*Activity4*' und ‚*Activity7*' gelöscht und in anderen Instanzen die Aktivitäten ‚*Activity4*', ‚*Activity5*' und ‚*Activity7*' verschoben.

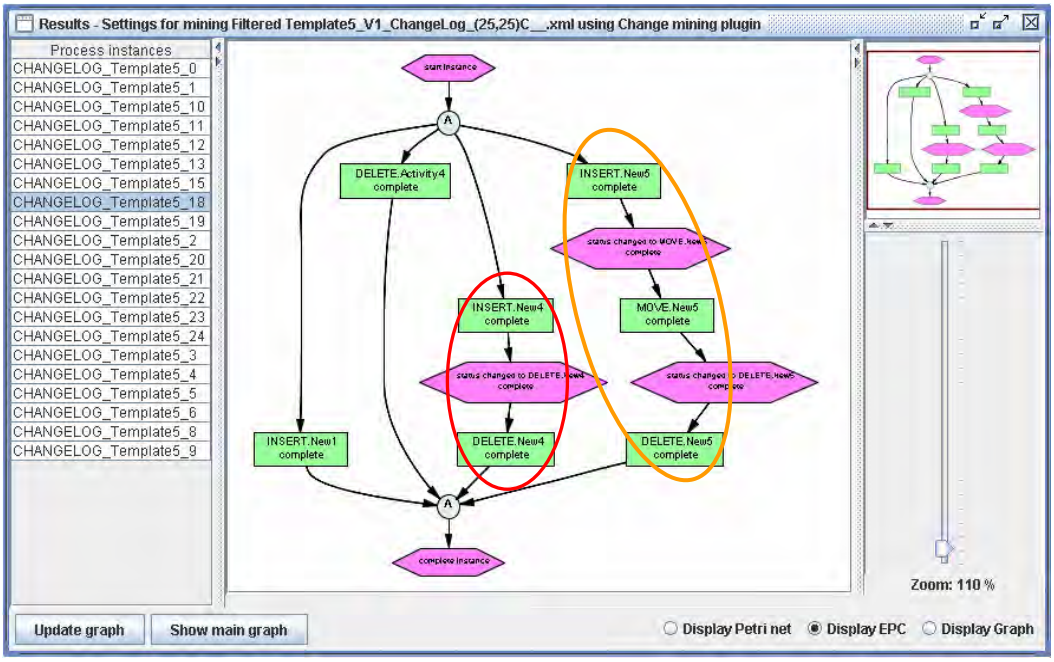


Abbildung 6-16: Beispiel Change Mining: Template5, V1, 25 Instanzen (davon Instanz 18 ausgewählt, alle Änderungen erlaubt); Ergebnis als EPK

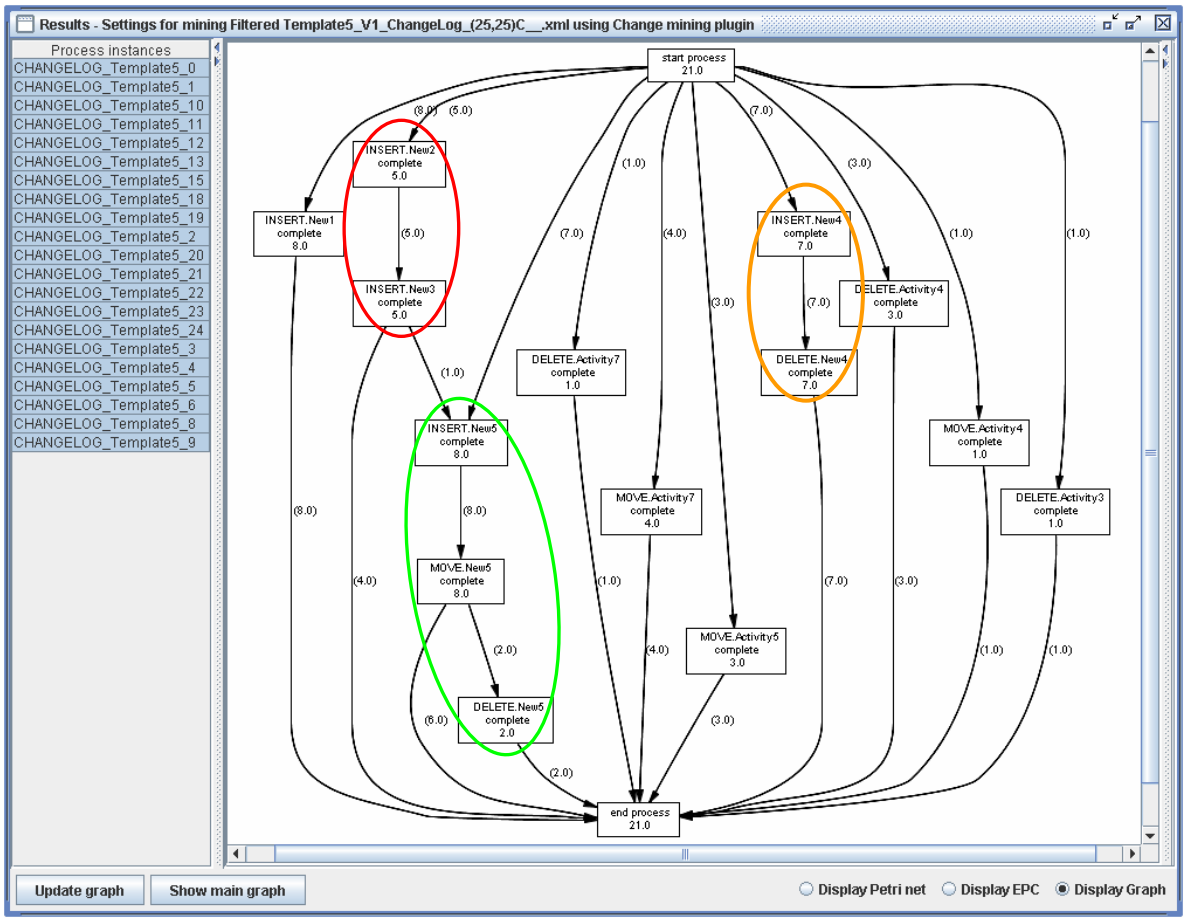


Abbildung 6-17: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 21 individuell geändert (alles erlaubt); Ergebnis als Workflow Graph

In solch einem Fall bewirkt auch eine größere Menge an Log-Daten keine Verbesserung, denn je mehr Instanzen geändert werden, umso mehr verschiedene Änderungen können auftreten, was das Ergebnis nur noch unübersichtlicher macht. Um die Auswirkungen der einzelnen Änderungen genauer bewerten zu können, werden im Folgenden Log-Daten verwendet, bei deren Erzeugung nur bestimmte Änderungen erlaubt waren.

Abbildung 6-18 zeigt den Änderungsgraph, wenn nur das Einfügen von Aktivitäten erlaubt ist. Weil nur neue Aktivitäten eingefügt werden können und diese in jeder Instanz gleich heissen (*New1-New3*), egal an welcher Stelle im Prozess sie eingefügt werden, kann der *Change Mining Algorithmus* die Abhängigkeiten zwischen den Änderungen korrekt erkennen. So kann die Aktivität ‚*New3*' nur eingefügt werden, nachdem die Aktivität ‚*New2*' eingefügt wurde, während das Einfügen der Aktivität ‚*New1*' unabhängig von den anderen Änderungen ist.

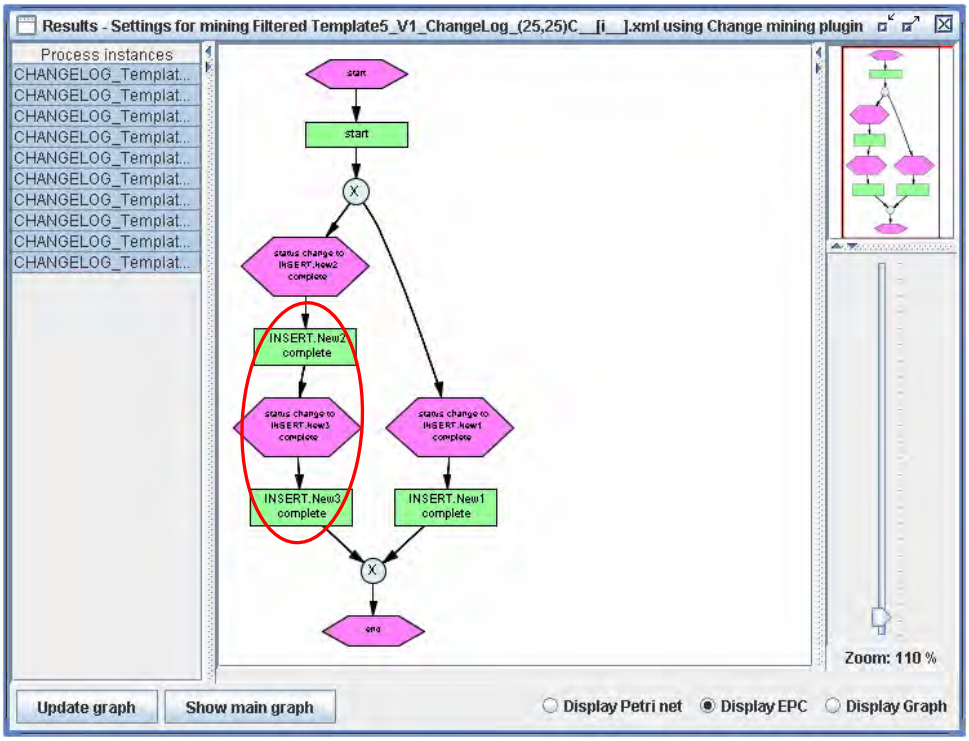


Abbildung 6-18: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 10 individuell geändert (nur Einfügen erlaubt); Ergebnis als EPK

Beim Löschen von Aktivitäten sind die Unterschiede zwischen den einzelnen Instanzen größer. Obwohl im Beispiel nur in sechs (von 25) Instanzen tatsächlich eine Aktivität gelöscht wurde, sind trotzdem drei verschiedene Instanzen gelöscht worden. Da die Aktivitäten alle in unterschiedlichen Instanzen gelöscht wurden gibt es natürlich auch keine Abhängigkeiten zwischen den einzelnen Änderungen, was der Algorithmus korrekt erkennt. Abbildung 6-19 zeigt den Ergebnisgraph dieser Änderungen.

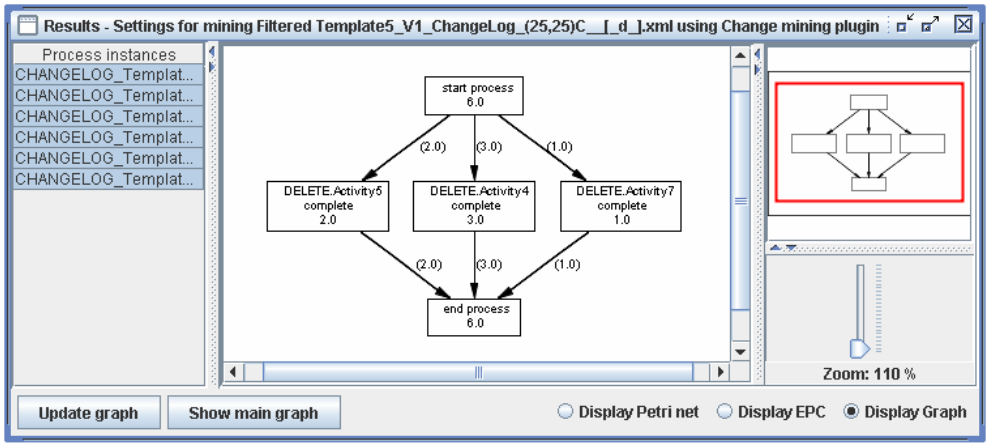


Abbildung 6-19: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 6 individuell geändert (nur Löschen erlaubt); Ergebnis als Workflow Graph

Abbildung 6-20 zeigt den Änderungsgraph wenn nur das Verschieben von Aktivitäten erlaubt ist. Im Beispiel wurden drei (von 25) Instanzen tatsächlich geändert. Weil wiederum bei keiner Instanz mehrere Änderungen vorkommen, gibt es keine Probleme die Abhängigkeiten zwischen den einzelnen Änderungen zu erkennen bzw. zu erkennen, dass es keine Abhängigkeiten gibt.

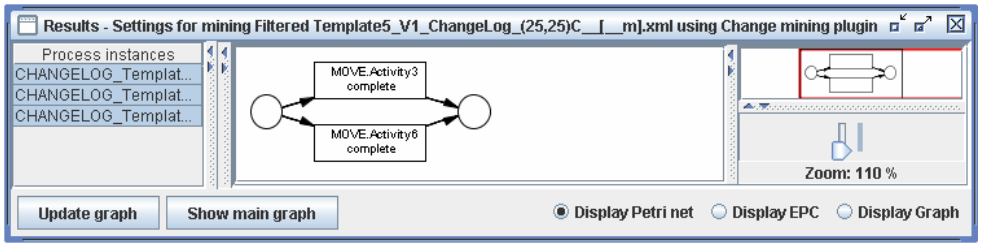


Abbildung 6-20: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 3 individuell geändert (nur Verschieben erlaubt); Ergebnis als Petrinetz

Zusammenfassend kann also gesagt werden, dass die Abhängigkeiten von wenigen einzelnen Änderungen vom *Change Mining Algorithmus* sehr gut erkannt werden können, vor allem wenn nur bestimmte Änderungen erlaubt sind. Aber auch wenn im Log viele verschiedene Änderungen vorkommen erkennt der Algorithmus recht gut, welche Änderungen tatsächlich voneinander abhängen und welche Änderungen parallel ablaufen können, ohne das Endergebnis zu verändern.

Abbildung 6-21 und Abbildung 6-22 zeigen den Änderungsgraphen wenn nur Einfügen und Löschen bzw. nur Einfügen und Verschieben erlaubt ist. Den bisherigen Erkenntnissen entsprechend enthalten auch diese Graphen nichts Neues. Sie zeigen nur noch mal, dass die Abhängigkeiten zwischen Einfügen und Löschen (Aktivität ‚*New4*' in Abbildung 6-21) und die Abhängigkeiten zwischen Einfügen und Verschieben (Aktivität ‚*New5*' in Abbildung 6-22) vom *Change Mining Algorithmus* richtig erkannt werden.

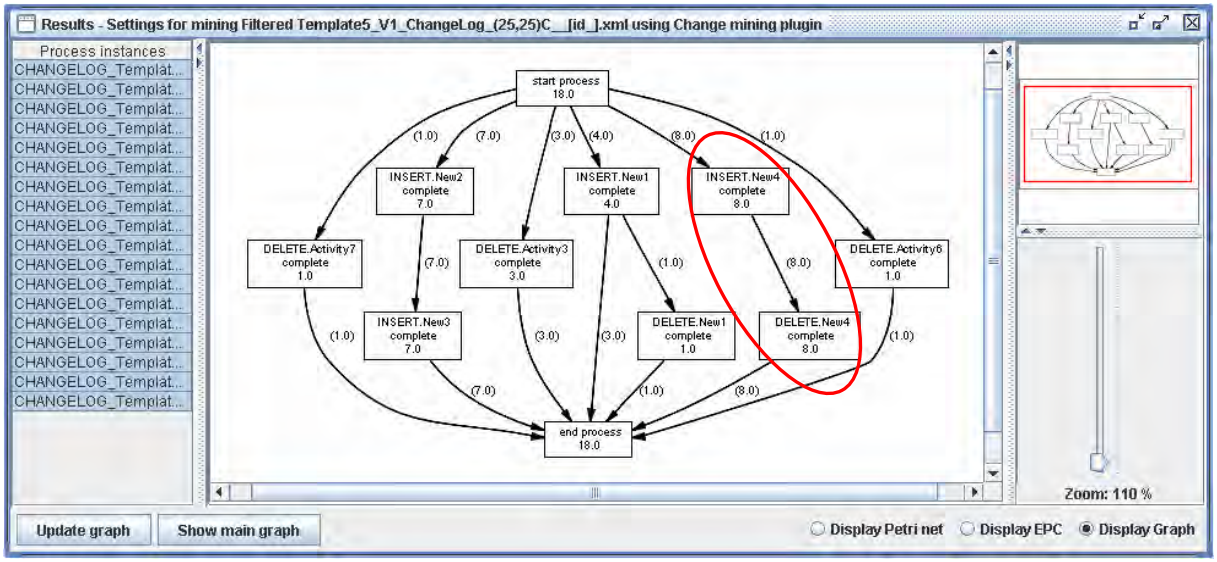


Abbildung 6-21: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 18 individuell geändert (nur Einfügen und Löschen erlaubt); Ergebnis als Workflow Graph

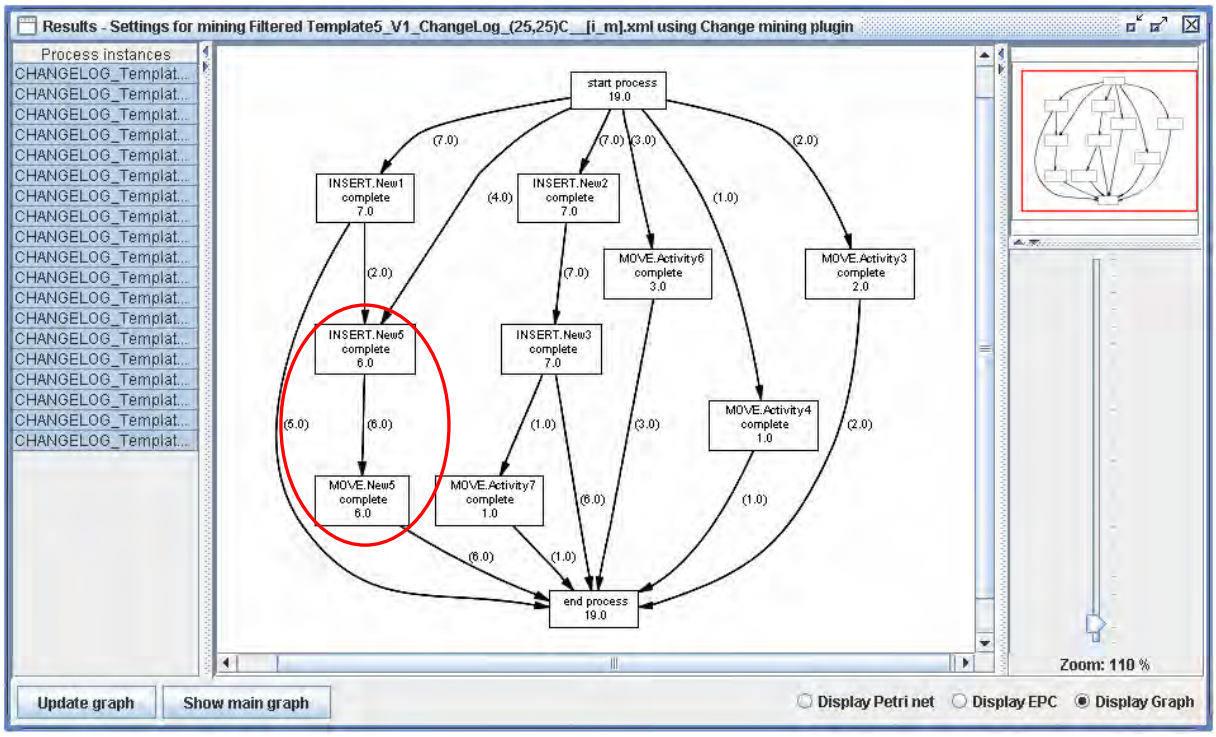


Abbildung 6-22: Beispiel Change Mining: Template5, V1, 25 Instanzen (alle ausgewählt), davon 19 individuell geändert (nur Einfügen und Verschieben erlaubt); Ergebnis als Workflow Graph

Die Betrachtung weiterer Prozesse (‚*Template8, V1*' und ‚*Template10, V1*', siehe auch Kapitel 3.3.5) brachte keine weiteren Erkenntnisse, da auch bei diesen Prozessen die Änderungen nach dem Zufallsprinzip durchgeführt werden und größere Prozesse nur bedeuten, dass es

mehr mögliche Aktivitäten zum Löschen und Verschieben gibt. Aber die Abhängigkeiten zwischen den einzelnen Änderungen, speziell die Abhängigkeiten zwischen den neu eingefügten Aktivitäten, werden ähnlich gut erkannt wie beim Prozess ‚*Template5, V1*' weshalb an dieser Stelle auf weitere Beispielgraphen verzichtet wird.

6.3 Fazit

In diesem Kapitel wurden die Vor- und Nachteile der einzelnen Algorithmen beim *Mining* von Änderungs-Logs noch mal kurz dargelegt. Die für das *Mining* von Ausführungs-Logs benutzten Kriterien (Anzahl Instanzen, Menge *Noise*, Komplexität Prozess) eignen sich nicht (oder nur bedingt) für einen Vergleich der Algorithmen bzgl. *Mining* von Änderungs-Logs. Mit der Anzahl der Instanzen steigt die Anzahl der möglichen unterschiedlichen Änderungsoperationen, die abgelaufen sein können, weshalb ein guter Algorithmus unabhängig von der Anzahl der Instanzen die Abhängigkeiten zwischen den verschiedenen Änderungsoperationen erkennen muss. *Noise* spielt im Zusammenhang mit Änderungs-Logs keine Rolle, da nicht jede Änderung bei jeder Instanz stattfindet und sich die Änderungsgraphen der einzelnen Instanzen deshalb ohnehin unterscheiden. Die Komplexität eines Prozesses beeinflusst das *Mining* von Änderungs-Logs dahingehend, dass durch mehr vorhandene Aktivitäten auch mehr verschiedene Änderungsoperationen möglich sind (z.B. können mehr Aktivitäten gelöscht oder verschoben werden).
Für das *Mining* von Änderungs-Logs und für den Vergleich der Ergebnisse wird nur der zugrunde liegende Prozess unterschieden. Der Prozess ‚*OP-Vorbereitung*' ist ein wohlbekannter Prozess, bei dem von vornherein bestimmt wurde, welche Änderungen stattfinden können. Es gibt genau sechs (logische) Änderungen, die nur genauso auftreten können, wie sie definiert wurden (siehe auch Kapitel 5.2.1). Für beliebige Prozesse kann nicht von vornherein bestimmt werden, welche Änderungen stattfinden sollen, weil der zugrunde liegende Prozess nicht bekannt ist. Deshalb bestimmt eine Zufallsfunktion, welche Aktivitäten gelöscht und verschoben werden (und an welche Stelle Aktivitäten eingefügt werden), weshalb die Anzahl der möglichen Änderungsoperationen deutlich höher ist, als beim Prozess ‚*OP-Vorbereitung*'.

Tabelle 4 zeigt eine Übersicht über die Algorithmen und deren Eigenschaften bezüglich der vorgestellten Kriterien.

Tabelle 4: Übersicht Algorithmen

	Prozess OP-Vorbereitung	Beliebiger Prozess
Alpha-Algorithmus	-	--
Multi-Phase Mining	O	-
Tsinghua-Alpha-Algorithmus	--	--
Heuristics Miner	O	O
Change Mining	++	+

Der *Alpha-Algorithmus* ist für das *Mining* von Änderungs-Logs nur bedingt geeignet. Wenn es nur wenige verschiedene Änderungen gibt (Prozess ‚*OP-Vorbereitung*'), dann wird die sequentielle Abfolge der Änderungen noch ganz gut wiedergegeben, allerdings werden auch Beziehungen zwischen den Änderungen vermutet, die nur durch die Anordnung im Log zustande kommen, tatsächlich aber so nicht gegeben sind. Bei beliebigen Prozessen wird bedingt durch die größere Anzahl unterschiedlicher Änderungen ein Ergebnis präsentiert, das keine Rückschlüsse mehr auf die tatsächlich abhängigen Änderungsoperationen zulässt.

Multi-Phase Mining ist etwas besser für das *Mining* von Änderungs-Logs geeignet, weil die Änderungen, die bei einer einzelnen Instanz stattgefunden haben, gut durch die Instanzgraphen illustriert werden. Allerdings kann auch das *Multi-Phase Mining* nur die sequentielle Abfolge der Änderungen wiedergeben. Bei aggregierten Graphen helfen die Häufigkeitszahlen die bei der Ansicht als Workflow Graph annotiert sind, um die Abhängigkeiten zwischen den Änderungen zu finden, die tatsächlich bestehen. Bei beliebigen Prozessen sind bedingt durch die vielen unterschiedlichen Änderungen viele überflüssige Beziehungen im Graph vorhanden, so dass der Überblick über das Ergebnis schwer fällt.

Für das *Mining* von Änderungs-Logs ist der *Tsinghua-Alpha-Algorithmus* gänzlich ungeeignet, da die gelieferten Ergebnisgraphen keinerlei Rückschlüsse über die Beziehungen zwischen den Änderungsoperationen zulassen.

Der *Heuristics Miner* ist von den bestehenden Algorithmen noch am besten für das *Mining* von Änderungs-Logs geeignet. Auch er liefert ein Ergebnis mit Häufigkeitszahlen, so dass die tatsächlichen Beziehungen zwischen den Änderungen mit einiger Sicherheit bestimmt werden können (kommt eine Beziehung häufig vor, so ist es sehr wahrscheinlich, dass die Änderungen tatsächlich voneinander abhängen). Im Gegensatz zum *Multi-Phase Mining* (und auch zum *Alpha-Algorithmus*) werden beim *Heuristics Miner* nur die häufigsten Kanten gezeigt, so dass schon einige Kanten wegfallen, die bei den anderen Algorithmen das Ergebnis unübersichtlich machen. Auch das Ergebnis bei beliebigen Prozessen ist im Gegensatz zu dem der anderen Algorithmen noch gut zu gebrauchen. Die unabhängigen Änderungsoperationen werden nahezu parallel wiedergegeben, während die tatsächlich abhängigen in einer sequentiellen Abfolge präsentiert werden. Zusätzlich unterstützen einen die Häufigkeitswerte bei der Bestimmung der tatsächlich vorhandenen Abhängigkeiten.

Erwartungsgemäß ist der *Change Mining Algorithmus* am besten bei der Bestimmung der Abhängigkeiten zwischen den Änderungsoperationen. Das Konzept der Kommutativität bringt große Vorteile bei der Erkennung unabhängiger Beziehungen zwischen Änderungen, so dass das Ergebnis deutlich übersichtlicher wird und auch nur die Kanten enthält, die tatsächliche Abhängigkeiten zwischen den Änderungen beschreiben. Darüber hinaus kann auch der *Change Mining Algorithmus* mit Hilfe von Instanzgraphen anzeigen, welche Änderungen bei einer einzelnen Instanz stattgefunden haben. Bei beliebigen Prozessen ist das Ergebnis immer noch gut, durch die größere Anzahl unterschiedlicher Änderungen gibt es aber auch mehr parallele Pfade im Ergebnisgraph, die das Ergebnis etwas unübersichtlicher machen.

Die Ergebnisse zeigen, dass ein erster großer Schritt in Richtung *Mining* von Änderungs-Logs getätigt wurde, aber dass es auch noch weitere Verbesserungsmöglichkeiten gibt. Denkbar wäre z.B., die Erweiterung um das Konzept der Kommutativität auch bei anderen Algorithmen einzubauen (z.B. *Heuristics Miner*), um zu sehen, wie sich deren Ergebnis durch Kom-

mutativität verbessert.
Der *Change Mining Algorithmus* kann auch bestimmte (implizite) Abhängigkeiten zwischen Änderungen nicht erkennen, wenn die Änderungen kommutativ sind. Z.B. wurden am Prozess ‚*OP-Vorbereitung*' zwei logische Änderungsoperationen vorgenommen, erstens das Einfügen von zwei parallelen Schritten (vgl. Änderung2 in Kapitel 5.2.1), zweitens das Einfügen eines zusätzlichen Schrittes nach dem ein anderer gelöscht wurde (vgl. Änderung1 in Kapitel 5.2.1), die bewusst abhängig voneinander sind. Dennoch kann diese Art der Abhängigkeit mit dem *Change Mining Algorithmus* nicht erkannt werden und die Beziehungen zwischen diesen Änderungen werden aus dem Ergebnis entfernt.

Um das *Mining* von Änderungs-Logs weiter erforschen zu können, wurde der im Kontext dieser Arbeit in Zusammenarbeit mit der TU Eindhoven entwickelte *Change Mining Algorithmus* [GRRA06] bereits als Plug-in in *ProM* (Version 3.1) eingebunden. Das Plug-in basiert auf dem *Multi-Phase Mining Algorithmus* [DoAa04] und wurde erweitert, um bei Änderungs-Logs herauszufinden, welche Änderungen voneinander abhängen und welche nicht. Grundsätzlich kann aber jeder *Mining* Algorithmus, der auf der expliziten Erkennung von kausalen Abhängigkeiten basiert, genauso für das *Mining* von Änderungs-Logs erweitert werden.

7 Related Work

Im Folgenden werden einige verwandte Arbeiten aufgeführt. Speziell das Forschungsgebiet *Case-Based Reasoning* erscheint im Kontext dieser Arbeit interessant.

7.1 Adaptives Prozess-Management und Process Mining

Neben den bereits vorgestellten Ansätzen der Universität Ulm [ReDa98, RRD04a, RRD04b, RRD04c, RRD04d, RRJK06] beschäftigt sich z.B. auch Herbst in [Herb98] mit dem Thema Änderungen in Prozessen. Mit *Process Mining* beschäftigen sich neben der Forschungsgruppe um W.M.P. van der Aalst, deren Ergebnisse [ADHM03, AMW05, AWM02, AWM04, DoAa04, MDAW04, MWA04, MWA05, WeAa01, WeAa03] in dieser Arbeit bereits ausführlich betrachtet wurden, auch weitere Forscher. In [GGPS04a, GGPS04b] werden z.B. neben dem *Mining* vom Kontrollfluss auch globale Randbedingungen betrachtet. In [GuOb97] wird der Vergleich zweier Petrinetze, die so genannte Delta-Analyse betrachtet. Und Golani und Pinter beschäftigen sich beispielsweise in [GoPi03] mit Zeitintervallen in Log-Daten.

7.2 Case-based Reasoning

Für eine effektive Unterstützung von Prozessoptimierungsbemühungen benötigen Prozessentwickler auch semantische Informationen über die Gründe von Änderungen. Dies ist insbesondere wichtig, wenn die gleichen oder sich ähnliche Instanzänderungen immer wieder stattfinden. Dialogorientiertes *Case-based Reasoning* (CBR) [ABM01] ist eine Möglichkeit Ausführungs- und Änderungs-Logs mit semantischen Informationen anzureichern. Dialogorientiertes CBR ist eine interaktive Erweiterung des CBR Paradigma [Kolo93].

CBR ist ein zeitgemäßer Ansatz zur Lösung von Problemen und zum Lernen. Neue Probleme werden durch den Bezug auf frühere Erfahrungen behandelt, die in Fällen (*cases*) beschrieben sind. Dazu werden die Lösungen früherer Probleme auf die neue Problemsituation angepasst. Abbildung 7-1 zeigt eine Übersicht zu diesem Vorgang.

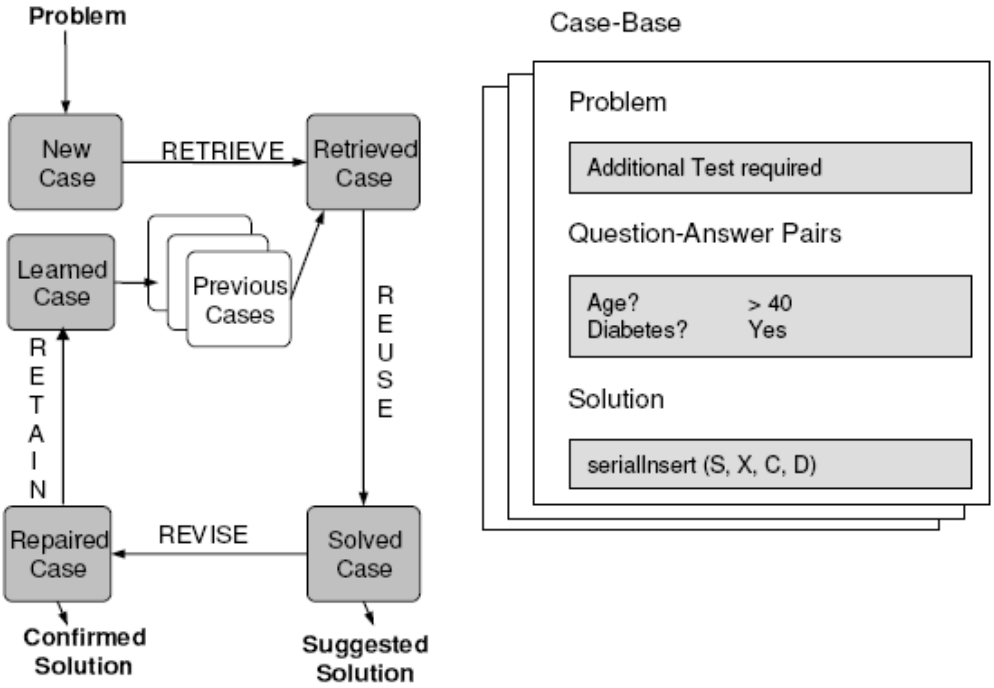


Abbildung 7-1: Case-Based Reasoning [WRRW05]

Das Schlussfolgern basierend auf früheren Erfahrungen ist ein mächtiger und von Menschen häufig angewandter Weg, Probleme zu lösen [AaPl94]. Ein Arzt z.B. benutzt Erfahrungen früherer Fälle um die Krankheit eines (neuen) Patienten zu erkennen. Ein Fall (*case*) ist ein in einen Zusammenhang gesetztes Stück Wissen, dass eine Erfahrung repräsentiert, typischerweise bestehend aus einer Problembeschreibung und der dazugehörigen Lösung [Kolo93].

Die Verwendung von dialogorientiertem CBR, einer Erweiterung zu CBR, ist sinnvoll, wenn der Benutzer aktiv in den Folgerungs-Prozess eingebunden werden soll [AhMu01]. Der Nutzer wird dazu durch eine Frage-Antwort-Reihe geleitet, die in Zusammenhang mit dem Fall steht. Im Gegensatz zu herkömmlichem CBR benötigt dialogorientiertes CBR im Voraus (vom Benutzer) keine komplette Problemspezifikation (um den Fall aufzufinden) und auch kein Wissen über die Bedeutung der einzelnen Merkmale der Problemlösung. Stattdessen unterstützt das System den Benutzer beim Auffinden relevanter Fälle durch die Präsentation einer Reihe von Fragen um die gegebene Situation beurteilen zu können. Dialogorientiertes CBR ist speziell für die Behandlung von ungewöhnlichen und unerwarteten Situationen geeignet, die nicht auf eine vollautomatische Weise behandelt werden können.

Im Zusammenhang mit Prozess-Management-Systemen und *Process Mining* kann dialogbasiertes CBR verwendet werden, um bei Instanzänderungen die Informationen über die Gründe einer Änderung in eine Fall-Datenbank (*case-base*) zu übernehmen, entweder durch hinzufügen eines neuen Falls oder durch Wiederverwenden eines vorhandenen Falls. Ein Fall repräsentiert eine konkrete Ad-Hoc-Änderung einer oder mehrerer Instanzen, er besteht aus einer textuellen Problembeschreibung, einer Reihe von Frage-Antwort-Paaren und einem Lösungsteil. Die Frage-Antwort-Paare beschreiben die Gründe und den Kontext der Ad-Hoc-Abweichung und der Lösungsteil gibt die konkreten Änderungsoperationen wieder. Wenn

gleichartige Ad-Hoc-Änderungen ($\hat{=}$ Fälle) häufig genug auftreten, soll der Prozessentwickler benachrichtigt werden und in der Optimierung des Prozessmodells unterstützt werden. Die Kombination von *Process Mining* und dialogbasiertem CBR zusammen mit adaptivem Prozess-Management erlaubt eine Unterstützung des kompletten Prozesslebenszyklus und die nahtlose Integration der gefundenen Unstimmigkeiten in ein neues, optimiertes Schema.

An der Universität Innsbruck existiert bereits ein Prototyp (*CBRFlow*, [WWB04]), der adaptives Prozess-Management und dialogorientiertes *Case-Based Reasoning* verbindet um die Vorteile von beiden Technologien zu vereinen. Des Weiteren wurden auch die Systeme *ADEPT* und *CBRFlow* zusammen gebracht [WRWR05], um beide Technologien gemeinsam nutzen zu können.

Es könnte als Nachteil gesehen werden, dass die Eingabe von Gründen bei einem dialogbasierten CBR System vom Benutzer erfolgen muss, weil die manuelle Eingabe von Gründen zeitraubend ist und es denkbar wäre, die Gründe auch automatisch zu erfassen, z.B. aus Datenwerten in den Log-Daten. Andererseits gibt es auch Applikationen (z.B. Klinikprozess), bei denen erwünscht ist, dass der Benutzer selbst seine Änderungen dokumentieren kann (warum z.B. eine zusätzliche Untersuchung durchgeführt wurde). Für solche Applikationen ist CBR gut geeignet, weshalb CBR als zusätzliche Technologie gesehen werden kann, die nicht für alle Anwendungsgebiete gleich gut geeignet ist, aber dennoch einen gewissen Nutzen bringt, wenn sie verwendet wird.

8 Zusammenfassung und Ausblick

Im Folgenden werden die Ergebnisse dieser Arbeit noch einmal kurz zusammengefasst und ein Ausblick auf zukünftige Arbeiten gegeben.

8.1 Zusammenfassung

Die Aufgabe dieser Diplomarbeit bestand in der Integration von adaptivem Prozess-Management und *Process Mining*. Ziel einer solchen Integration ist die Unterstützung des Prozessentwicklers bei der Optimierung des Prozessmodells durch eine Analyse der bisher abgelaufenen Änderungen. Dazu wurden zuerst die einzelnen Technologien betrachtet, um einen Einblick in deren Funktionsweise zu erhalten. Adaptives Prozess-Management bietet eine umfassende Unterstützung für Prozessabläufe und Prozessänderungen, sowohl auf Instanz- als auch auf Typebene, aber keine Unterstützung, um zu einem Prozessmodell zu kommen oder von vorhergehenden Änderungen zu lernen. *Process Mining* auf der anderen Seite unterstützt den Anwender bei der Erstellung eines Prozessmodells durch die Rekonstruierung des Prozessmodells aus den Log-Daten bereits abgelaufener Prozesse. Des Weiteren kann *Process Mining* verwendet werden, um rekonstruierte Prozesse mit den ursprünglich modellierten zu vergleichen, um so Abweichungen festzustellen.

Für die Evaluierung der bisherigen *Mining* Möglichkeiten wurde der *ADEPT Demonstrator* erweitert, um die für den Vergleich benötigten Ausführungs-Log-Daten automatisch erzeugen zu können. Die Herausforderung dabei bestand darin, möglichst realistische, also auch fehlerhafte Log-Daten (inkl. *Noise*) zu erzeugen. Die Ergebnisse der Evaluierung wurden verwendet, um die Grundlagen für eine neue Art von *Mining* Verfahren zu schaffen, dem *Mining* von Änderungs-Logs.

Zuerst wurde in Zusammenarbeit mit der TU Eindhoven ermittelt, welche Informationen in einem Änderungs-Log enthalten sein müssen, dann wurde dafür ein Datenmodell vorgestellt. Der *ADEPT Demonstrator* wurde nochmals erweitert, um auch Änderungs-Log-Daten automatisch erzeugen zu können. Die Herausforderung hierbei war, sämtliche Benutzerinteraktionen abzufangen, da Ad-Hoc-Änderungen (Einfügen, Verschieben, Löschen) normalerweise von einem autorisierten Anwender vorgenommen werden.

Mit den erzeugten Log-Daten wurden nun erstmal bereits bestehende *Mining* Verfahren getestet. Anhand der dadurch gewonnenen Erkenntnisse wurde ein *Mining* Algorithmus speziell für das *Mining* von Änderungs-Logs entwickelt, der so genannte *Change Mining Algorithmus*. Dieser Algorithmus wurde ebenfalls in Zusammenarbeit mit der TU Eindhoven erarbeitet und im Rahmen dieser Arbeit auch in *ProM* eingebunden.

Der *Change Mining Algorithmus* basiert auf einem bereits bestehenden Algorithmus (*Multi-Phase Mining*), weshalb die *Mining* Ergebnisse bestehender Algorithmen mit denen des neuen Algorithmus verglichen wurden. Hierbei zeigte sich, dass der neue Algorithmus die Abhängigkeiten zwischen verschiedenen Änderungen aufgrund des verwendeten Konzepts der Kommutativität deutlich besser erkennen kann, als die bisherigen Verfahren. Dennoch gibt es noch einige Verbesserungsmöglichkeiten, so können z.B. implizite Abhängigkeiten vom

Change Mining Algorithmus nicht erkannt werden. Ebenfalls denkbar wäre es, auch andere Algorithmen um das Konzept der Kommutativität zu erweitern.

8.2 Ausblick

Auch zusammen bieten adaptives Prozess-Management und *Process Mining* noch keine komplette Unterstützung für den Lebenszyklus eines Prozesses. Deshalb ist in zukünftigen Arbeiten die Integration weiterer Technologien denkbar. Besonders interessant erscheint in dieser Hinsicht das Thema *Decision Mining* [RoAa06]. Bisher wird diese Technik dazu verwendet durch klassisches *Process Mining* gewonnene Petrinetze und deren Log-Daten zu analysieren, um zu bestimmen, wie sich Datenwerte auf die Entscheidungen (z.B. ODER-Verzweigungen) in einem Prozess auswirken. In einem Krankenhausprozess ist zum Beispiel denkbar, dass gewisse Untersuchungen nur stattfinden, wenn bestimmte Voraussetzungen erfüllt sind (z.B. Patient Raucher, über 45). *Decision Mining* wird dazu verwendet, genau diese Datenabhängigkeiten (Voraussetzungen) bei unbekannten Prozessen herauszufinden bzw. um zu erkennen, wie bestimmte Datenwerte die Pfadbestimmung einer Instanz beeinflussen. *Decision Mining* ist in dieser Form bereits umgesetzt und in *ProM* eingebunden.

Im Umfeld von Änderungs-Logs könnte *Decision Mining* die mit dem *Change Mining Algorithmus* ermittelten Änderungsprozesse (siehe Abschnitt 6.2) als Grundlage verwenden und dann ähnlich wie in [RoAa06] beschrieben die Entscheidungsgrundlagen an XOR-Verzweigungen ermitteln. Bei Änderungsprozessen entsprechen die XOR-Verzweigungen den Gründen für oder gegen eine Änderung. Werden z.B. bestimmte Änderungen an Instanzen immer dann durchgeführt, wenn bestimmte Voraussetzungen erfüllt sind (z.B. Patient zuckerkrank, unter 35), dann soll *Decision Mining* dazu verwendet werden, diese Gründe für eine Änderung herauszufinden. Die Herausforderung dabei ist, dass die erforderlichen Datenwerte nicht in den Änderungs-Logs stehen, sondern in den dazugehörigen Ausführungs-Logs. Deshalb müssen die Log-Daten zusammen betrachtet werden, um ein entsprechendes Ergebnis erhalten zu können.

Neben der Integration von *Decision Mining* ist auch die Integration von *Case-Based Reasoning* (CBR; siehe Kapitel 7.2) in ein Rahmenwerk für eine umfassende Prozessunterstützung denkbar. In solch einem Rahmenwerk kann CBR für die Aufzeichnung der Gründe einer Änderung benutzt werden. Wird eine bestimmte Änderung nun oft mit der gleichen Begründung durchgeführt, dann kann eine Prozessoptimierung angestoßen werden.

Literaturverzeichnis

[AaDo02] **van der Aalst, W.M.P., van Dongen, B.:** *Discovering Workflow Performance Models from Timed Logs*. In: International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002). LNCS 2480, pp. 45-63, 2002.
(http://www.processmining.org)

[AaPl94] **Aamodt, A., Plaza, E.:** *Case-Based Reasoning: Foundational Issues, Methodological Variations and System Approaches*. AI Communications 7, pp. 39-59, 1994.

[AaSo04] **van der Aalst, W.M.P., Song, M.:** *Mining Social Networks: Uncovering Interaction Patterns in Business Processes*. International Conference on Business Process Management (BPM 2004), J. Desel, B. Pernici, M. Weske, LNCS 3080, pp. 244-260, 2004.
(http://www.processmining.org)

[ABM01] **Aha, D.W., Breslow, L., Munoz-Avilla, H.:** *Conversational Case-Based Reasoning*. Applied Intelligence 14, pp. 9-32, 2001.

[ADHM03] **van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.:** *Workflow Mining: A Survey of Issues and Approaches*. Data and Knowledge Engineering, 47(2): pp. 237-267, 2003.
(http://www.processmining.org)

[AhMu01] **Aha, D.W., Munoz-Avilla, H.:** *Introduction: Interactive Case-Based Reasoning*. Applied Intelligence 14, pp. 7-8, 2001.

[AMW05] **van der Aalst, W.M.P., Medeiros, A.K.A., Weijters, A.J.M.M.:** *Genetic Process Mining*. 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005). G. Ciardo, P. Darondeau, LNCS 3536, pp. 48-69, 2005.
(http://www.processmining.org)

[AWM02] **van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.:** *Workflow Mining: Which processes can be rediscovered?* BETA Working Papers Series, WP 75, Eindhoven University of Technology, Eindhoven, 2002.
(http://www.processmining.org)

[AWM04] **van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.:** *Workflow Mining: Discovering Process Models from Event Logs*. IEEE Transactions on Knowledge and Data Engineering (TKDE), 16(9): pp. 1128-1142, 2004.
(http://www.processmining.org)

[DMVW05] **van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.:** *The ProM framework: A new era in process mining tool support*. 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005). G. Ciardo, P. Darondeau, LNCS

3536, pp. 444-454, 2005.
(http://www.processmining.org)

[DoAa04] **van Dongen, B.F., van der Aalst, W.M.P.:** *Multi-phase Process Mining: Building instance graphs*. Conceptual Modeling – ER 2004, P. Atenzi et al. (eds.): LNCS 3288, pp. 362-376, 2004.
(http://www.processmining.org)

[GBG05] **Gaaloul, W., Baina, K., Godart, C.:** *Towards Mining Structural Workflow Patterns*. In: 16th International Conference on Database and Expert Systems Applications DEXA'05, Copenhagen Denmark, 2005.

[GGPS04a] **Greco, G., Guzzo, A., Pontieri, L., Sacca, D.:** *Mining Expressive Process Models by Clustering Workflow Traces*. Proceedings 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-04), Sydney, Australia, 2004.

[GGPS04b] **Greco, G., Guzzo, A., Pontieri, L., Sacca, D.:** *On the Mining of Complex Workflow Schemas*. Proceedings Dodicesimo Convegno Nazionale su Sistemi Evoluti per Basi di Dati (SEBD04), Cagliari, Italien, 2004.

[GoPi03] **Golani, M., Pinter, S.S.:** *Generating a Process Model from a Process Audit Log*. In: Proceedings of International Conference on Business Process Management (BPM'03), Eindhoven, pp. 136-151, 2003.

[GRRA06] **Günther, C.W., Rinderle, S., Reichert, M., van der Aalst, W.M.P.:** *Change Mining in Adaptive Process Management Systems*. CoopIS 2006, Montpellier, 2006. (accepted for publication)

[GuOb97] **Guth, V., Oberweis, A.:** *Delta Analysis of Petri Net based Models for Business Processes*. In: Proceedings of International Conference on Applied Informatics (1997), pp. 23-32, 1997.

[Herb98] **Herbst, J.:** *An inductive Approach to Adaptive Workflow Systems*. In: Proc. Workshop Towards Adaptive Workflow Systems (CSCW'98), Seattle, 1998.

[Kolo93] **Kolodner, J.L.:** *Case-Based Reasoning*. Morgan Kaufmann, 1993.

[Laue04] **Lauer, M.:** *Effiziente Realisierung von Prozess-Schemaevolution in Hochleistungs-Prozess-Management-Systemen*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[MAW03] **de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.:** *Workflow Mining: Current Status and Future Directions*. In: R. Meersman et al. (eds.): CoopIS/DOA/ODBASE 2003, LNCS 2888, pp. 389-406, 2003.
(http://www.processmining.org)

[MDAW04] **de Medeiros, A.K.A., van Dongen, B.F., van der Aalst, W.M.P., Weijters, A.J.M.M.:** *Process Mining: Extending the alpha-algorithm to Mine Short Loops*. BETA Working Paper Series, WP 113, Eindhoven University

of Technology, Eindhoven, 2004.
(http://www.processmining.org)

[MWA04] **de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.:** *Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results*. BETA Working Paper Series, WP 124, Eindhoven University of Technology, Eindhoven, 2004.
(http://www.processmining.org)

[MWA05] **de Medeiros, A.K.A., Weijters, A.J.M.M., van der Aalst, W.M.P.:** *Genetic Process Mining: A Basic Approach and its Challenges*. Workshop on Business Process Intelligence (BPI), Nancy, 2005.
(http://www.processmining.org)

[MWAB02] **Maruster, L., Weijters, A.J.M.M., van der Aalst, W.M.P., van den Bosch, A.:** *Process Mining: Discovering direct successors in process logs*. In: Proceedings of the 5th International Conference on Discovery Science (Discovery Science 2002), LNAI 2534, pp. 364-373, 2002.
(http://www.processmining.org)

[ReDa98] **Reichert, M., Dadam, P.:** *$ADEPT_{flex}$ – Supporting Dynamic Changes of Workflows Without Loosing Control*. Journal of Intelligent Information Systems 10, pp. 93-129, 1998.

[RoAa06] **Rozinat, A., van der Aalst, W.M.P.:** *Decision Mining in Business Processes*. BPM Center Report BPM 06-10, BPMcenter.org, 2006.
(http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-10.pdf)

[RRD04a] **Rinderle, S., Reichert, M., Dadam, P.:** *Flexible Support of Team Processes by Adaptive Workflow Systems*. Distributed and Parallel Databases 16, pp. 91-116, 2004.

[RRD04b] **Rinderle, S., Reichert, M., Dadam, P.:** *On Dealing with Structural Conflicts between Process Type and Instance Changes*. In: Proceedings of International Conference on Business Process Management (BPM'04), Potsdam, pp. 274-289, 2004.

[RRD04c] **Rinderle, S., Reichert, M., Dadam, P.:** *Correctness criteria for dynamic changes in workflow systems – a survey*. Data and Knowledge Engineering 50, pp. 9-34, 2004.

[RRD04d] **Rinderle, S., Reichert, M., Dadam, P.:** *Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications*. In: R. Meersman et al. (eds.): CoopIS/DOA/ODBASE 2004, LNCS 3290, pp. 101-120, 2004.

[RRJK06] **Rinderle, S., Reichert, M., Jurisch, M., Kreher, U.:** *On Representing, Purging, and Utilizing Change Logs in Process Management Systems*. In Proceedings of International Conference on Business Process Management (BPM'06), Wien, 2006.

[WeAa01] **Weijters, A.J.M.M., van der Aalst, W.M.P.:** *Process Mining: Discovering Workflow Models from Event-Based Data*. Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001): pp. 283-290, 2001.
(http://www.processmining.org)

[WeAa03] **Weijters, A.J.M.M., van der Aalst, W.M.P.:** *Rediscovering Workflow Models from Event-Based Data using Little Thumb.* Integrated Computer-Aided Engineering, Vol. 10/2: pp. 151-162, 2003.
(http://is.tm.tue.nl/staff/aweijters/ICAE_v5.pdf)

[WRRW05] **Weber, B., Reichert, M., Rinderle, S., Wild, W.:** *Towards a Framework for the Agile Mining of Business Processes*. In: C. Bussler et al. (eds.): BPM 2005 Workshops (1st Int. Workshop on Business Process Intelligence (BPI 2005)), Nancy, France, September 2005, LNCS 3812: pp. 191-202.
(http://www.informatik.uni-ulm.de/dbis/01/dbis/downloads/WRRW05.pdf)

[WRWR05] **Weber, B., Rinderle, S., Wild, W., Reichert, M.:** *CCBR-Driven Business Process Evolution*. In: Proceedings of International Conference on Case-Based Reasoning (ICCBR'05), Chicago, pp. 610-624, 2005.

[WWAW04] **Wen, L., Wang, J., van der Aalst, W.M.P., Wang, Z., Sun, J.:** *A Novel Approach for Process Mining Based on Event Types.* BETA Working Paper Series, WP 118, Eindhoven University of Technology, Eindhoven, 2004.
(http://is.tm.tue.nl/staff/wvdaalst/publications/p232.pdf)

[WWB04] **Weber, B., Wild, W., Breu, R.:** *CBRFlow: Enabling Adaptive Workflow Mangement Through Conversational Case-Based Reasoning*. In: P. Funk et al. (eds.): Proceedings of European Conference on Case-Based Reasoning (ECCBR'04) Madrid, LNAI 3155, pp. 434-448, 2004.

[WWS06] **Wen, L., Wang, J., Sun, J.:** *Detecting Implicit Dependencies Between Tasks from Event Logs*. In: X. Zhou et al. (eds.): APWeb 2006, LNCS 3841, pp. 591-603, 2006.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

A Übersicht Beispielprozesse

Im Folgenden sind alle verwendeten Beispielprozesse aufgeführt. Die erste Abbildung (a) zeigt den Prozess so, wie er im *ADEPT Demonstrator* angezeigt wird, die zweite Abbildung (b) zeigt den Prozess so, wie er nach dem *Mining* eines fehlerlosen Log-Datensatzes (mit 50 Instanzen) beim *Tsinghua-Alpha-Algorithmus* angezeigt wird. Die einzelnen Prozesse unterscheiden sich in ihrer Komplexität und in den verwendeten Konstrukten (UND-Block, ODER-Block, Synchronisations-Kanten).

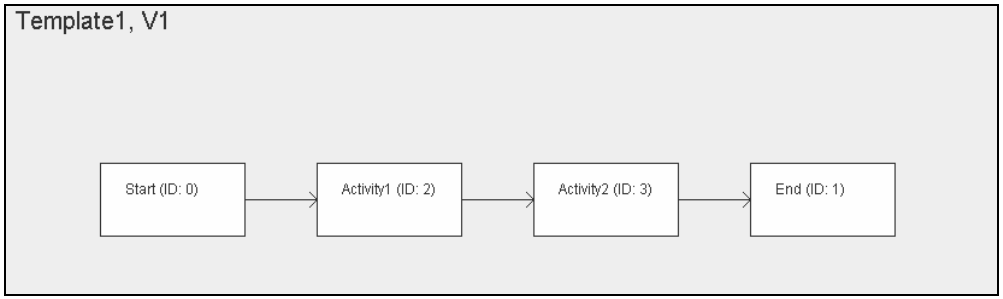


Abbildung A-1a: Template1, V1 im Demonstrator

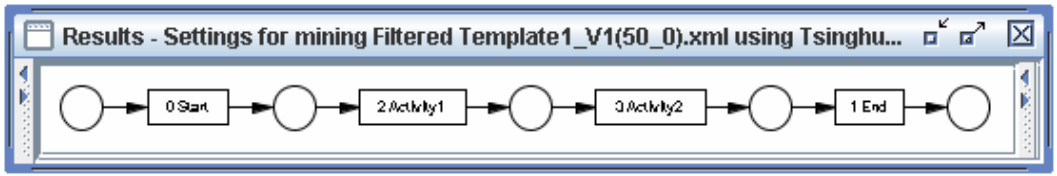


Abbildung A-1b: Template1, V1 in ProM (Tsinghua-Alpha-Algorithmus)

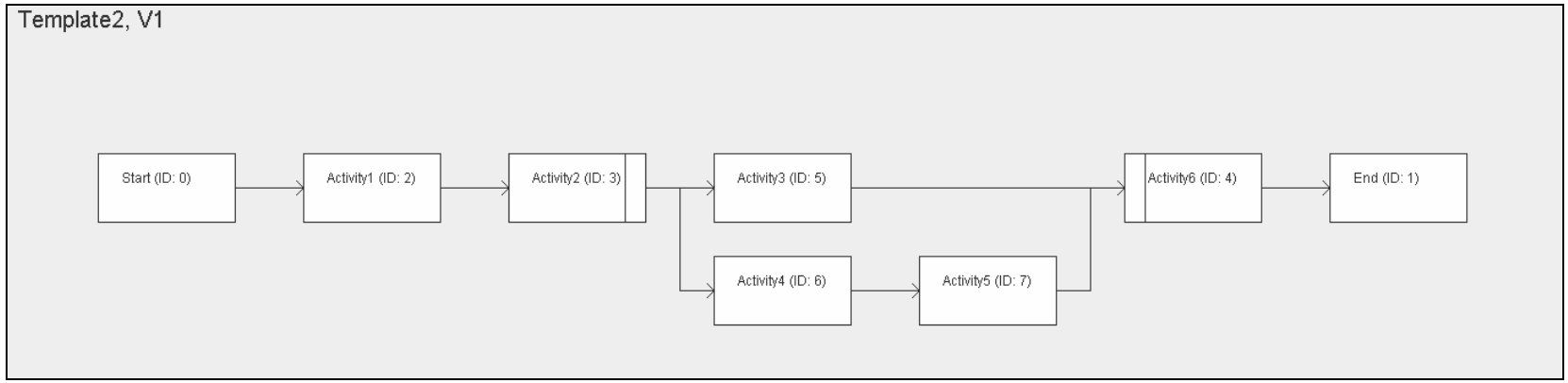


Abbildung A-2a: Template2, V1 im Demonstrator

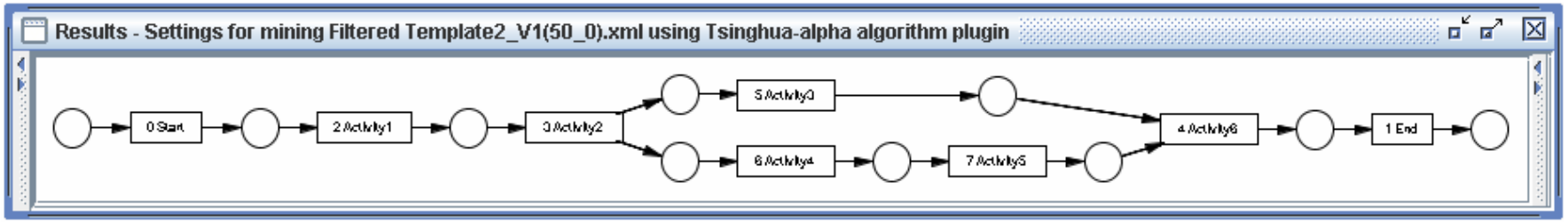


Abbildung A-2b: Template2, V1 in ProM (Tsinghua-Alpha-Algorithmus)

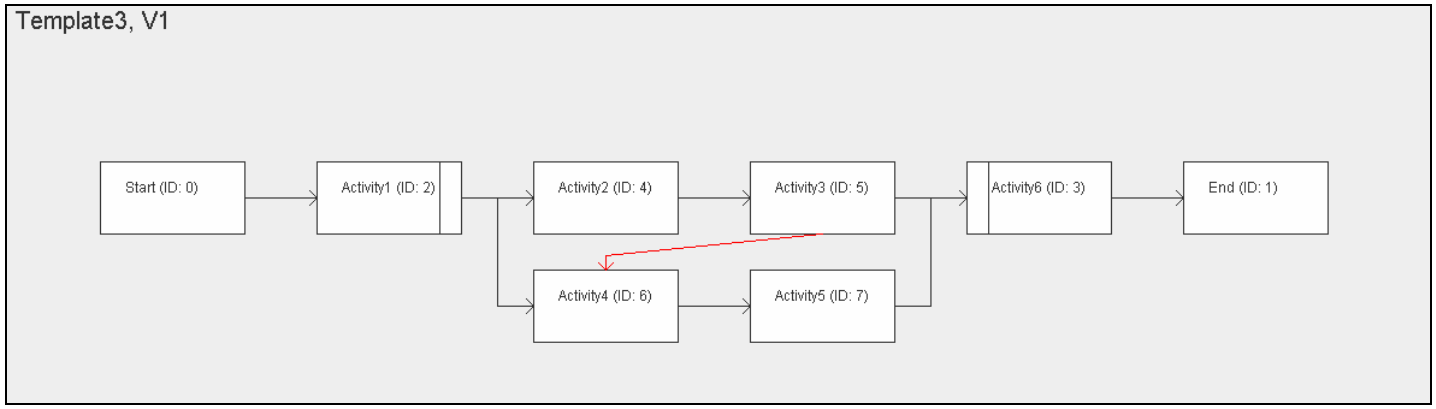


Abbildung A-3a: Template3, V1 im Demonstrator

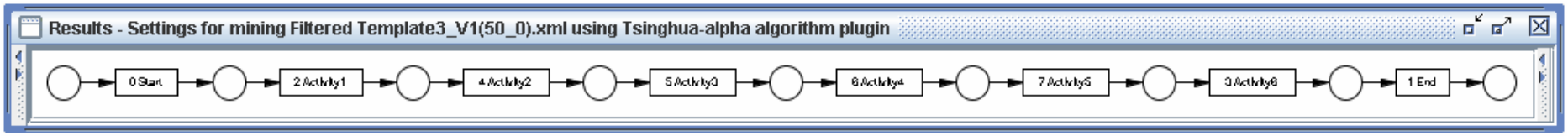


Abbildung A-3b: Template3, V1 in ProM (Tsinghua-Alpha-Algorithmus)

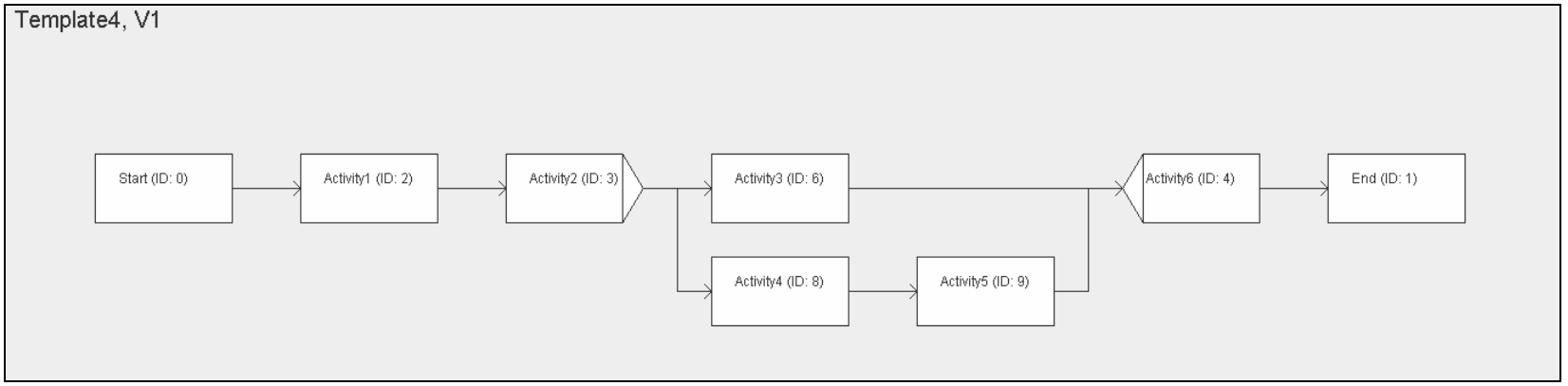


Abbildung A-4a: Temlpate4, V1 im Demonstrator

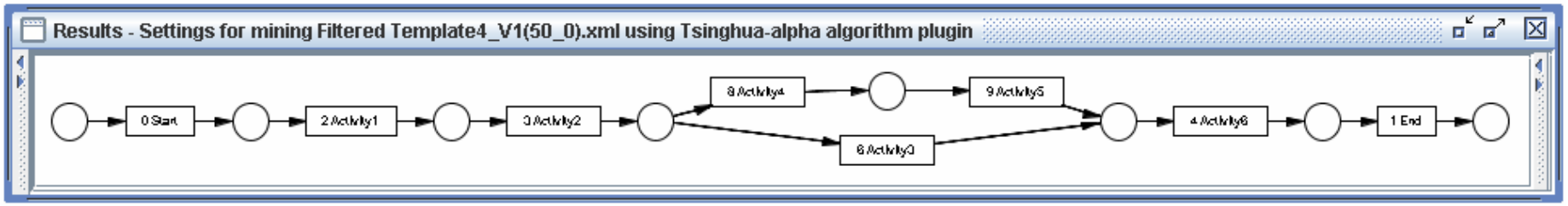


Abbildung A-4b: Template4, V1 in ProM (Tsinghua-Alpha-Algorithmus)

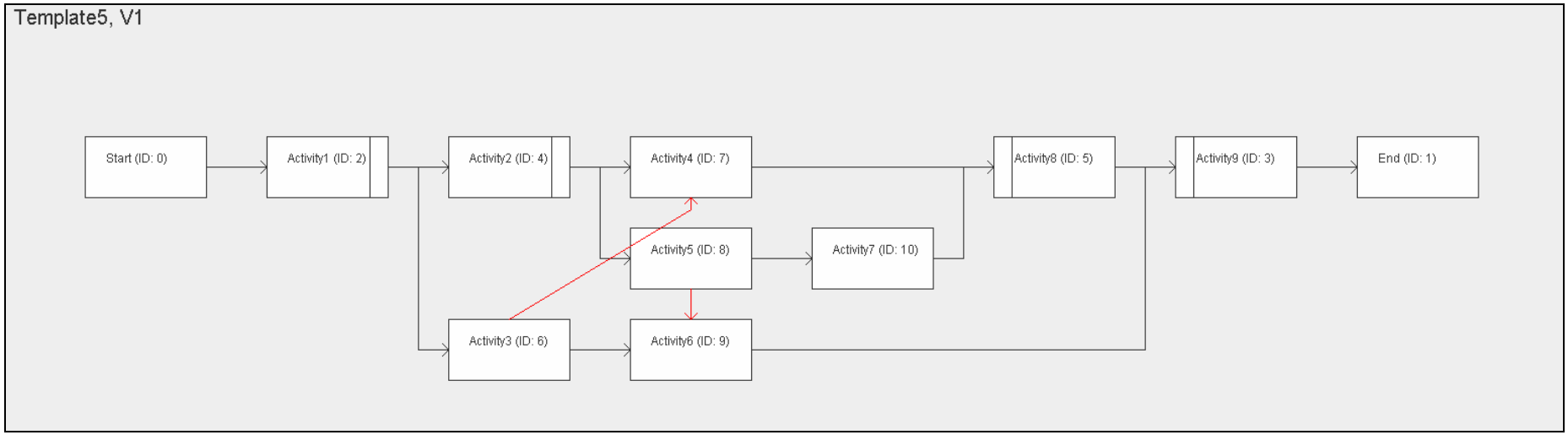


Abbildung A-5a: Template5, V1 im Demonstrator

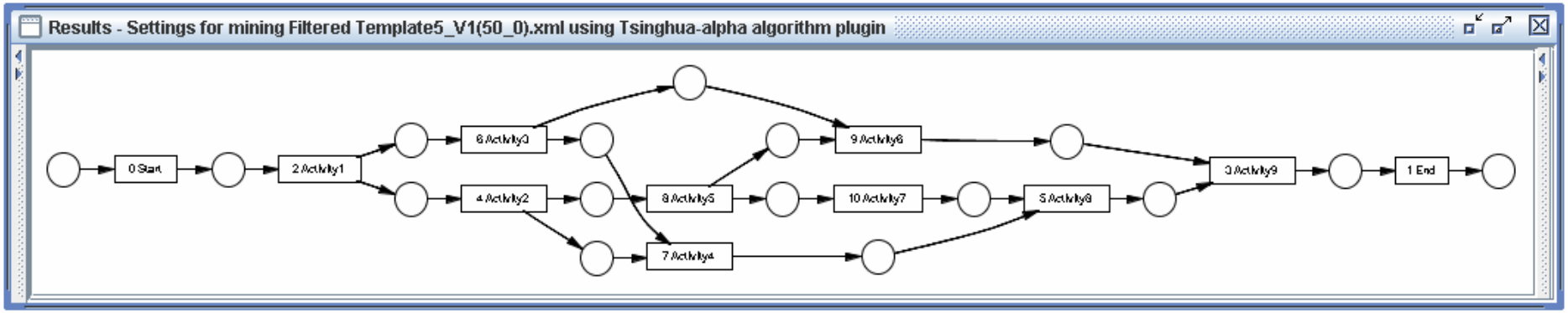


Abbildung A-5b: Template5, V1 in ProM (Tsinghua-Alpha-Algorithmus)

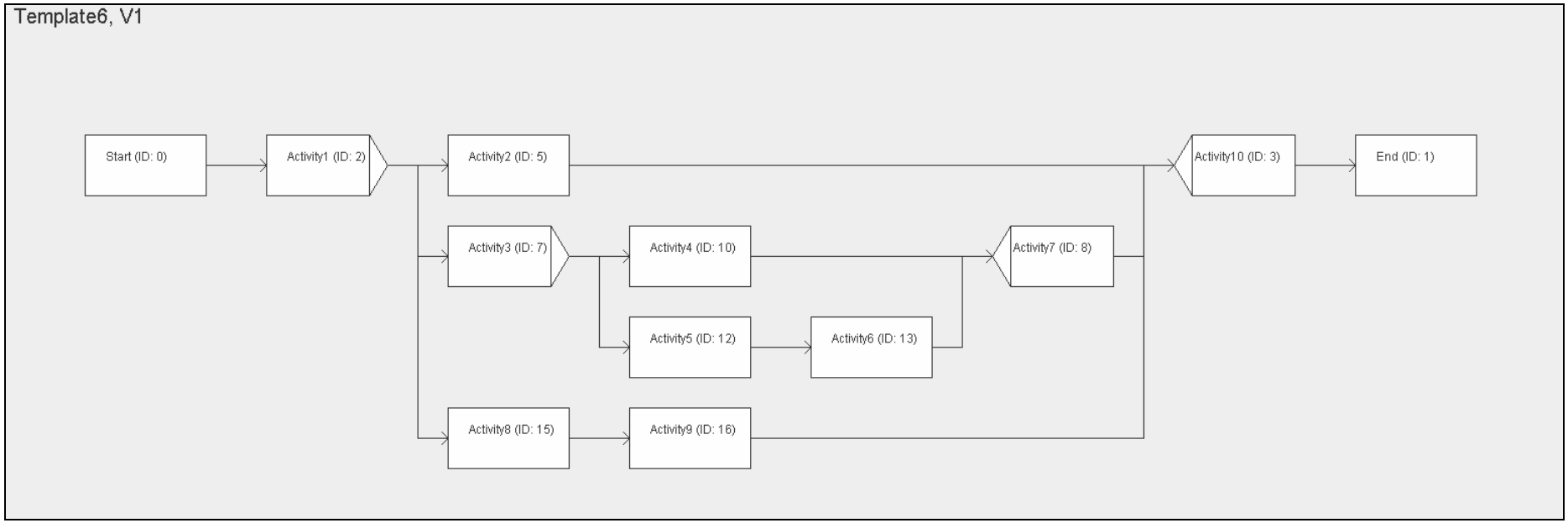


Abbildung A-6a: Template6, V1 im Demonstrator

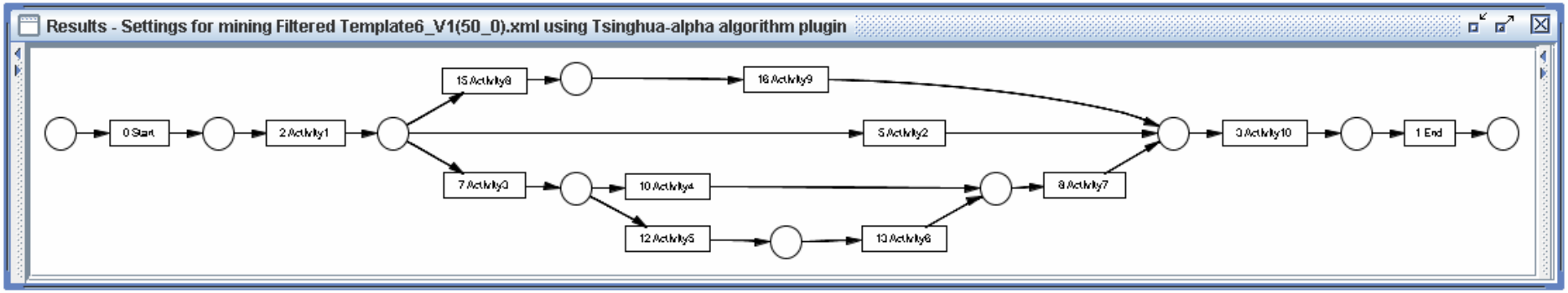


Abbildung A-6b: Template6, V1 in ProM (Tsinghua-Alpha-Algorithmus)

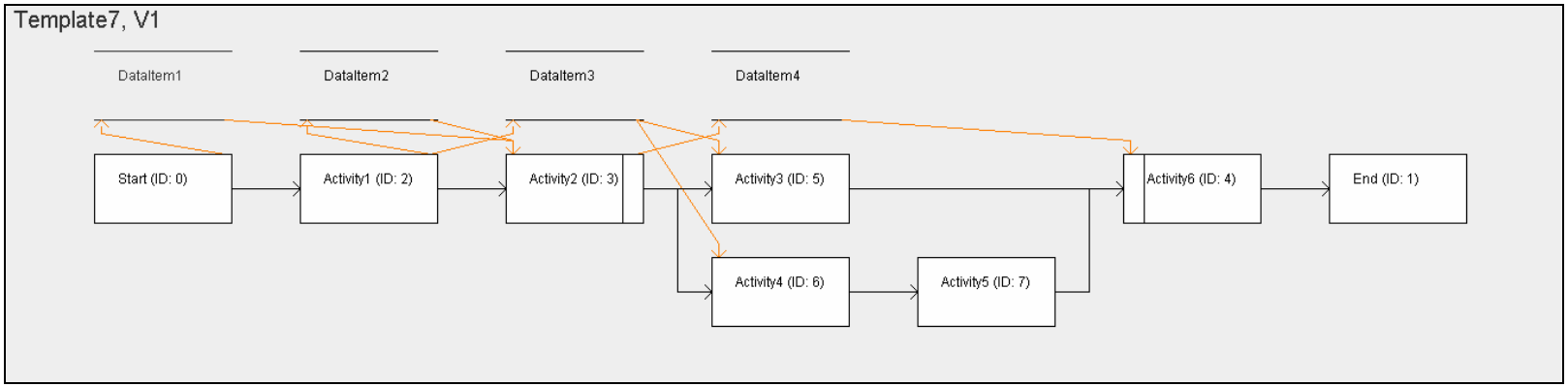


Abbildung A-7a: Template7, V1 im Demonstrator

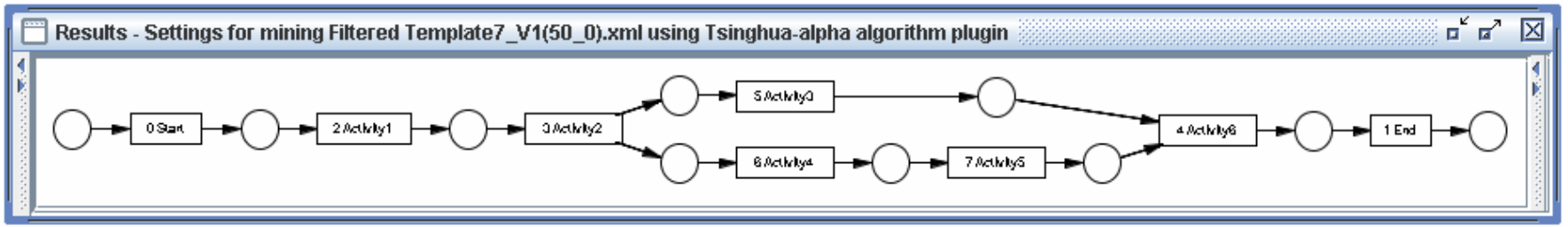


Abbildung A-7b: Template7, V1 in ProM (Tsinghua-Alpha-Algorithmus)

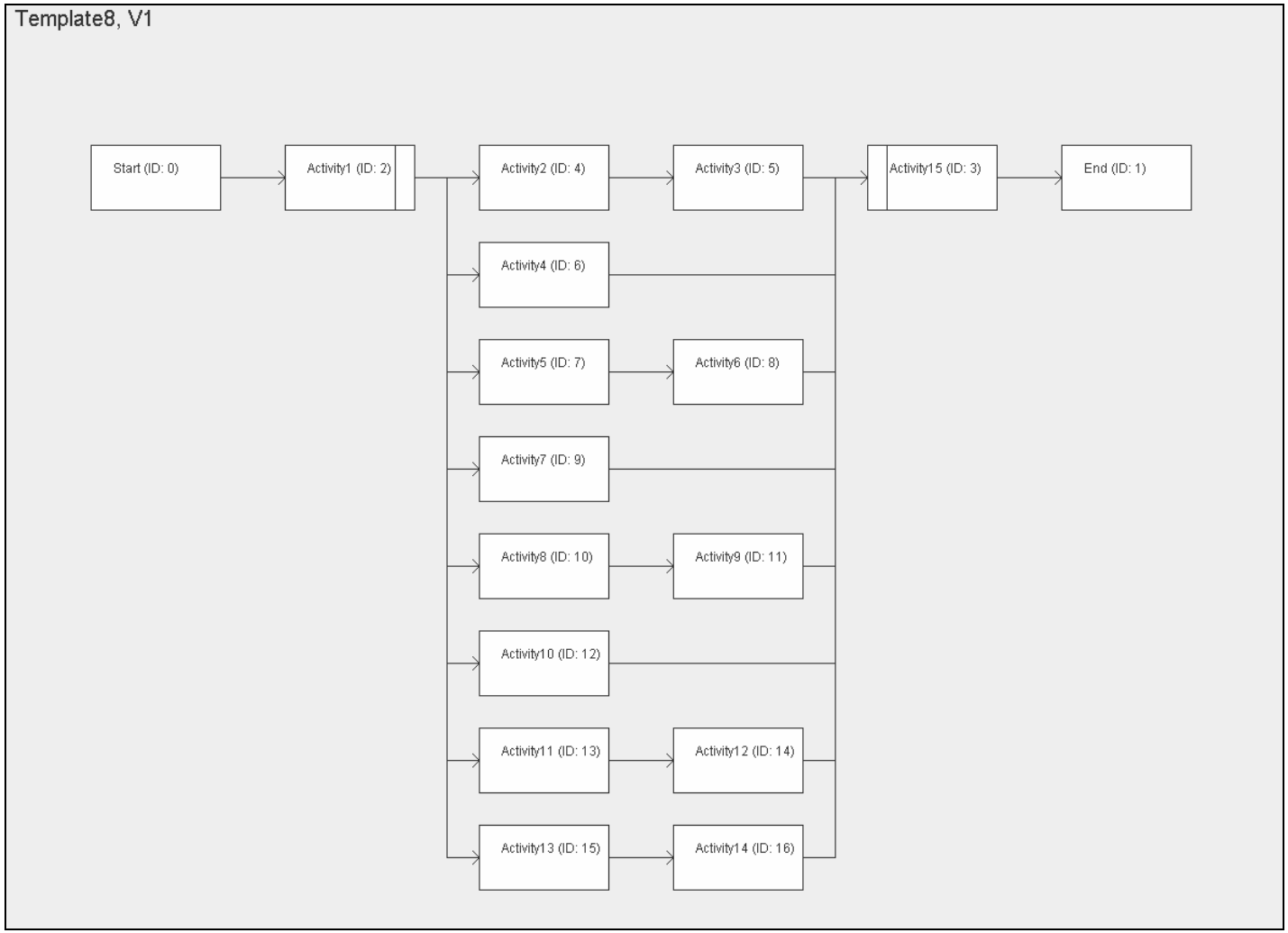


Abbildung A-8a: Template8, V1 im Demonstrator

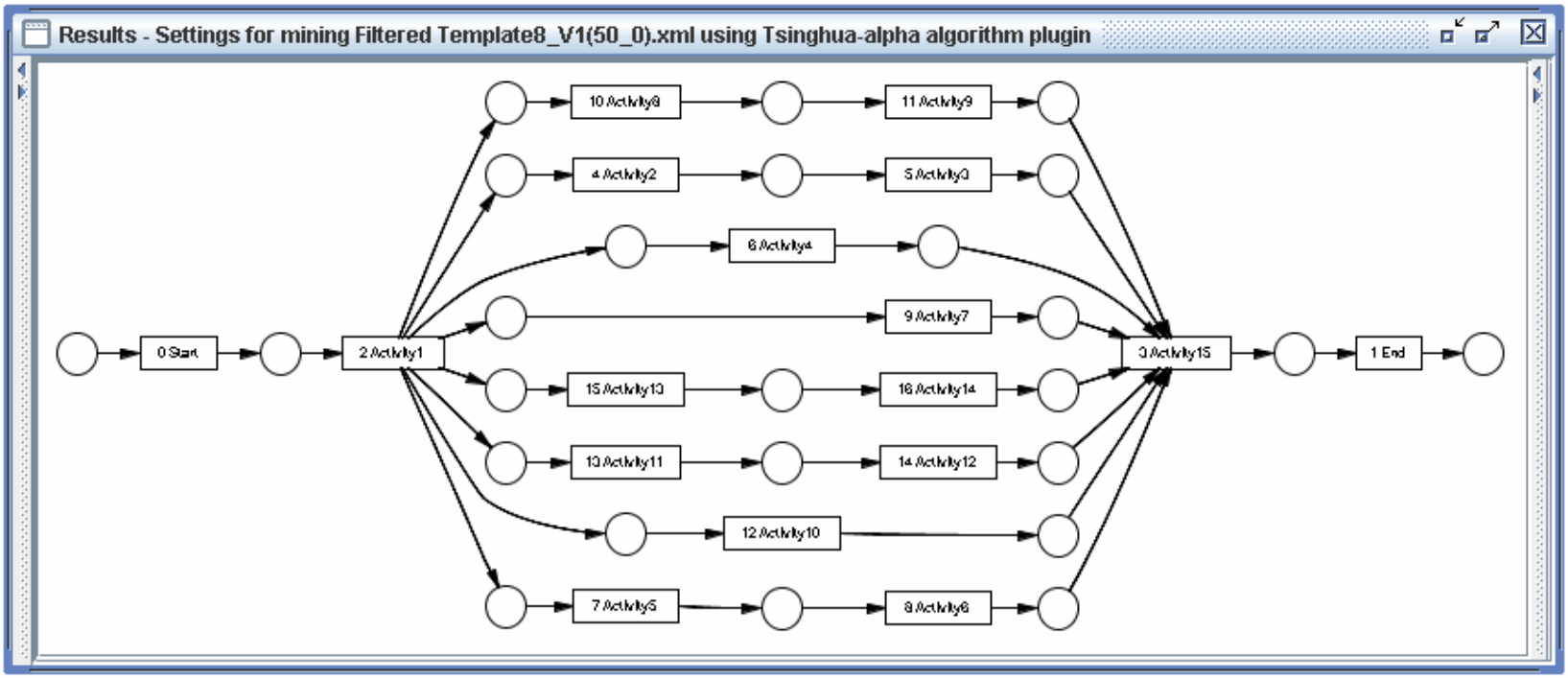


Abbildung A-8b: Template8, V1 in ProM (Tsinghua-Alpha-Algorithmus)

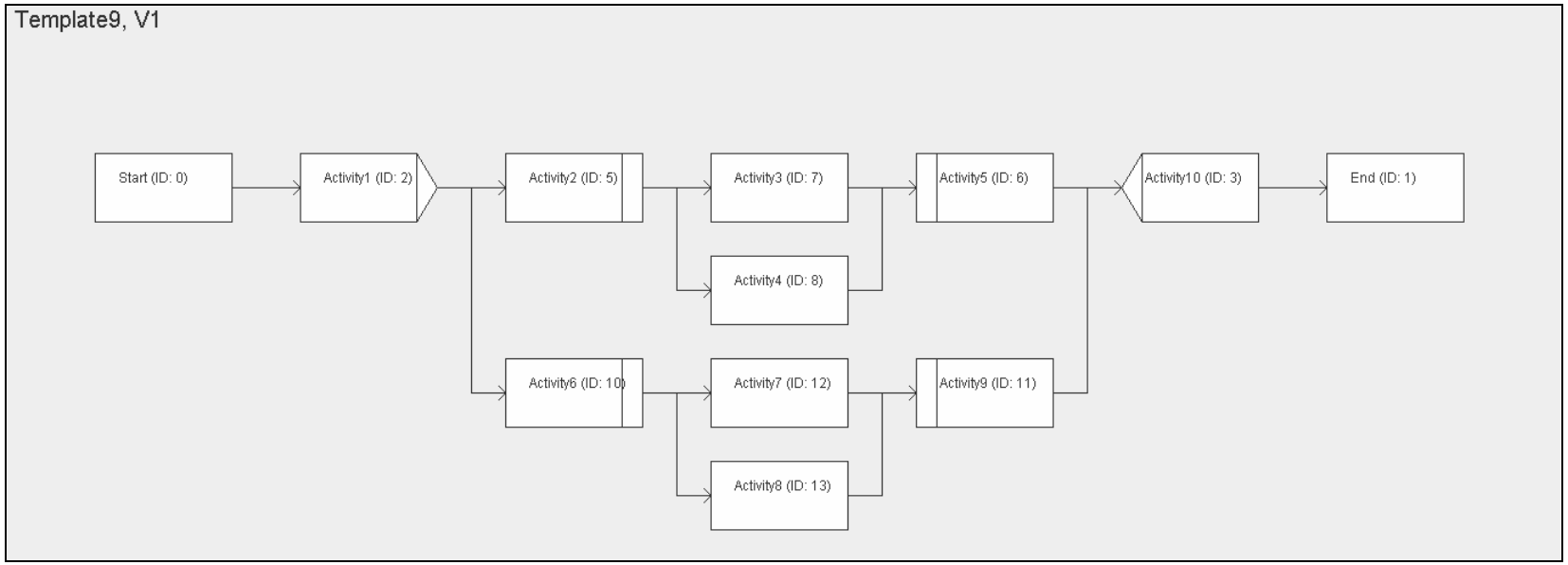


Abbildung A-9a: Template9, V1 im Demonstrator

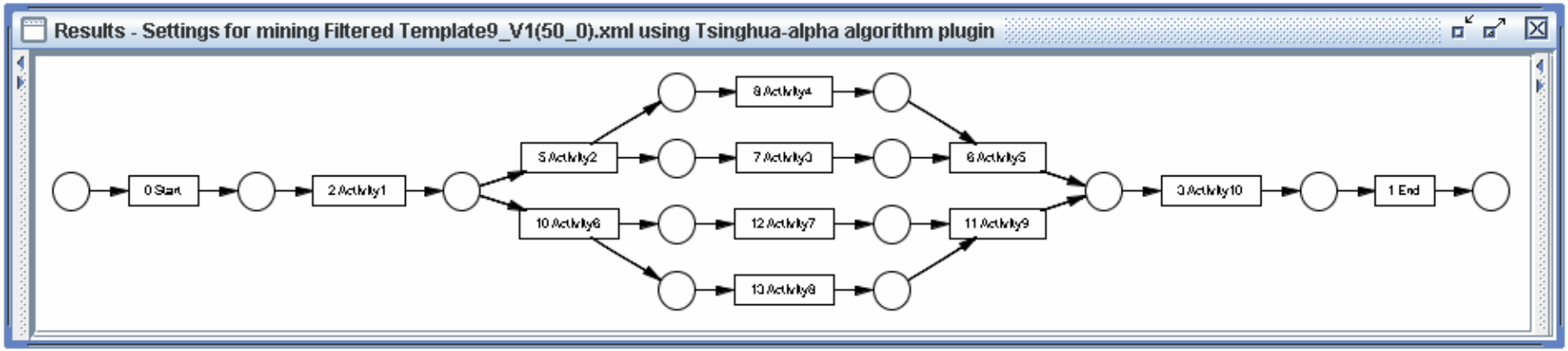


Abbildung A-9b: Template9, V1 in ProM (Tsinghua-Alpha-Algorithmus)

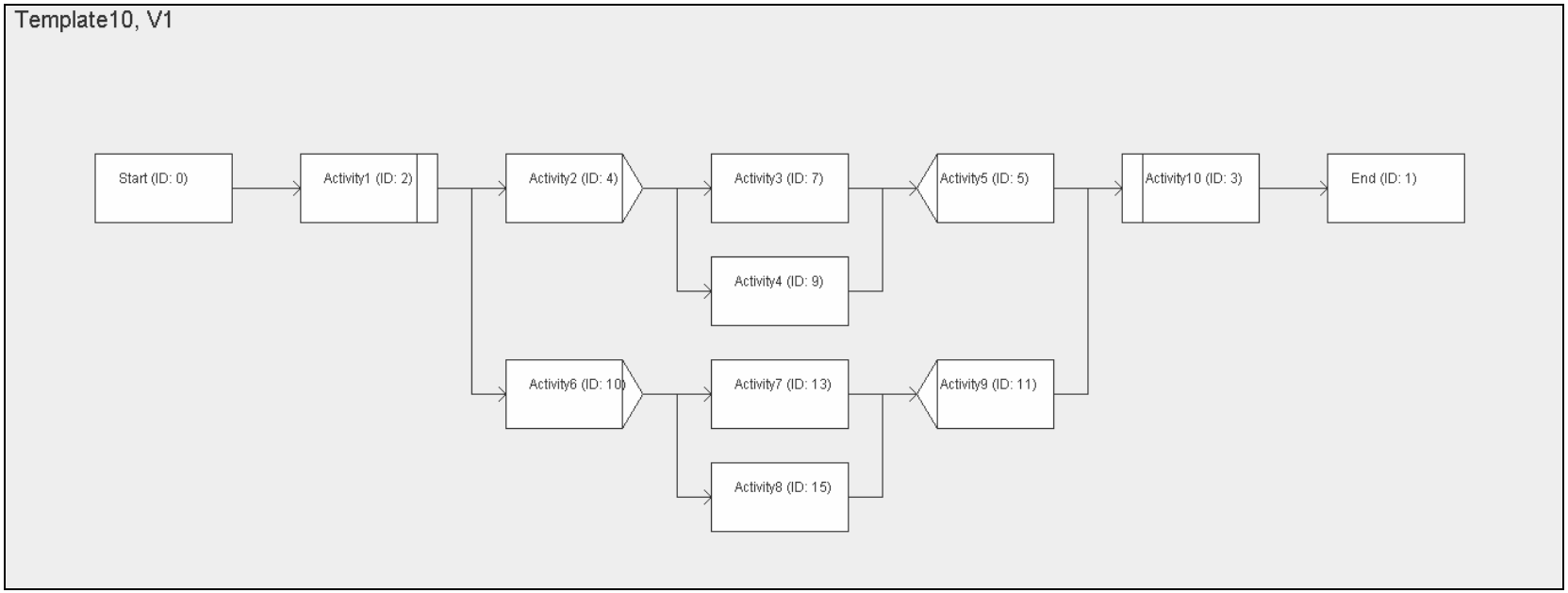


Abbildung A-10a: Template10, V1 im Demonstrator

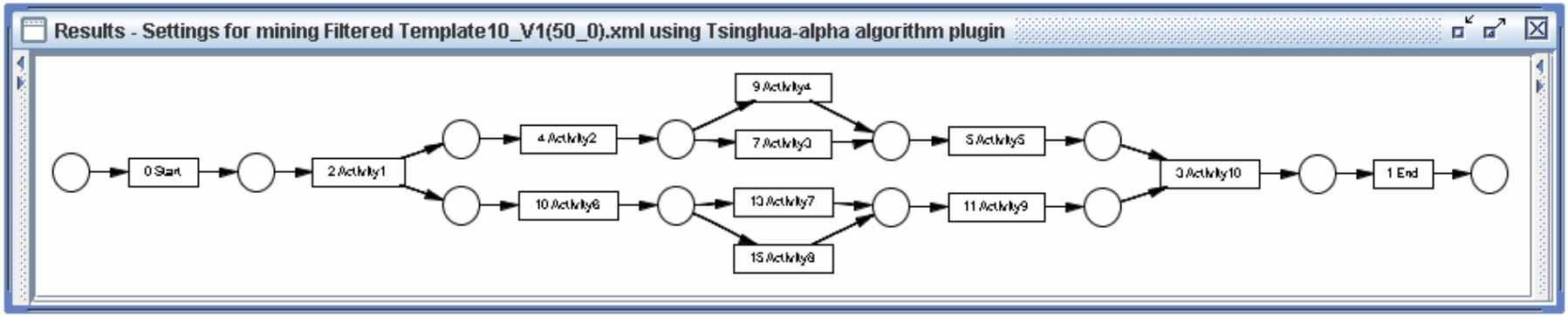


Abbildung A-10b: Template10, V1 in ProM (Tsinghua-Alpha-Algorithmus)

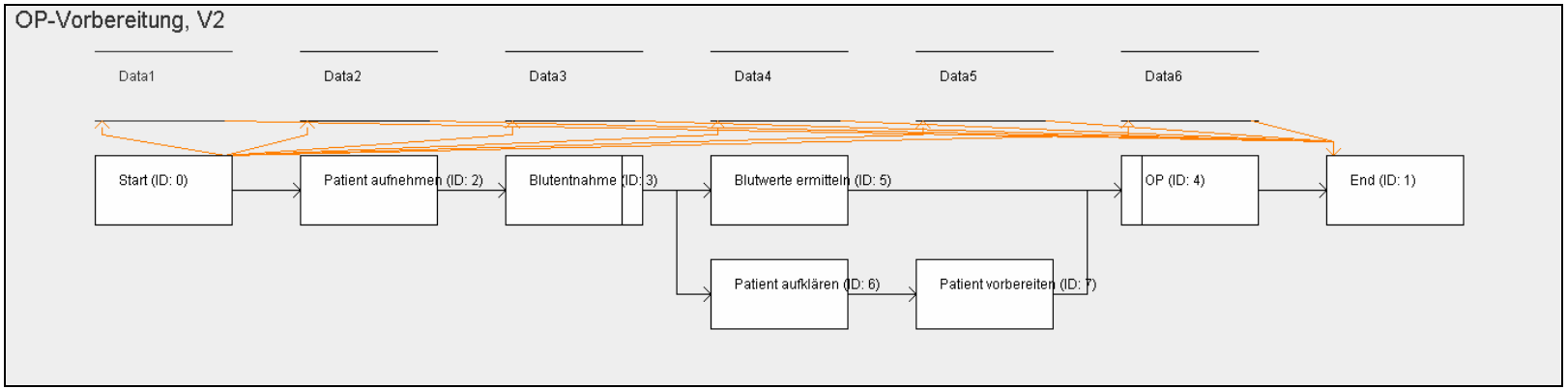


Abbildung A-11: OP-Vorbereitung, V2

Erklärung

Marco Waimer, Matrikelnummer 440 225

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum

Unterschrift