Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 113

Repräsentation und Vergleich des Kontrollflusses von BPEL Prozessmodellen mit temporalen Aktivitätszustandsnetzwerken

Jonas Scheurich

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Dipl.-Inf. Sebastian Wagner
Beginn am:	15. Februar 2016
Beendet am:	16. August 2016
CR-Nummer:	H.4.1

Kurzfassung

Die Business Process Execution Language (BPEL) ist ein Standard zur Modellierung von automatisierten Geschäftsprozessen. Das Verhalten eines Geschäftsprozesses kann mit BPEL unterschiedlich modelliert werden, da verschiedene graphbasierte und sequenzielle strukturierende Aktivitäten verwendbar sind. Die Äquivalenz der Traces unterschiedlicher Modellierungen eines Geschäftsprozesses sind nicht direkt in einem BPEL Modell nachweisbar. Konzepte wie Behavioural Profiles oder die Validierung von Geschäftsprozess-Konsolidierungen sind auf eine Repräsentation eines BPEL Modells angewiesen, mit dem die verschiedenen Traces einer Ausführung von zwei BPEL Prozessen verglichen werden können. Diese Arbeit beschreibt ein Konzept zur Repräsentation von BPEL Prozessen durch ein temporalen Aktivitätenzustandsnetzwerk, das mit einem Punkt Algebra Netzwerk modelliert wird. In einem Punkt Algebra Netzwerk werden die Startzeitpunkte der Zustände aller Aktivitäten modelliert und alle Zeitpunkte in Relation gesetzt. Zum Vergleich der Traces der Basisaktivitäten von zwei BPEL Prozessen, die aus dem temporalen Aktivitätenzustandsnetzwerk eines BPEL Prozesses ermittelt werden, beschreibt diese Arbeit einen Algorithmus. Die Transformation von BPEL Modellen in ein temporales Aktivitätenzustandsnetzwerk und die Funktonalität des Vergleiches der Traces wurde als eine JAVA Anwendung realisiert, die auch in bestehende Projekte eingebunden werden kann.

Abstract

The Business Process Execution Language (BPEL) is a standard to model automated business processes. A business process behaviour can be modelled with BPEL by different alternatives, because BPEL provides graph-based and sequence-based structured activities. The equivalence of the execution traces of different models of a business process can't prove on a BPEL model. Concepts like behavioural profiles or the validation of a business process consolidation are depended on a representation of a BPEL model which provides an equivalence analysis of execution traces. This thesis describes a concept to represent BPEL models by a temporal activity state network. This network is realized with time point algebra. The starting time point of all activity states of all activities and the relations between the time points are modelled by a time point algebra network. This thesis describes an algorithm to compare the basic activity execution traces of two BPEL processes, extracted of a temporal activity state network. The functionality to transform BPEL processes and compare execution traces is implemented by an JAVA application, that can be used by existing projects.

Inhaltsverzeichnis

1 Einleitung **9**
- 1.1 Motivation . . . 9
- 1.2 Problembeschreibung . . . 10
- 1.3 Beitrag . . . 10
- 1.4 Abgrenzung der Arbeit . . . 11
- 1.5 Verwandte Arbeiten zur Repräsentation und dem Vergleich von BPEL Prozessen 11
  - 1.5.1 Repräsentation durch Petri-Netze nach Hinz et al. . . . 12
  - 1.5.2 Repräsentation und Verifikation durch Petri-Netze nach Ouyang . . . 13
  - 1.5.3 Verifikation durch Modellchecker . . . 14
  - 1.5.4 Repräsentation durch endliche abstrakte Zustandsautomaten . . . 15
  - 1.5.5 Repräsentation und Verifikation durch eine Prozess Algebra . . . 15
  - 1.5.6 Einordnung der verwandten Arbeiten . . . 15
- 1.6 Gliederung . . . 16

2 Hintergrund **17**
- 2.1 Business Process Execution Language (BPEL) . . . 17
  - 2.1.1 Basis-Aktivitäten . . . 17
  - 2.1.2 Strukturierende Aktivitäten . . . 18
- 2.2 Temporale Algebren mit Branching Time . . . 21
- 2.3 Punkt Algebra mit Branching Time . . . 22
  - 2.3.1 Operationen der Punkt Algebra . . . 24
- 2.4 Intervall Algebra mit Branching Time . . . 25
  - 2.4.1 Operationen der Intervall Algebra . . . 27
- 2.5 Constraint Propagation . . . 28
  - 2.5.1 Motivation der Constraint Propagation am Beispiel Punkt Algebra . . . 29
  - 2.5.2 Constraint Probagation Algorithmus nach Allen . . . 29
  - 2.5.3 Anwendbarkeit des Algorithmus von Allen auf Punkt Algebra . . . 32
  - 2.5.4 Komplexität des Algorithmus von Allen für Punkt und Intervall Algebra 32

3 Repräsentation und Vergleich durch temporale Aktivitätenzustandsnetzwerke 35
- 3.1 Problemanalyse und Vorüberlegungen . . . 35
- 3.2 Anforderungen an eine Äquivalenz-Analyse . . . 36
- 3.3 Lösungsansatz . . . 36
- 3.4 Aktivitätenzustände . . . 37

3.5 Transformation eines BPEL Prozesses in ein temporales Aktivitätenzustandsnetzwerk . . . 38
3.6 Vergleich von temporalen Aktivitätenzustandsnetzwerken . . . 41
3.7 Betrachtung der Gesamt-Komplexität . . . 44

4 Temporale Modelle 45
4.1 Temporale Aktivitätenzustandsnetzwerke . . . 45
4.1.1 Temporale Repräsentation einer Aktivität . . . 45
4.1.2 Temporale Repräsentation einer empty-Aktivität . . . 46
4.1.3 Temporale Repräsentation einer throw-Aktivität . . . 47
4.1.4 Temporale Repräsentation einer Aktivität mit Fault Handler . . . 47
4.2 Temporale Repräsentation von strukturierenden Aktivtäten . . . 48
4.2.1 Flow mit Aktivität . . . 48
4.2.2 Scope mit Aktivität . . . 50
4.2.3 Scope mit Fault Handler . . . 51
4.2.4 Fault Handler mit Aktivität . . . 52

5 Umsetzung 55
5.1 Anforderungen an die Implementierung . . . 55
5.2 Design der Implementierung . . . 56
5.3 Generisches Datenmodell für temporale Algebren . . . 56
5.4 Datenmodell Punkt Algebra . . . 58
5.4.1 Berechnung der Hülle eines Punkt Algebra Netzwerkes . . . 59
5.5 Transformation BPEL2Punkt Algebra . . . 60
5.5.1 Klassen-Design einer Transformation . . . 61
5.5.2 Erweiterbarkeit durch ein Factory-Repository . . . 63
5.5.3 Implementierung einer Transformation . . . 64
5.6 Vergleich von temporalen Aktivitätenzustandsnetzwerken . . . 65
5.7 Einsatz der Implementierung . . . 67
5.7.1 Konsolenanwndung . . . 67
5.7.2 Klassenimport . . . 67
5.7.3 Markierung von Kontrollflüssen in BPEL Prozessen . . . 68

6 Zusammenfassung und Ausblick 69

Literaturverzeichnis 71

Abbildungsverzeichnis

1.1 Validierung von BPEL Prozessen . . . 11
1.2 Modellierung eines Flows durch Petri-Netze . . . 12
1.3 Transformation von Aktvitäten in ein Petri-Netz . . . 13
1.4 BPEL Modellierung durch Zustandsautomaten . . . 14
1.5 BPEL Modellierung in LOTUS . . . 15

2.1 Temporales Netzwerk . . . 21
2.2 Temporales Netzwerk mit Branching Time . . . 22
2.3 Beispiel eines Punkt Algebra Netzwerkes . . . 23
2.4 Beispiel eines Intervall Algebra Netzwerkes . . . 26
2.5 Lineare Relationen der Punkt Algebra . . . 26
2.6 Nicht-lineare Relationen der Punkt Algebra I . . . 27
2.7 Nicht-lineare Relationen der Punkt Algebra II . . . 27
2.8 Beispiele Constraint Propagation . . . 29
2.9 Propagierung eines Constraints . . . 30

3.1 Lösungsansatz . . . 36
3.2 Das Basis-Zustandsmodel mit Fault Handler . . . 37
3.3 Geschichtetes temporales Aktivitätenzustandsnetzwerk . . . 39
3.4 Beispiel für einen inkonsistenten Kontrollfluss . . . 42

4.1 Temporales Aktivitätenzustandsnetzwerk für eine Aktivität . . . 46
4.2 Temporales Aktivitätenzustandsnetzwerk für eine empty-Aktivität . . . 46
4.3 Temporales Aktivitätenzustandsnetzwerk für eine throw-Aktivität . . . 47
4.4 Temporales Aktivitätenzustandsnetzwerk für eine Aktivität mit Fault Handler 48

5.1 Design der Implementierung . . . 56
5.2 Generisches Datenmodell . . . 57
5.3 Punkt Algebra Datenmodell . . . 59
5.4 Beispiel BPEL Prozess und Netzwerk-Objekte . . . 61
5.5 Basisklasse `AbstractActivityNetwork` . . . 62
5.6 Factory Repository für Transformationsklassen . . . 63
5.7 Klasse `ProcessEquals` . . . 66
5.8 Einsatz der Implementierung als Klassenimport . . . 68

Tabellenverzeichnis

2.1 BPEL Basis-Aktivitäten 18
2.2 Inverse der Punkt Algebra Relationen mit Branching Time 24
2.3 Kompositionstabelle für Punkt Algebra mit Branching Time 25
2.4 Komplexität des Algorithmus von Allen für verschiedene Algebren 32

3.1 Prozeduren des Algorithmus 3.1 41
3.2 Prozeduren des Algorithmus 3.2 44

4.1 Temporale Verknüpfung eines Flows mit den Kind-Aktivitäten 49
4.2 Temporale Verknüpfung von Scopes mit der Kind-Aktivität 51
4.3 Temporale Verknüpfung von Scopes mit dem Fault Handlern 52
4.4 Temporale Verknüpfung von Fault Handlern mit der Kind-Aktivität 53

5.1 Methoden zur Implementierung der Transformation einer Aktivität 64
5.2 Methoden zur Implementierung des Interface `IActivityNetworkFactory` . . 65
5.3 Repräsentation des Ergebnis eines Vergleiches 66

Verzeichnis der Algorithmen

2.1 Constraint Propagation Algorithmus nach Allen 31

3.1 Transformation eines BPEL Prozesses in Punkt Algebra 40
3.2 Vergleich von zwei BPEL Prozessen 43

1 Einleitung

Die Unterstützung von Business Prozessen durch IT-Technologien ist ein wichtiges Werkzeug des Business Process Management. Die Optimierung bestehender Prozesse oder die Einführung neuer Prozesse bilden Ausgangspunkte zum Einsatz von Technologien, wie die Business Process Execution Language (BPEL) oder auch Business Process Model and Notation (BPMN) [FR14].

Ein wichtiges Kriterium bei der Optimierung von BPEL Prozessen ist die Äquivalenz der Traces von möglichen Ausführungen eines BPEL Prozesses. Diese Arbeit stellt eine temporale Repräsentation von BPEL Prozessen vor, die dazu dient, die möglichen Traces von zwei BPEL Prozessen zu vergleichen. Grundlage der temporalen Repräsentation ist das Zustandsmodell nach Kopp et al. [KHK+11].

Das in dieser Arbeit beschriebene Konzept wurde in einer Java Implementierung umgesetzt.

1.1 Motivation

In der Modellierung von Geschäftsprozessen werden verschiedene Modellierungstechniken und -perspektiven eingesetzt. Zum Beipsiel werden in der Reisekostenabrechnung und der Finanzplanung unterschiedliche Modelle eingesetzt. Mit diesen unterschiedlichen Modellen kann das gleiche Verhalten modelliert werden. Das Konzept der Behavioural Profiles [WMW11] bestimmt Konsistenzen zwischen zwei Prozessbeschreibungen, die auf unterschiedlichen Meta-Modellen basieren. Die Konsistenzen werden jeweils aus einem temporalen Modell der unterschiedlichen Prozess-Darstellungen bestimmt.

Zur Optimierung der Ausführung von BPEL Prozessen kann die Struktur des BPEL Prozesses vor der Ausführung durch eine BPEL-Engine angepasst werden [WRK+13]. Ein wichtiges Qualitätsmerkmal der Optimierung von BPEL Prozessen ist die Äquivalenz der Kontrollflusssemantik vor und nach der Optimierung. Eine Validierung von Optimierungen muss transparent von den strukturierenden Aktivitäten der BPEL Prozesse durchgeführt werden, da die Struktur des Prozesses durch die Optimierung verändert wird.

Die in diesem Abschnitt genannten Ansätze benötigen als Grundlage ein temporales Modell eines BPEL Prozesses. Zur Durchführung dieser Analysen beschreibt diese Arbeit ein temporales Aktivitätenzustandsmodell.

1.2 Problembeschreibung

Kontrollflusssemantiken von automatisierten Geschäftsprozessen können durch verschiedene strukturierende Aktivitäten in BPEL modelliert werden. Diese strukturierenden Aktivitäten durchlaufen zur Laufzeit einer Prozess-Instanz verschiedene Zustände. Diese Zustände sind nicht in einem BPEL Geschäftsprozess enthalten. Eine genaue Analyse der Relation der Aktivitätenzustände ist nicht direkt im BPEL-Modell möglich.

BPEL erlaubt es durch verschiedene Modellierungskonstrukte einen Geschäftsprozess auf verschiedene Weise zu modellieren. Das Verhalten eines Geschäftsprozesses wird durch seine Basis-Aktivitäten und den strukturierenden Aktivitäten beschrieben. Durch die verschiedenen Modellierungskonstrukte gibt es unterschiedliche Beschreibungen des Verhaltens eines Geschäftsprozesses, die zur Durchführung von Analysen herangezogen werden kann.

1.3 Beitrag

Im Rahmen dieser Masterarbeit wurde ein temporales Aktivitätenzustandsmodell für BPEL Prozesse zur Analyse von BPEL Prozessen entworfen. Das temporale Aktivitätenzustandsmodell umfasst die einzelnen Startzeitpunkte der Zustände der BPEL Aktivitäten und beschreibt die temporalen Abhängigkeiten zwischen den einzelnen Startzeitpunkten der Aktivitätenzustände, die aus der Struktur- und der Verhaltensbeschreibung eines BPEL Prozesses und der BPEL Spezifikation abgeleitet werden.

Durch eine temporale Repräsentation der Aktivitätenzustände eines BPEL Prozesses steht ein Modell zur Verfügung auf dem Analysen transparent von den strukturierenden Aktivitäten durchgeführt werden können. Die Kontrollflusssemantik und die operationale Semantik der strukturierenden Aktivitäten ist in diesem temporalen Modell trotzdem weiterhin enthalten.

Die Implementierung des Konzeptes dieser Arbeit umfasst ein Framework, dass eine Transformation von Basis- sowie strukturierende Aktivitäten in ein temporales Modell umfasst. Die Integrationsfähigkeit in bestehende Softwaresysteme ist durch den Einsatz eines EMF-basierten BPEL Meta-Modell gewährleistet. Weiterhin enthält die Implementierung zur Äquivalenz-Analyse von BPEL Prozessen ein Analyse-Tool zum Vergleich von temporalen Aktivitätenzustandsmodellen.

Konzepte wie z. B. Behavioural Profiles oder die Konsolidierung von BPEL Prozessen erhalten durch diese Arbeit ein formales Modell sowie ein Analyse-Tool für den Vergleich des Traces von BPEL Prozessen.

1.4 Abgrenzung der Arbeit

Diese Arbeit beschäftigt sich ausschließlich mit der Analyse von BPEL Modellen, einem Teilaspekt des Business-Process-Modeling Lebenszyklus [LLN11]. In Hinblick auf eine Prozesskette zur Optimierung einer Ausführung eines BPEL Prozesses, wie in Abbildung 1.1 ist nur die Validierung einer Optimierung hinsichtlich der Äquivalenz der Aktivitätenausführungsreihenfolge Teil dieser Arbeit.

Abbildung 1.1: Einordnung einer Validierung von BPEL Prozessen zwischen der Konsolidierung und der Ausführung.

Die Definition des temporalen Aktivitätenzustandsmodells ist ausschließlich aus der BPEL Spezifikation [OAS07] abgeleitet. Erweiterungen von BPEL und Ausführungssemantiken, die nicht der Spezifikation entsprechen, werden in dieser Arbeit nicht berücksichtigt. Das temporale Aktivitätenzustandsmodell betrachtet die Zustände *init*, *executing*, *dead*, *aborted*, *terminating*, *terminated*, *fault handling*, *fault*, *fault caught* und *completed* der Aktivitäten eines BPEL Prozesses. Das temporale Modell bildet die folgenden Aktivitäten ab: Scope, Fault Handler, Flow mit Links, Receive, Reply, Invoke, Assign, Exit, Wait, Empty und Throw.

Erweiterungen von BPEL, Choreographien, Variablen sowie Nachrichten und deren Korrelation werden in dieser Arbeit nicht berücksichtigt.

Die Transformation eines BPEL Prozesses in ein temporales Aktivitätenzustandsmodell benötigt eine Analyse der Kontrollflussbedingungen wie z. B. Join- oder Transition-Conditions. Diese Analyse ist kein Teil dieser Arbeit.

1.5 Verwandte Arbeiten zur Repräsentation und dem Vergleich von BPEL Prozessen

Im Rahmen der Forschung zu automatisierten Geschäftsprozessen wurden auf den entsprechenden Konferenzen auch Arbeiten zur Verifikation von BPEL Prozessen vorgestellt. Eine formale Darstellung eines BPEL Prozesses ist Grundlage für einen formalen Vergleich von BPEL Prozessen.

Neben der Repräsentation eines BPEL Prozesses als temporales Modell wurden in verschiedenen Forschungsarbeiten weitere Konzepte vorgestellt, die BPEL Prozesse in eine formale Darstellung transformieren. Diese verwandten Arbeiten werden in diesem Kapitel beschrieben.

Van Breugel und Koshkina beschreiben in einer Literatur Studie verschiedenen Ansätze zur formalen Darstellung von BPEL Prozessen [VK06].

1.5.1 Repräsentation durch Petri-Netze nach Hinz et al.

Hinz et al. und Lohmann et al. haben eine vollständige Transformation von BPEL Prozessen zu Petri-Netzen beschrieben und umgesetzt [HSS05; Loh07]. Petri-Netze modellieren durch Plätze, Transaktionen und Token das Verhalten von Software [W85]. Diese Transformation arbeitet mit verschiedenen Transformationspattern für die einzelnen BPEL Aktivitäten. Die Zustände einer Aktivität werden durch Plätze dargestellt, die über Transaktionen mit den inneren Aktivitäten interagieren.

Abbildung 1.2 zeigt das Pattern zur Modellierung eines BPEL Flows durch Petri-Netze. Jedes der Pattern der Transformation enthält ein Interface aus Plätzen nach außen und modelliert die Abhängigkeiten des eigenen Interfaces zu den Plätzen der inneren Aktivitäten über verschiedene Transaktionen.

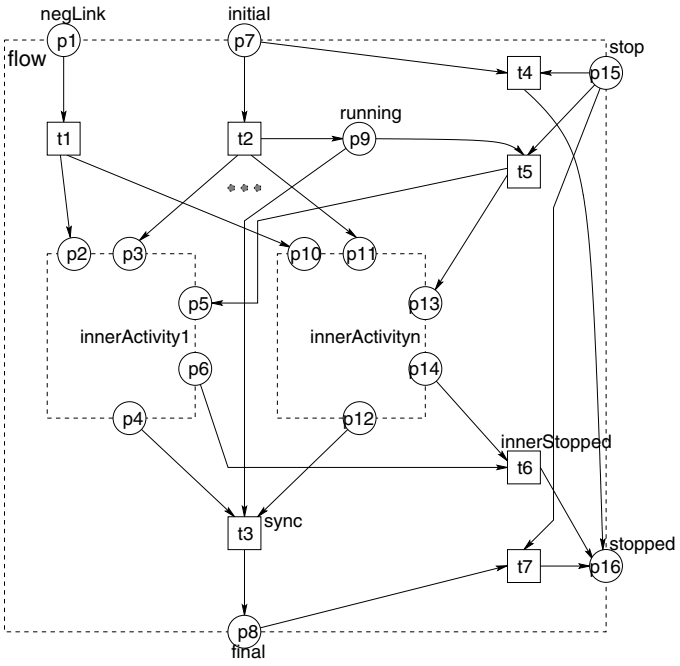


Abbildung 1.2: Modellierung eines Flows durch Petri-Netze [HSS05]

Die Transformation umfasst, die Modellierung von Basis-Aktivitäten, strukturierender Aktivitäten und der Daten von BPEL Prozessen [Loh07]. Die Korrelation von Nachrichten ist kein Teil der Transformation. Die verschiedenen Analyse-Techniken für Petri-Netze erlauben die

Verifikation eines BPEL Prozesses hinsichtlich Korrektheit, Vollständigkeit und Terminierung [HSS05].

1.5.2 Repräsentation und Verifikation durch Petri-Netze nach Ouyang

Ein weiterer Ansatz zur Repräsentation und Verifikation von BPEL Prozessen durch Petri Netze beschreiben Ouyang et al. [OVV+07]. Im Gegensatz zu dem Ansatz mit Petri-Netzen aus Kap. 1.5.1 modelliert die Transformation die Aktivitätenzustände *start* und *completed* der Aktivitäten in einem Petri-Netz. Das weitere Verhalten von BPEL und der Kontrollfluss wird mit weiteren Transitionen und Plätzen um die Abbildung dieser beiden Zustände der Aktivitäten modelliert. Die Abbildung 1.3 zeigt Teile der Transformationsdefinition. Die Transformation umfasst, alle Aktivitäten, Nachrichten und Nachrichten-Korrelation und Choreographien.

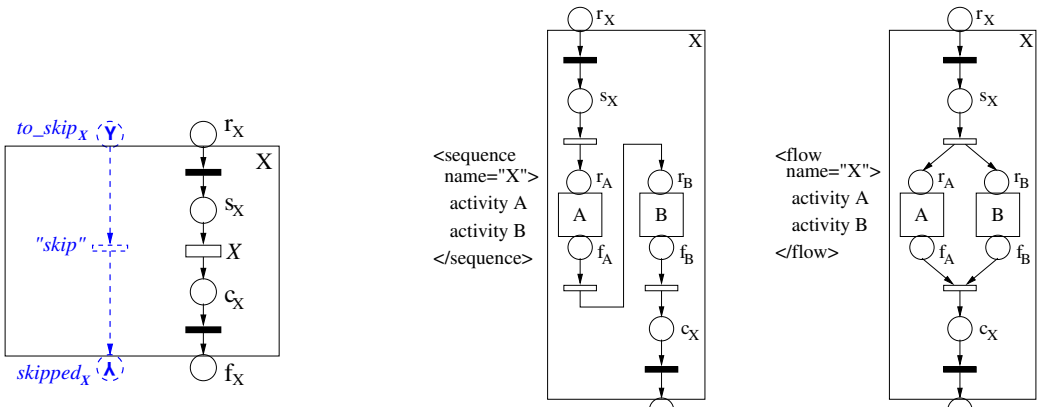


Abbildung 1.3: Transformation einer Aktivität, sowie einer Sequenz und eines Flows in ein Petri-Netz nach [OVV+07]

Zur Analyse von BPEL Prozessen als Petri-Netz definieren Ouyang et al. folgende Analysen:

- Unerreichbare Aktivitäten
- Erkennung von Verletzungen von BPEL Constraints (z.B. zwei receive-Aktivitäten eines Prozesses konsumieren den gleichen Nachrichten-Typ)
- Garbage Collection von Queued Messages
- Deadlocks in interagierenden Prozessen

1.5.3 Verifikation durch Modellchecker

Fu et al. beschreiben eine Transformation von BPEL Choreographien [KEvL+11] zur Verifikation mit dem Modellchecker Spin[1] [FBS04]. Dazu wird eine BPEL Choreographie in einen Zustandsautomaten transformiert. Dieser Zustandsautomat wird wiederum in Promela, der Programmiersprache von Spin, übersetzt.

Abbildung 1.4 zeigt die Transformation für die Aktivitäten Sequence und Flow. Der Zustandsautomat stellt für die Ausführung jeder einzelnen Aktivität einen Zustand dar. Gemäß der Semantik von BPEL sind diese Zustände mit Transitionen verbunden. Das vollständige Zustandsmodell der einzelnen Aktivitäten wird nicht betrachtet.

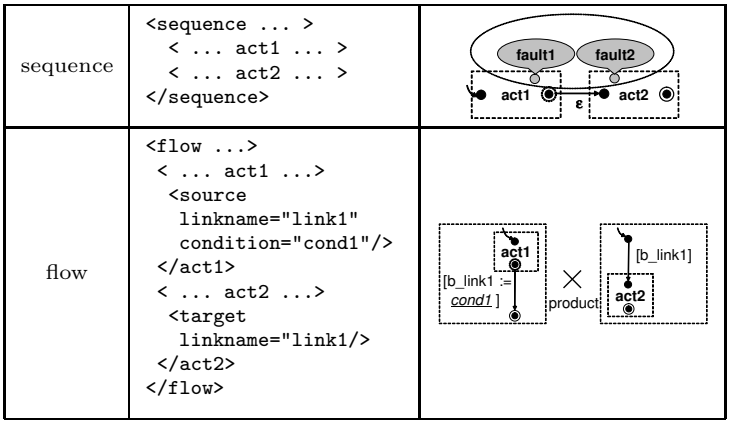


Abbildung 1.4: Modellierung von BPEL Aktivitäten durch Zustandsautomaten zur Verifikation mit dem Modelchecker Spin [FBS04]

Die Transformation in den Zustandsautomaten umfasst im Gegensatz zu anderen Transformationen die Unterscheidung der verschieden MessageTyps eines BPEL Prozesses und die möglichen Änderungen von Variablen durch XPath-Ausdrücke. Dadurch sind auch die Variablen durch einen Modellchecker validierbar. Nach der Transformation können durch LTL-Ausdrücke formulierte Eigenschaften der BPEL Choreographie mit dem Modelchecker verifiziert werden. Folgende Eigenschaften, können auf dem Promela Modell verifiziert werden :

- Synchrone Kompatibilität
- Autonomie
- Verlustfreiheit der Komposition

[1]Spin Model Checker: http://www.http://spinroot.com/

1.5.4 Repräsentation durch endliche abstrakte Zustandsautomaten

Auf Grundlage eines verteilten abstrakten Zustandsautomaten haben Farahbod et al. eine Transformation von BPEL Prozessen in endliche abstrakte Zustandsautomaten beschrieben [FGV05]. Der Zustandsautomat bildet eine reduzierte Menge der Aktivitätenzustände ab, die jeweils von Agenten verwaltet werden. Der Formalismus beschreibt Basis-Aktivitäten, strukturierende Aktivitäten und Nachrichten.

1.5.5 Repräsentation und Verifikation durch eine Prozess Algebra

Ein weiterer Ansatz zur temporalen Verifikation von BPEL Prozessen sind Prozess Algebren. Es existieren verschiedene Prozess-Algebren wie z. B. CCS, π-calculus oder LOTUS. Farrera beschreibt eine Transformation von BPEL in die Prozess-Algebra LOTUS und umgekehrt [Fer04]. Nach der Transformation können mit bestehenden Werkzeugen Verifikationen des BPEL Prozesses durchgeführt werden. Die Abbildung 1.5 zeigt einen Ausschnitt der Darstellung von strukturierenden Aktivitäten eines BPEL Prozesses durch die Prozess Algebra LOTUS.

<pick ... > <onMessage ... variable="m1"> < ... act1 ...> </onMessage> <onMessage ... variable="m2"> < ... act2 ... > </onMessage> </pick>	(g1?m1:Nat; ..act1..) [] (g2?m2:Nat; ..act2..)
<sequence ...> < ... act1 ... > < ... act2 ... > </sequence>	..act1..; ..act2..
<flow ... > < ... act1 ... > <source linkname="link1" condition="cond1"/> </act1> < ... act2 ... > <target linkname="link1"/> </act2> </flow>	..act1..; ([cond1]->link1 !1; [] [not(cond1)]->link1 !0;) || (link1 ?x:Bool; ([x=1]->..act2.. [] [x=0]->i;))

Abbildung 1.5: Darstellung von BPEL Sequenzen und Flows durch die Prozess Algebra LOTUS [Fer04]

1.5.6 Einordnung der verwandten Arbeiten

Die in diesem Kapitel vorgestellten verwandten Arbeiten beschäftigen sich mit der formalen Repräsentation von BPEL Prozessen oder Choreographien durch Petri Netze, Zustandsautomaten, abstrakte Zustandsautomaten und Prozess Algebren.

All diese Ansätze verwenden eine geringe Zustandsmenge für Aktivitäten. Teilweise wird nur der $executing$-Zustand betrachtet. Die Analysemöglichkeiten begrenzen sich auf die Analyse von Eigenschaften der BPEL Prozesse, wie z. B. Deadlockfreiheit. Eine Analyse der Traces wird nicht beschrieben.

1.6 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Hintergrund beschreibt den theoretischen Hintergrund dieser Arbeit.

Kapitel 3 – Repräsentation und Vergleich durch temporale Aktivitätenzustandsnetzwerke beschreibt ein Konzept zur Repräsentation eines BPEL Prozesses durch Punkt Algebra und den Vergleich von zwei BPEL Prozessen, die als ein temporales Aktivitätenzustandsnetzwerk vorliegen.

Kapitel 4 – Temporale Modelle beschreibt die temporalen Modelle der einzelnen BPEL Aktivitäten, die im Konzept verwendet werden.

Kapitel 5 – Umsetzung dokumentiert die Umsetzung des Konzepts in eine JAVA Applikation.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Hintergrund

Dieses Kapitel stellt die theoretischen Hintergründe, die Grundlage dieser Arbeit vor. Grundlegende Begriffe und Konstrukte von BPEL werden beschrieben. Die temporalen Algebren Punkt Algebra [VKv89] und Intervall Algebra [All83] und deren Zusammenhang werden erläutert. Zur Erweiterung der beiden Algebren ist das Konzept der Branching Time [Ivo99; Bro01; Rei94] beschrieben. Das Kapitel schließt mit dem Constraint Propagation Algorithmus von Allen [All83], der die Hülle eines temporalen Netzwerkes berechnet.

2.1 Business Process Execution Language (BPEL)

Dieser Abschnitt erläutert und beschreibt die domänenspezifische, imperative Programmiersprache zur Definition von ausführbaren Geschäftsprozessen BPEL [LLN11] die Aufrufe verschiedener Webservices orchestriert. Die Details von BPEL sind der Spezifikation [OAS07] zu entnehmen, auf der dieser Abschnitt basiert. In BPEL können mittels Basis-Aktivitäten und verschiedener strukturierender Aktivitäten Geschäftsprozesse modelliert werden. Diese Beschreibung des Geschäftsprozesses kann von einer Workflow-Maschine interpretiert zur Ausführung gebracht werden.

2.1.1 Basis-Aktivitäten

In der BPEL Spezifikation sind verschiedene Basis-Aktivitäten definiert. Basis-Aktivitäten stellen die elementaren Schritte des Prozess-Verhaltens dar. Die Tabelle 2.1 gibt einen Überblick über die Basis-Aktivitäten von BPEL.

receive	Eine receive-Aktivität blockiert, bis eine Nachricht empfangen wurde. Die empfangene Nachricht wird in einer Variable gespeichert. Ist eine receive-Aktivität eine unsynchronisierte Aktivität wird mit dem Empfang einer Nachricht eine Prozess-Instanz erstellt, wenn das Attribut $createInstance = yes$ gesetzt ist.
reply	Eine reply-Aktivität erstellt eine Nachricht und sendet diese zum Absender, der entsprechenden invoke-Aktivität, zurück.
invoke	Eine invoke-Aktivität führt einen synchronen oder asynchronen Aufruf eines Webservices durch. Zur Behandlung von Fehlern kann ein Fault Handler hinzugefügt werden. Der Fault Hanlder wird ausgeführt, wenn während der Ausführung der Aktivität ein Fehler auftritt. Mit einem Compensation Handler werden Aktivitäten beschrieben, die das Verhalten dieser invoke-Aktivität im Falle eines Abbruchs nach der erfolgreichen Ausführung der Aktivität kompensieren.
assign	Eine assign-Aktivität dient dazu Werte von z. B. Variablen zu kopieren oder Variablen neue Werte zuzuweisen.
throw	Eine throw-Aktivität signalisiert einen internen Fehler und enthält Informationen über den Fehler. Diese Informationen können von einem Fault Handler zur weiteren Fehlerbehebung verarbeitet werden.
exit	Eine exit-Aktivität beendet die Prozess-Instanz mit allen enthaltenen Aktivitäten ohne weiteres Termination Handling, Fault Handling oder Compensation Handling.
wait	Eine wait-Aktivität verzögert den Ablauf des Prozesses um einen konkreten Zeitabschnitt oder bis zu einem spezifiziertem Zeitpunkt.
empty	Eine empty-Aktivität stellt kein Verhalten des Prozesses dar und dient zur einfacheren Modellierung der Struktur des Prozesses, oder als Synchronisierungspunkt für Kontrollflusslinks.
compensate	Startet das Compensation Handling für die aktuelle Scope-Aktivität.
compensateScope	Startet das Compensation Handling für einen konkreten Scope.
rethrow	Eine rethrow-Aktivität wird in Fault Handlern genutzt um den gefangenen Fehler an übergeordnete strukturierte Aktivitäten zu signalisieren.

Tabelle 2.1: BPEL Basis-Aktivitäten

2.1.2 Strukturierende Aktivitäten

Der Kontrollfluss eines BPEL Prozesses wird durch Sequenzen, Verzweigungen, Schleifen und Flows modelliert. BPEL stellt zwei verschiedene Konzepte zur Modellierung des Prozess-Verhaltens zur Verfügung. Zum einen eine graph-basierte Strukturierung. Mittels Links können zwischen Aktivitäten Kontrollflusspfade modelliert werden. Als zweite Möglichkeit zur Model-

lierung von Prozess-Verhalten stehen unterschiedliche sequenzielle strukturierende Aktivitäten zur Verfügung.

Flow-Aktivität

Ein Flow enthält mehre Aktivitäten, die durch Links miteinander verbunden sind. Mit einem Flow lässt sich somit auch komplexes Prozess-Verhalten beschreiben. In einem Flow werden Links definiert, die Kontrollflusspfade zwischen einzelnen Aktivitäten des Flows darstellen.

Die Aktivitäten eines Flows enthalten Verweise auf eingehenden und ausgehenden Links. Links werden von Aktivitäten innerhalb und auch außerhalb des Flows als eingehender oder ausgehender Link referenziert. Für die eingehenden Links einer Aktivität des Flows können Join-Conditions in einer Aktivität beschrieben werden, die den Kontrollfluss an dieser Stelle einschränken. Für ausgehende Links können Transition-Conditions beschrieben werden, die die Aktivierung von nachfolgenden Aktivitäten einschränken.

Es ist möglich Links über die Grenzen von strukturierenden Aktivitäten zu erstellen, die keine Schleifen-Semantik aufweisen. Die Grenzen folgender Aktivitäten dürfen nicht mit Links gekreuzt werden: While, ForEach, RepeatUntil, Event Handler, Compensation Handler. Ein Fault Handler darf keine eingehenden Links aufweisen. Ausgehende Links die keine Aktivitäten des fehlerhaften Scopes zum Ziel haben, sind jedoch möglich.

Ist die Dead-Path-Elimination aktiviert, werden die Aktivitäten, die auf einen Link folgen, der durch die Transition-Condition deaktiviert wurde, ebenfalls deaktiviert.

Tritt in einer Aktivität einer Flow-Aktivität ein Fehler auf, wird die Ausführung der Flow-Aktivität abgebrochen. Der signalisierte Fehler kann von einer hierarchisch übergeordneten scope-Aktivität verarbeitet werden.

Scope-Aktivität

Ein Scope dient dazu einer Kindaktivität einen gekapselten Ausführungskontext zu bieten. Dieser eigene Ausführungskontext beinhaltet Partner Links, Variablen, Event Handlern, einem Fault Handler, einen Compensation Handler und einen Termination Handler.

Ein Event Handler wird aktiviert, während der Scope ausgeführt wird. Für jedes Event, für das ein Event Handler aktiviert ist, wird die zugehörige Aktivität ausgeführt, wenn dieses Event auftritt. Der Zustand des Event Handlers ist synchronisiert mit seinem Scope. Wird der Scope terminiert, terminiert auch der Event Handler, etc.

Ein Fault Handler wird ausgeführt, falls die Kindaktivität des Scopes mit einem Fehler endet. Je nach Fehlertyp wird ein passender Fault Handler ausgewählt, dessen Aktivitäten die Fehlerauswirkungen behandeln. Der Standard Fault Handler ruft den Compensation Handler des Scopes auf der eine Rethrow-Aktivität ausführt.

Ein Compensation Handler wird ausgeführt, um Effekte eines erfolgreich beendeten Scopes zu kompensieren. Wurde ein Scope mehrfach ausgeführt, da dieser Teil einer Schleife ist, wird auch der Compensation Handler mehrfach ausgeführt. Enthält die Kindaktivität des Scopes weitere Scopes werden im Fall des Standard Compensation Handler die Compensation Handler ausgeführt, die in der Kindaktivität enthalten sind. Diese Ausführung geschieht, nachdem der Compensation Handler des übergeordneten Scopes ausgeführt wurde. Die Ausführung eines Compensation Handler wird über die Aktivitäten Compensate oder CompensateScope gestartet. Der Standard Compensation Handler ruft die Compensate-Aktivität auf, um den Compensation Handler eines Scopes zu starten, die in diesem Scope enthalten sind.

Ein Termination Handler wird ausgeführt, wenn ein Scope während der Ausführung der Kindaktivität terminiert wird. Wurde ein Scope zur Terminierung markiert, wird zunächst die Aktivität des Scopes terminiert und die Event Handler deaktiviert. Anschließend wird der Termination Handler des Scopes ausgeführt. Der Standard Termination Handler startet mit der Compensate-Aktivität den Compensation Handler des Scopes.

Sequence-Aktivität

Eine Sequence enthält eine Menge von geordneten Aktivitäten, die in dieser Reihenfolge bei Ausführung der Sequenz ausgeführt werden.

If-Aktivität

Eine If-Aktivität enthält einen oder mehrere geordnete bedingte Zweige und einen optionalen alternativen Zweig. Jeder Zweig enthält eine Aktivität. Bei Ausführung der If-Aktivität werden die Bedingungen der Zweige der Ordnung nach evaluiert. Die Aktivität des ersten Zweiges, dessen Bedingung wahr ist, wird ausgeführt. Ist keine der Bedingungen wahr, wird der alternative Zweig ausgeführt.

While-Aktivität

Eine While-Aktivität enthält eine Bedingung und eine Aktivität. Die Aktivität wird solange wiederholt, bis die Bedingung vor einer möglichen Ausführung der Aktivität unwahr ist.

RepeatUntil-Aktivität

Eine RepeatUntil-Aktivität enthält eine Bedingung und eine Aktivität. Die Aktivität wird solange wiederholt, bis die Bedingung nach der Ausführung der Aktivität unwahr ist.

ForEach-Aktivität

Eine ForEach-Aktivität enthält einen Scope, der hintereinander oder parallel ausgeführt wird. Ob eine ForEach-Aktivität erfolgreich abgeschlossen wurde, kann durch eine Branch-Condition beschrieben werden.

Pick-Aktivität

Eine Pick-Aktivität blockiert den Kontrollfluss, bis ein Event aus einer definierten Menge von Events auftritt und führt eine zugeordnete Aktivität aus. Die definierten Events können unterschiedliche Nachrichten oder zeitabhängige Events sein. Eine Pick-Aktivität kann auch zum Instanziieren von Prozessen genutzt werden. In diesem Fall darf die Pick-Aktivität keine zeitabhängigen Events enthalten.

2.2 Temporale Algebren mit Branching Time

Temporale Algebren beschreiben eine partielle Ordnung von Knoten [Bro01] mit Hilfe eines Vorgänger-Operators $\prec$. Abbildung 2.1 zeigt mehrere Knoten, deren Relation zueinander beschrieben ist. Mit der Ausnahme der Konten i und j besteht zwischen den Knoten des temporalen Netzwerkes eine totale Ordnung.

Wird die Relation von mehreren Knoten zueinander beschrieben, ergibt sich ein temporales Netzwerk. Ein temporales Netzwerk ist ein vollständiger Graph, der durch Relationen eingeschränkt wird, wie in Abbildung 2.1 gezeigt. Die Auswirkungen einer Einschränkung von Relationen zwischen zwei Knoten auf die Relationen zwischen anderen Knotenpaaren können durch Algorithmen berechnet werden.

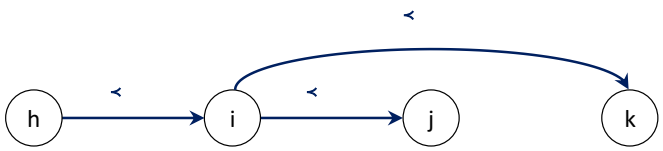


Abbildung 2.1: Temporales Netzwerk

Das Konzept der Branching Time für temporale Algebren wurde von Düntsch et al. formal beschrieben [Ivo99]. Zur Darstellung von Knoten, die auf unabhängigen Zweigen liegen, wurde mit der Branching Time für temporale Netzwerke der Unrelated-Operator $\|$ eingeführt. Das temporale Netzwerk verzweigt sich an einem Knoten t in zwei Zweige b und b', so dass folgende Bedingungen gelten:

1. Wenn $t' \parallel t''$, dann existiert ein t mit $t \prec t'$ und $t \prec t''$.
2. Für jedes Paar an Zweigen b und b' gilt $b \cap b' \neq \emptyset$.

Die Branching Time beschreibt Knoten, die nicht vergleichbar sind, da diese Konten sich in verschiedenen Möglichkeiten der Zukunft befinden. Der Knoten t aus der Bedingung 1 stellt den letztmöglichen Knoten dar, an dem entschieden wird welcher Zweig gewählt wird. Aus dieser Entscheidung ergeben sich die Zweige b und b' die alle weiteren Knoten enthalten, die dem Konten t' bzw. t'' nachgeordnet sind. Abbildung 2.2 zeigt ein temporales Netzwerk, welches sich am Knoten i in zwei Zweige aufteilt.

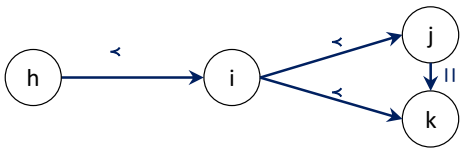


Abbildung 2.2: Temporales Netzwerk mit Branching Time

2.3 Punkt Algebra mit Branching Time

Die Punkt Algebra beschreibt die Relation zwischen zwei Zeitpunkten. Erstmals wurde die Punkt Algebra von Reich [Rei94] eingeführt. Das Konzept der Branching Time wurde von Broxvall in das Konzept der Punkt Algebra übernommen wurde [Bro01]. Die Punkt Algebra beschreibt eine Menge von Zeitpunkten, die linear auf einem Zeitstrahl angeordnet sind. Das Verhältnis der Zeitpunkte wird durch die Relationen <, >, = und $\neq$ beschrieben. Punkt Algebra Netzwerke die nicht die Relation $\neq$ verwenden, werden als stetig bezeichnet. In dieser Arbeit kommen nur stetige Punkt Algebra Netzwerke zur Anwendung. In einer Punkt Algebra mit Branching Time existieren Zeitpunkte, die in keiner linearen Relation zueinander stehen. Durch die Erweiterung der Punkt Algebra mit Branching Time sind die Zeitpunkte statt auf einem linearen Zeitstrahl auf einem Zeitbaum angeordnet. Zeitpunkte, die auf unterschiedlichen Zweigen des Baumes liegen, stehen in keiner linearen Relation zueinander. Zeitpunkte, die auf unterschiedlichen Zweigen angeordnet sind, werden als unrelated (Relation $\|$) bezeichnet.

Abbildung 2.3 zeigt ein Beispiel eines Punkt Algebra Netzwerkes und eine mögliche Abbildung des Graphes auf einen Zeitstrahl. Für die Zeitpunkte C und D existieren zwei verschiedene Abbildungen auf den Zeitstrahl. Entweder finde C vor, nach oder gleichzeitig mit dem Zeitpunkt D statt.

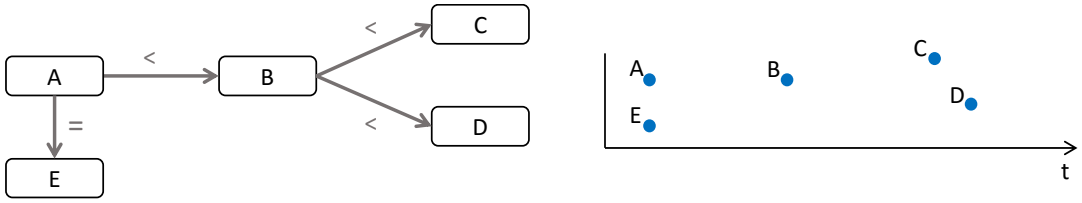


Abbildung 2.3: Beispiel eines Punkt Algebra Netzwerkes als Graph und eine mögliche Abbildung auf den Zeitstrahl.

Eine Problem Instanz Π besteht aus einer Menge von Variablen V über eine Domäne $\mathcal{D}$ und wird im folgendem definiert [RW04a]. Weiterhin enthält die Problem Instanz eine endliche Menge von Constraints Θ. Ein Constraint enthält eine Menge von Relationen $R_1 \dots R_n$ aus einem Set von Relationen $\mathcal{S}$, sowie die zwei Variablen x_1 und x_2 zwischen denen das Constraint besteht.

Instanz $\Pi =< V, \Theta >$

Constraint $(\{R_1, ..., R_n\} x_1, x_2) \in \Theta$

Relation $R \in \mathcal{S}$

Für die Punkt Algebra ist die Domäne $\mathcal{D}$ die kontinuierliche Zeit, die Variablen V sind die Zeitpunkte des Punkt Algebra Netzwerkes und das Set $\mathcal{S}$ setzt sich aus linearen und nicht-linearen Relationen zusammen:

Lineare Relationen

- $x < y$ wenn x Vorgänger von y in $\mathcal{D}$.
- $x > y$ wenn y Vorgänger von x in $\mathcal{D}$.
- $x = y$ wenn x, y der gleiche Punkt sind.

Nicht-lineare Relation

- $x \parallel y$ wenn x, y zu unterschiedlichen Branches oder Bäumen gehören.

Daraus ergibt sich das Set der möglichen Relationen:

$$\mathcal{S} = \{<, >, =, \parallel\}$$

Initial ist die Problem Instanz ein vollständiger Graph, dessen Constraints mit $\Theta = (\mathcal{S}, i, j) : 1 < i < n, 1 < j < n$ initialisiert sind. Ein solches Constraint wird als T-Relation bezeichnet [RW04a]. Erst durch die Einschränkung von Constraints ergibt sich ein spezifisches Punkt Algebra Netzwerk. Jede Einschränkung hat Auswirkungen auf die anderen Constraints des Punkt Algebra Netzwerkes, die durch die Hülle des Punkt Algebra Netzwerkes berechnet werden.

2.3.1 Operationen der Punkt Algebra

Die Operationen für Schnitt, Inverse und Komposition der Constraints der Punkt Algebra werden in den folgenden Abschnitten definiert. Diese Operationen werden später zur Berechnung der Hülle eines Punkt Algebra Netzwerkes verwendet.

Schnitt von Constraints

Teil des Schnittes zweier Constraints sind die Relationen, die in beiden Constraints enthalten sind. Der Schnitt der Menge von Relationen von zwei Constraints ist wie folgt definiert.

$$(R_1, \ldots, R_n) \cap \{R'_1, \ldots, R'_m) = \bigcup r \in \mathcal{S} : r \subseteq (R_1, \ldots, R_n) \text{ and } r \subseteq (R'_1, \ldots, R'_m)$$

Inverse von Constraints

Die Inverse eines Constraints wird aus den Inversen der einzelnen Relationen gebildet:

$$(R_1, \ldots, R_n)^{-1} = (R_1^{-1}, \ldots, R_n^{-1})$$

Inverse von Relationen

Die Relationen $\|$ und $=$ sind symmetrischen Relationen. Die Relationen $<$ und $>$ bilden ein inverses Relationen-Paar.

R	R^{-1}
$<$	$>$
$>$	$<$
$\|$	$\|$
$=$	$=$

Tabelle 2.2: Inverse der Punkt Algebra Relationen mit Branching Time

Komposition von Constraints

Die Komposition zweier Punkt Algebra Constraints beschreibt die Konkaternierung zweier Constraints. Die Komposition der Relationen der beiden Constraints berechnet sich wie folgt [RW04a]:

$$(R_1, \ldots, R_n) \otimes (R'_1, \ldots, R'_m) = \bigcup_{1 \leq i \leq n, 1 \leq j \leq m} (R_i \circ R'_j)$$

Komposition von Relationen

Die Komposition der Punkt Algebra Relationen mit Branching Time wurde von Broxvall [Bro01] definiert und ist in Abbildung 2.3 definiert. Der linke Operand wird durch die Zeilen dargestellt. Die Spalten stellen den rechten Operanden dar.

$\circ$	$<$	$>$	$\parallel$	$=$
$<$	$<$	$< = >$	$< \parallel$	$<$
$>$	$< = \parallel >$	$>$	$\parallel$	$>$
$\parallel$	$\parallel$	$> \parallel$	$< = \parallel >$	$\parallel$
$=$	$<$	$>$	$\parallel$	$=$

Tabelle 2.3: Kompositionstabelle für Punkt Algebra mit Branching Time

2.4 Intervall Algebra mit Branching Time

Im vorherigen Abschnitt wurde beschrieben, wie mit temporaler Logik Zeitpunkte in Relation gesetzt werden. Es gibt Fragestellungen, die es nötig machen, die Zeiträume zwischen zwei Zeitpunkten zu betrachten. Zum Beispiel wenn die Durchführung mehrerer Aktivitäten in Relation gesetzt werden soll.

Der Zeitraum zwischen zwei Zeitpunkten wird auch als ein temporales Intervall bezeichnet (in dieser Arbeit werden die Begriffe temporales Intervall und Intervall gleichbedeutend verwendet). Zur Beschreibung der Relation zweier Intervalle hat Allen die Intervall Algebra beschrieben [All83]. Die formale Beschreibung der Intervall Algebra unterscheidet sich zur Punkt Algebra nur in der Bedeutung der Knoten und der Definition des Relationen Set $\mathcal{S}$. Die Knoten eines Intervall Algebra Netzwerkes stellen die Intervalle dar, die Kanten der Constraints werden aus einer Menge von Relationen gebildet.

Initial ist ein Intervall Algebra Netzwerk ein vollständiger Graph, dessen Constraints mit $\Theta = (\mathcal{S}_{Iv}, i, j) : 1 < i < n, 1 < j < n$ initialisiert sind. Ein solches Constraint wird ebenfalls als T-Relation bezeichnet [RW04a].

Abbildung 2.4 zeigt ein Beispiel eines Intervall Algebra Netzwerkes und eine mögliche Abbildung des Graphes auf den Zeitstrahl.

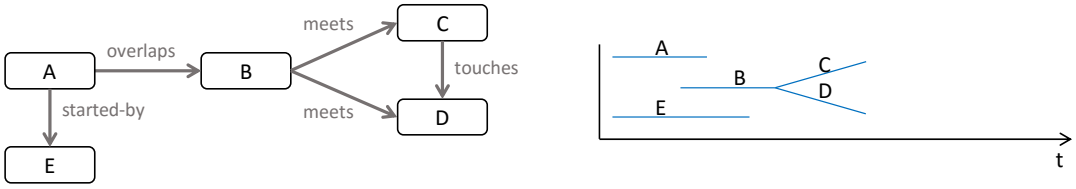


Abbildung 2.4: Beispiel eines Intervall Algebra Netzwerkes als Graph und eine mögliche Abbildung auf den Zeitstrahl.

Abbildung 2.5 beschreibt die linearen Relationen der Intervall-Algebra mit Branching Time (und deren Inversen), die ebenfalls Teil von $\mathcal{S}$ sind, mit den zugehörigen Beschreibungen in Punkt Algebra. Die Zeitpunkte eines Intervalls i werden mit $-i$ für den Startzeitpunkt und mit $+i$ für den Endzeitpunkt dargestellt.

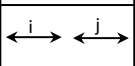
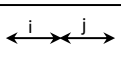
Relation & Symbol	i before j j after i	i meets j j met-by i	i overlaps j j overlapped-by i	i starts j j started-by i	i finishes j j finished-by i	i during j j contains i	i equals j
Symbol & Inverses Symbol	i {b} j j {bi} i	i {m} j j {mi} i	i {o} j j {oi} i	i {s} j j {si} i	i {f} j j {fi} i	i {d} j j {di} i	i {e} j
Punkt Relation	+i ≺ -j	+i = -j	-i ≺ -j, -j ≺ +I, +i ≺ +j	-i = -j, +i ≺ +j	-j ≺ -i, -i ≺ +j, +j = +i	-j ≺ -i, +i ≺ +j	-i = -j, +i = +j
Illustriertes Beispiel							

Abbildung 2.5: Die linearen Relationen der Punkt Algebra [All83]

Auch für Intervall Algebra können die Relationen zu Intervallen beschrieben werden, die in keiner linearen Relation zueinander stehen, da diese auf unterschiedlichen Pfaden des Zeitbaumes liegen. Die Intervall Algebra wurde von Ragni et al. um das Konzept der Branching Time erweitert [RW04a]. Es existiert nun keine lineare Zeit mehr, sondern die Zeit wird durch Branching Punkte in einen Baum eingeteilt. Ein Branching Punkt erzeugt zwei oder mehrere nachfolgende Pfade, deren Intervalle in keiner linearen Relation zu einander stehen. Weitere nicht-lineare Relationen werden benötigt, um die Relation von zwei Intervallen beschreiben, deren Branching Punkt zwischen oder vor den beiden Relationen liegt.

Die Abbildungen 2.6 und 2.7 beschreiben die nicht-linearen Relationen der Intervall-Algebra mit Branching Time (und deren Inversen), die ebenfalls Teil von $\mathcal{S}$ sind, mit den zugehörigen Beschreibungen in Punkt Algebra.

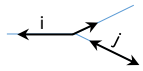
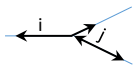
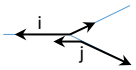
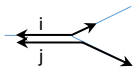
Relation & Symbol	i partially-before j j partially-after i	i partially-meets j j partially-met-by i	i partially-overlaps j j partially-overlapped-by i	i partially-starts j j partially-started-by i
Symbol & Inverses Symbol	i {pb} j j {pbi} i	i {pm} j j {pmi} i	i {po} j j {poi} i	i {ps} j j {psi} i
Punkt Relation	-j || +i , ∃t(-i ≺ t ∧ t ≺ +i ∧ t ≺ -j)	-i ≺ -j , -j ≺ +i, +i || +j, ¬∃t(-j ≺ t ∧ t ≺ +i ∧ t ≺ +j)	-i ≺ -j, +i || +j , ∃t(-j ≺ t ∧ t ≺ +i ∧ t ≺ +j)	-i = -j, +i || +j, ∃t(-i ≺ t ∧ t ≺ +i ∧ t ≺ +j)
Illustriertes Beispiel				

Abbildung 2.6: Die nicht-linearen Relationen der Punkt Algebra mit Branching Punkt zwischen den Intervallen [RW04a]

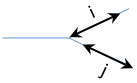
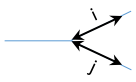
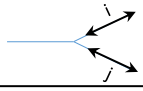
Relation & Symbol	i adjacent j j adjacent-by i	i touches j j partially-met-by i	i unrelated j
Symbol & Inverses Symbol	i {a} j j {ai} i	i {t} j	i {u} j
Punkt Relation	-i ≺ -j, -j || +i, ¬∃t(-i ≺ t ∧ t ≺ +i ∧ t ≺ -j)	-i = -j , +i || +j, ¬∃t(-i ≺ t ∧t ≺ +i ∧t ≺ +j)	-i || -j
Illustriertes Beispiel			

Abbildung 2.7: Die nicht-linearen Relationen der Punkt Algebra mit Branching Punkt vor den Intervallen [RW04a]

Desweiteren existieren noch die nicht-linearen Relationen *initally-before*, *initally-after*, *initally-meets*, *initally-met-by* und *initally-equals*.

2.4.1 Operationen der Intervall Algebra

Die Operationen für Schnitt, Inverse und Komposition der Constraints der Intervall Algebra werden in den folgenden Abschnitten definiert.

Schnitt von Constraints

Teil des Schnittes zweier Constraints sind die Relationen, die in beiden Constraints enthalten sind. Der Schnitt der Menge von Relationen von zwei Constraints ist wie folgt definiert.

$$(R_1, \ldots, R_n) \cap (R'_1, \ldots, R'_m) = \bigcup r \in \mathcal{S} : r \subseteq (R_1, \ldots, R_n) \text{ and } r \subseteq (R'_1, \ldots, R'_m)$$

Inverse von Constraints

Die Inverse eines Constraints wird aus den Inversen der einzelnen Relationen gebildet:

$$(R_1, \ldots, R_n)^{-1} = (R_1^{-1}, \ldots, R_n^{-1})$$

Inverse von Relationen

Komposition von Constraints

Die Komposition zweier Intervall Algebra Constraints beschreibt die Konkaternierung zweier Constraints. Die Komposition der Relationen der beiden Constraints berechnet sich wie folgt [RW04a]:

$$(R_1, \ldots, R_n) \otimes (R'_1, \ldots, R'_m) = \bigcup_{1 \leq i \leq n, 1 \leq j \leq m} (R_i \circ R'_j)$$

Komposition von Relationen

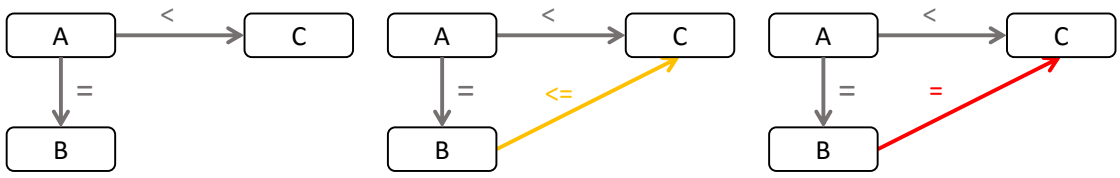
Die Komposition in linearer Intervall Algebra wurde von Allen beschrieben [All83]. Mit der Einführung der Intervall Algebra mit Branching Time wurde der Kompositionsoperator für Intervall Algebra Relationen durch den Kompositionsoperator für Intervall Algebra mit Branching Time ersetzt [RW04a; RW04b].

2.5 Constraint Propagation

Folgender Abschnitt beschreibt die Constraint Propagation oder auch Resoning für Punkt und Intervall Algebra. Dazu wird die Hülle eines temporalen Netzwerkes berechnet. Da die benötigten Algorithmen für Punkt und Intervall Algebra auf Constraint Probagation Problem (CSP) Algorithmen basieren, werden diese zusammen beschrieben.

2.5.1 Motivation der Constraint Propagation am Beispiel Punkt Algebra

In einem Punkt Algebra Problem sind meist nicht alle Constraints definiert oder alle Relationen-Sets der Constraints minimal definiert. Die Beispiele der Abbildungen 2.8 enthalten drei Variablen und zwei bzw. drei Constraints. In Beispiel 2.8.a ist das Constraint (T, B, C) (die T-Relation entspricht allen Relationen) nicht definiert und muss aus den Constraints $(\{=\}, A, B)$ und $(\{<\}, A, C)$ ermittelt werden. Das Beispiel 2.8.b enthält das Constraint $(\{<, =\}B, C)$. Aus dem Constraint $(\{<\}, A, C)$ ergibt sich allerdings, dass der Zeitpunkt A vor Zeitpunkt C stattfindet. Ohne die Relation $=$ wäre das Constraint $(\{<, =\}B, C)$ richtig. Das Constraint $(\{<, =\}B, C)$ ist also nicht minimal. Das Beispiel 2.8.c enthält das Constraint $(\{=\}B, C)$. Diese Relation $=$ steht im Widerspruch zu dem Constraint $(\{<\}, A, C)$, da die Zeitpunkte auf Grund des Constraints A und B gleich $(\{=\}, A, B)$ sind.



(a) Constraint nicht gesetzt **(b)** Constraint überspezifiziert **(c)** Constraint ist ein Widerspruch

Abbildung 2.8: Beispiele von Fällen, die eine Constraint Propagation notwendig machen.

2.5.2 Constraint Probagation Algorithmus nach Allen

Für die Intervall Algebra wurde von Allen ein Constraint Propagation Algorithmus beschrieben [All83], der im Algorithmus 2.1 dargestellt ist [VK13]. Der Algorithmus ähnelt den *arc consistency algorithm*, der erstmals von Mackworth vorgestellt wurde [Mac77].

Der Algorithmus von Allen berechnet die Hülle des Intervall Algebra Problems ausgehend von einem Delta der Constraints. Das Delta entspricht der Reduktion eines Constraints, wenn zum Beispiel ein Constraint geändert oder neu hinzugefügt wird. Initial besteht zwischen den Intervallen ein vollständiges temporales Netzwerk, dass mit der T-Relation initialisiert ist. Die T-Relation enthält alle Relationen der Algebra. Das temporale Netzwerk ist durch eine Adjazenzmatrix repräsentiert. Wird durch die Prozedur `Add` ein Constraint eingeschränkt, wird dieses Constraint einer FIFO-Schlange *Queue* hinzugefügt. Diese FIFO-Schlange stellt das Delta zwischen der letzten Berechnung der Hülle und dem aktuellem Zustand des Intervall Algebra Problems dar.

Die Prozedur `Closure` berechnet die Hülle für das aktuelle Delta indem solange über die FIFO-Schlange *Queue* iteriert wird, bis sich keine Änderungen mehr im Interval Algebra Netzwerk

ergeben. Für jedes Paar $< i, j >$, das ein Constraint darstellt, wird für alle Intervalle k überprüft, ob sich die benachbarten Constraints $< i, k >$ und $< k, j >$ aus dem Constraint $< i, j >$ konstruieren lassen. Abbildung 2.9 zeigt die beiden Kanten, die jeweils aus dem Constraint $< i, j >$ konstruiert werden.

Die Notation für ein Knotenpaar $< i, j >$ ist die Kurzform eines Constraints (R, ij) und wird in der Literatur zur Beschreibung der Algorithmen verwendet, die eine Adjazenzmatrix verwenden. In diesem Abschnitt wird zur besseren Nachvollziehbarkeit die Notation der Knotenpaare für Constraints aus den Originalquellen verwendet.

(a) $< i, j >$ nach $< i, k >$ **(b)** $< i, j >$ nach $< k, j >$

Abbildung 2.9: Propagierung des Constraint $< i, j >$

Zur Durchführung der Konstruktion sind die folgenden Gleichungen nötig [All83; VK13; RW04a]. Die beiden Constraints $< i, k >$ und $< k, j >$ berechnen sich aus dem Schnitt des ursprünglichen Constraints mit der Komposition aus $< i, j >$ mit $< i, k >$ bzw. $< k, j >$.

$$tempIK = Table[i, k] + (Table[i, j] \times Table[j, k])$$
$$tempKJ = Table[i, k] + (Table[i, j] \times Table[j, k])$$

Ist der Schnitt aus $< i, j >$ und $< i, k >$ bzw. $< k, j >$ leer gilt $tempX = \emptyset$ und es liegt ein Widerspruch für $< i, j >$ vor. Das Intervall Algebra Problem ist somit nicht lösbar. Ergibt sich aus dem Schnitt aus $< i, j >$ und $< i, k >$ bzw. $< k, j >$ ein neues Constraint $< i, j >$ mit einer oder mehr Relationen wird das Paar $< i, j >$ an das Ende der FIFO-Schlange *Queue* angefügt, um die Auswirkungen des neuen Constraints $< i, j >$ zu berechnen.

Der Algorithmus von Allen terminiert, wenn sich keine Änderungen an den Constraints des Intervall Algebra Problems ergeben und die FIFO-Schlange *Queue* somit leer ist.

Korrektheit und Vollständigkeit

Der Algorithmus von Allen zur Berechnung der Hülle eines Intervall Algebra Netzwerkes ist zwar korrekt. Allen weißt aber auch nach, dass der Algorithmus für Intervall-Algebra nicht vollständig ist [All83]. Dazu führt er ein Gegenbeispiel an, das zwar widerspruchsfrei hinsichtlich der Algebra ist, aber hinsichtlich der Abbildung auf einen Zeitstrahl inkonsistent ist.

Algorithmus 2.1 Constraint Propagation Algorithmus nach Allen [All83]

Table is a two-dimensional array indexed by intervals, in which Table[i,j] holds the relation between intervals i and j. Table[i,j] is initialized to the additive identity vector consisting of all thirteen simple relations; except for Table[i,i], which is intialized to (EQUALS).
Queue is a FIFO data structure that keeps track of pairs of intervals whose relation has been changed.
Intervals is a list of all intervals about which assertions have been made.

```
procedure ADD(R_{i,j})
    oldRel ← Table[i, j]
    Table[i, j] ← Table[i, j] + R_{i,j}
    if Table[i, j] ≠ oldRel then
        add pair < i, j > to Queue
    end if
end procedure
procedure CLOSURE
    while Queue ≠ ∅ do
        Get next < i, j > from Queue
        PROPAGATE(i,j)
    end while
end procedure
procedure PROPAGATE(i,j)
    for all interval k in Intervals do
        tempIK ← Table[i, k] + (Table[i, j] × Table[j, k])
        if tempIK = ∅ then
            signal contradiction
        end if
        if Table[i, k] ≠ tempIK then
            add pair < i, k > to Queue
        end if
        Table[i, k] = tempIK
        tempKJ ← Table[k, j] + (Table[k, i] × Table[i, j])
        if tempKJ = ∅ then
            signal contradiction
        end if
        if Table[k, j] ≠ tempKJ then
            add pair < k, j > to Queue
        end if
        Table[k, j] = tempKJ
    end for
end procedure
```

2.5.3 Anwendbarkeit des Algorithmus von Allen auf Punkt Algebra

Die Intervall Algebra ähnelt sich formal der Punkt Algebra darin, dass beide Algebren ein Knoten-Kanten-Modell darstellen, dessen Kantengewichte temporale Relationen darstellen. Die Kantengewichte sind jeweils Relationen. Der semantische Unterschied der Algebren liegt in der temporalen Unterschiedlichkeit von Intervall und Zeitpunkt, sowie dem Set an Relationen. Vilain et al. haben beschrieben, wie sich die Intervall Algebra in Punkt Algebra überführen lässt [VKv89; VK13].

Vilain et al. führen den Nachweis für folgendes Theorem:

> **Theorem 5:** The constraint propagation algorithm [...] computes the closure of assertions in the continuous endpoint algebra. *Vilain et al. [VK13]*

Mit dem in Kapitel 2.3.1 beschriebenen Kompositionsoperator für Punkt Algebra Relationen lässt sich der Algorithmus von Allen auf Punkt Algebra anwenden, indem die Intervalle Zeitpunkten entsprechen und die Relationen der Punkt Algebra verwendet werden.

2.5.4 Komplexität des Algorithmus von Allen für Punkt und Intervall Algebra

Vilain et al. haben gezeigt, dass die Komplexität des Algorithmus von Allen O(n^3) beträgt [VK13].

Der FIFO-Schlange *Queue* im Algorithmus 2.1 können maximal O(n^2) Paare $< i, j >$ hinzugefügt werden, da maximal n^2 mögliche Relationen bestehen. Die Prozedur `Propagate` wird daher maximal n^2 mal durch die Prozedur `Closure` aufgerufen.

In der Prozedur `Propagate` wird die Schleife für jeden Knoten (Intervall oder Zeitpunkt) einmal durchlaufen. Die Anzahl der Relationen einer Algebra $|\mathcal{S}|$ ist gleich der maximalen Anzahl an möglichen Änderungen an einem Constraint und somit das Maximum wie oft ein Constraint durch die Prozedur `Propagate` der FIFO-Schlange *Queue* hinzugefügt wird. Für die Prozedur `Propagate` ergibt sich somit eine Komplexität von O($n + |\mathcal{S}|$).

Die Tabelle 2.4 zeigt die Komplexität für eine generische Anwendung des Algorithmus von Allen sowie des Algorithmus für Punkt und Intervall Algebra mit Branching Time.

Generisch	$O(n^2) + O(n + \|\mathcal{S}\|) = O(n^3 + \|\mathcal{S}\|)$
Punkt Algebra	$O(n^2) + O(n + 4) = O(n^3)$
Intervall Algebra	$O(n^2) + O(n + 24) = O(n^3)$

Tabelle 2.4: Komplexität des Algorithmus von Allen für verschiedene Algebren

Vilain et al. zeigen auch, dass die Berechnung der Hülle für Intervall Algebra ein NP-vollständiges Problem ist [VK13]. Das Problem der Berechnung der Hülle lässt sich auf das 3-SAT-Problem reduzieren, das NP-hart ist. Der Nachweis, dass die Berechnung der Hülle NP-vollständig ist, wird erbracht indem gezeigt wird, dass eine zufällig gewählte Lösung der Berechnung der Hülle in polynomieller Zeit gelöst werden kann.

3 Repräsentation und Vergleich durch temporale Aktivitätenzustandsnetzwerke

Dieses Kapitel beschreibt das Konzept einer Repräsentation und dem Vergleich des Kontrollflusses von BPEL Prozessen durch temporale Aktivitätenzustandsnetzwerke. Aus dem theoretischem Hintergrund des vorgehenden Kapitels werden Vorüberlegungen für ein Konzept abgeleitet. Diese Vorüberlegungen dienen zur Ableitung von Anforderungen an ein Konzept zum Vergleich des Kontrollflusses von BPEL Prozessen. Anschließend wird der Lösungsansatz und ein detailliertes theoretisches Konzept zur Umsetzung des Lösungsansatzes beschrieben.

3.1 Problemanalyse und Vorüberlegungen

Wagner et al. beschreiben in [WKL12] einen Ansatz zur Validierung von Prozesskonsolidierungsoperationen. Die Autoren betrachten nur den Zustand der Ausführung einer Basis-Aktivität. Die strukturierenden Aktivitäten eines BPEL Prozesses stellen ebenfalls eine Kontrollflussbeschreibung dar und müssen ebenfalls bei einem Vergleich des Kontrollflusses berücksichtigt werden.

Die Steuerung des Kontrollflusses durch hierarchische Aktivitäten ist abhängig von deren Zustand. Zum Beispiel wird durch die Aktivierung des Zustandes $faulthandling$ eines Scopes die Aktivitäten eines Fault Handlers ausgeführt.

Mit BPEL Prozessen werden auch parallele Abläufe modelliert. Aus dem Verhalten von Geschäftsprozessen lassen sich verschiedene Traces von Aktivitätenzuständen ableiten. Aus den Traces der Aktivitätenzustände ergeben sich temporale Abhängigkeiten, die durch temporale Relationen dargestellt werden können. Temporale Algebren ermöglichen die Beschreibung von temporalen Relationen. Ein Vorteil der temporalen Algebren ist, dass sich durch Constraint Satisfaction Problem (CSP) Algorithmen Relationen zwischen Punkten herleiten lassen, die nicht explizit definiert sind. Dieses Vorgehen entspricht der Berechnung der Hülle eines Graphen und wird auch Reasoning genannt. Somit können auch Relationen in einem Baum von Traces bestimmt werden, die nicht explizit durch den Trace beschrieben sind.

Die Intervall-Algebra eignet sich zur Abbildung der verschiedenen Zustände einer Aktivität, da ein Intervall den Start- und Endzeitpunkt darstellt in der sich ein System in einem Zustand befindet. Allerdings ist der CSP-Algorithmus für die Intervall Algebra von Allen [All83] nicht vollständig. Deshalb eignet sich die Intervall Algebra nicht für Vergleiche oder Verifikationen mit hohen Anforderungen an die Vollständigkeit des Algorithmus [VK13]. Für die Umsetzung einer temporalen Repräsentation von BPEL Prozessen muss daher eine andere temporale Algebra gewählt werden.

Aus diesem Grund wird für diese Arbeit die Punkt Algebra gewählt. Die Intervall Algebra lässt sich teilweise in Punkt Algebra überführen [RW04a]. Daher kann die Intervall Algebra als theoretische Grundlage zur Erstellung von Punkt Algebra Netzwerken für BPEL Prozesse dienen.

3.2 Anforderungen an eine Äquivalenz-Analyse

An eine Äquivalenz-Analyse von BPEL Prozessen mit einer temporalen Repräsentation werden folgende Anforderungen gestellt:

- Betrachtung der Zustände jeder einzelnen Aktivität eines BPEL Prozesses
- Beschreibung der Zustände und des Kontrollflusses mit Punkt Algebra.
- Vergleich des Kontrollflusses zweier BPEL Prozesse hinsichtlich ausgewählter Aktivitäten und Aktivitätenzustände transparent von den strukturierender Aktivitäten.

3.3 Lösungsansatz

Der Prozess zum Vergleich von zwei Prozessen gliedert sich in zwei Phasen, wie in Abbildung 3.1 dargestellt. In der ersten Phase werden die beiden BPEL Prozesse in ein Punkt Algebra Netzwerk überführt. In der zweiten Phase werden die beiden Punkt Algebra Netzwerke anhand der Relationen zwischen den Ausführungszuständen der Basis-Aktivitäten verglichen.

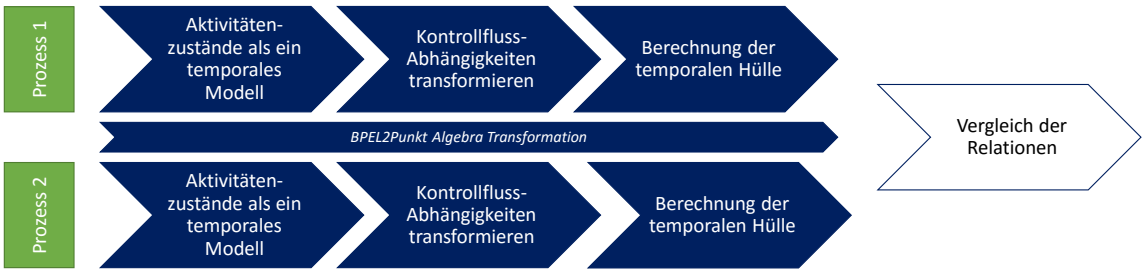


Abbildung 3.1: Ablauf der Repräsentation und des Vergleichs von BPEL Prozessen

Die Transformation der Aktivitätenzustände umfasst für jede Aktivität des BPEL Prozesses einen Zustandsautomaten, der durch Punkt Algebra Zeitpunkte und Constraints modelliert wird. Die Semantik der strukturierenden Aktivitäten, die auf den Zuständen der Aktivitäten basiert, wird ebenfalls durch Constraints abgebildet. Diese Semantik ist in der BPEL Spezifikation definiert [OAS07]

Liegt für jeden Prozess ein Punkt Algebra Netzwerk mit berechneter Hülle vor können die Relationen zwischen den Aktivitätenzuständen der Basis-Aktivitäten verglichen werden.

3.4 Aktivitätenzustände

Zur temporalen Modellierung der Zustände einer Aktivität wird ein Zustandsmodell der Aktivität benötigt, das in diesem Abschnitt beschrieben wird.

Eine Aktivität durchläuft zur Laufzeit verschiedene Zustände. Zum Beispiel von der Initialisierung über die Ausführung zum erfolgreichen Abschluss der Aktivität. Kopp et al. haben ein Zustandsmodell der Zustände einer BPEL Aktivität entworfen [KHK+11]. In der Arbeit von Kopp et al. wird ein abgeändertes Zustandsmodell einer Aktivität verwendet. Abbildung 3.2 zeigt das Zustandsmodell für eine Aktivität mit Fault Handler. Das Zustandsmodell das in dieser Arbeit verwendet wird, weicht in dem Zustand *aborted* von dem Zustandsmodell von Kopp ab. Terminiert die Aktivität vor der Ausführung, transferiert die Aktivität in den Zustand *aborted*. Terminiert die Aktivität nach der Ausführung, transferiert das Zustandsmodell in den Zustand *terminating*. Diese Aufteilung ist nötig, da in der Punkt Algebra keine Zustandsmodelle abgebildet werden können, die gerichtete oder ungerichtete Zyklen enthalten. Im Zustandsmodell von Kopp et al. bilden die Zustände *init*, *terminating* und *executing* einen Zyklus.

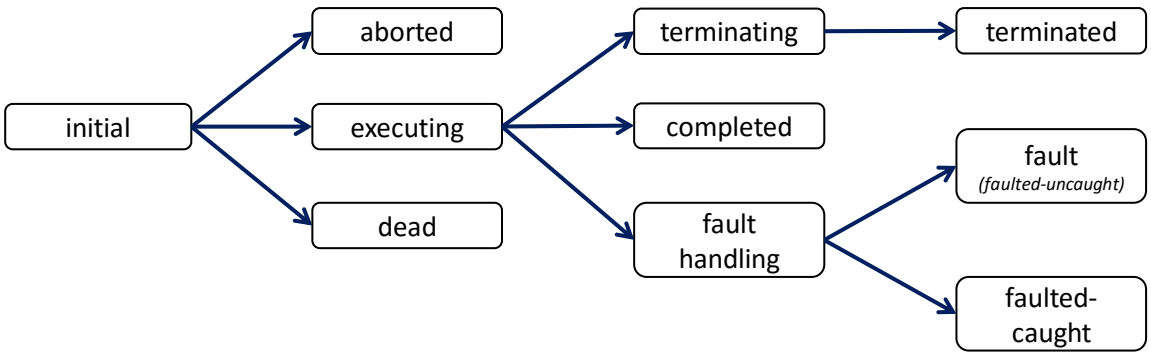


Abbildung 3.2: Das Basis-Zustandsmodel mit Fault Handler

Die Semantik des Zustandsmodelles für eine Aktivität ist folgendermaßen definiert [KHK+11]:

inital Der Zustand $inital$ ist Startzustand der Aktivität und wird bei der Initialisierung der übergeordneten Aktivität aktiv.

aborted Die Aktivität wurde vor der Ausführung terminiert.

dead Die Aktivität wurde durch die Death-Path-Elimination der Ausführungsengine als dead makiert.

executing Die Aktivität wird ausgeführt.

completed Die Aktivität wurde erfolgreich ausgeführt. Ausgehende Links werden durch die Ausführungsengine bearbeitet.

terminating Die Aktivität terminiert alle Kind-Aktivitäten.

terminated Alle Kind-Aktivitäten und die Aktivität selbst sind terminiert.

fault handling Nach einem Fehlerfall sind Fault Handler der Aktivität aktiv.

fault Nach einem Fehlerfall wurde der Fehler an die übergeordnete Aktivität propagiert.

fault caught Nach einem Fehlerfall wurde der Fehler nicht an die übergeordnete Aktivität propagiert.

Für Basis-Aktivitäten, die keinen Fault Handler enthalten, sind die Zustände $faulthandler$ und $faultcaught$ nicht enthalten. Stattdessen existiert ein Zustandsübergang zwischen den Zuständen $executing$ und $fault$.

3.5 Transformation eines BPEL Prozesses in ein temporales Aktivitätenzustandsnetzwerk

Das Zustandsmodell des Abschnitts 3.4 wird nun verwendet, um die Startzeitpunkte aller Aktivitätenzustände eines BPEL Prozesses in temporale Relationen zu setzen. Dazu wird das Konzept des geschichteten Punkt Algebra Netzwerkes eingeführt. Alle Aktivitätenzustände, die zu einer Aktivität gehören, bilden eine Gruppe. Da diese Gruppen sich durch die Struktur eines BPEL Prozesses in Schichten anordnen lassen, wird in dieser Arbeit eine Gruppe von Zeitpunkten, die Startzeitpunkte von Zuständen einer Aktivität sind, als eine Schicht bezeichnet.

Abbildung 3.3 zeigt zur Veranschaulichung für einige Aktivitäten jeweils eine reduzierte Schicht eines Punkt Algebra Netzwerkes. Die Relationen zwischen Aktivitätenzuständen der gleichen Aktivität werden als Intra-Aktivity-Constraints bezeichnet. Sind mehrere Aktivitäten Kinder der gleichen Vater-Aktivität, werden deren Schicht im Punkt Algebra Netzwerk auf der gleichen Ebene angeordnet. Im Beispiel der Abbildung 3.3 enthält die Aktivität i ein Kind und die Aktivität $i + 1$ zwei Kinder $i + 2a$ und $i + 2b$. Für jede Aktivität wurde eine Schicht im Punkt Algebra Netzwerk erstellt.

Der Kontrollfluss eines BPEL Prozesses, der durch strukturierenden Aktivitäten definiert wird, wird durch Inter-Aktivity-Constraints abgebildet. Diese beschreiben die Relation zwischen zwei Startzeitpunkten von Zuständen unterschiedlicher Aktivitäten.

Es sind also nur Relationen zwischen zwei benachbarten Schichten im geschichteten Punkt Algebra Netzwerk definiert. Wird ein CSP Algorithmus ausgeführt, der die Hülle berechnet, lassen sich auch Relationen zwischen Startzeitpunkten von Aktivitätenzuständen ermitteln, die nicht direkt auf benachbarten Schichten liegen.

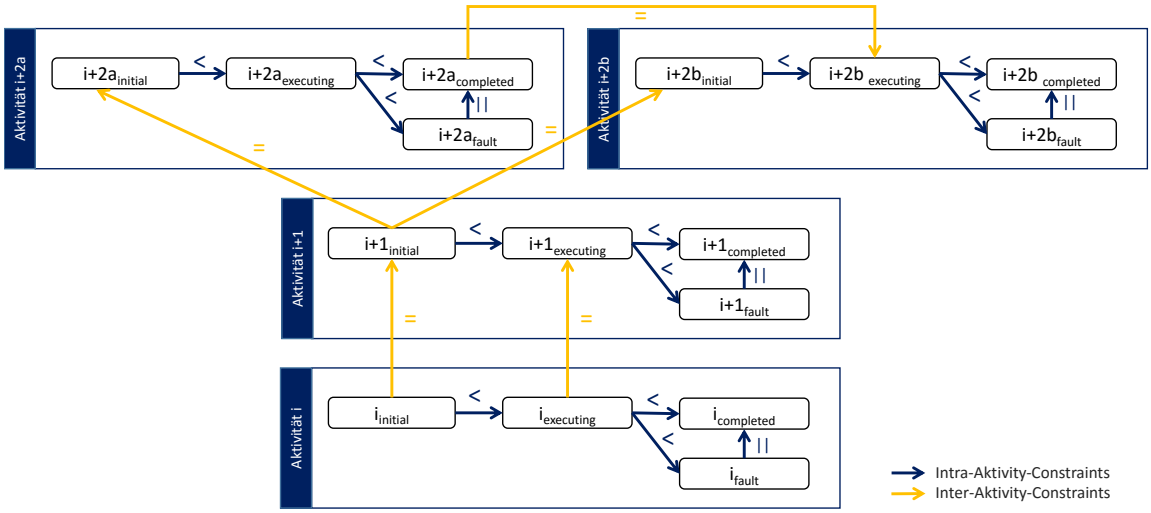


Abbildung 3.3: Geschichtetes Punkt Algebra Netzwerk. Jede Schicht enthält ein Netzwerk aus Intra-Aktivity-Constraints einer Aktivität des BPEL Prozesses.

Der Algorithmus 3.1 beschreibt die Transformation eines BPEL Prozesses in ein geschichtetes Punkt Algebra Netzwerk. Die Aktivitäten des BPEL Prozesses werden in einer Tiefensuche traversiert. Die Prozedur `Transform` fügt dem Punkt Algebra Netzwerk für jeden Zustand der Kind-Aktivitäten der Aktivität `act` einen Zeitpunkt hinzu, der den Startzeitpunkt des Zustandes darstellt. Zwischen diesen Startzeitpunkten von Zuständen werden die Intra-Activity-Constraints gemäß der Semantik einer Aktivität erstellt. Anschließend werden die Inter-Aktivity-Constraints zwischen den Startzeitpunkten der Aktivität *act* und den Startzeitpunkten der Aktivitätenzustände der Kinder der Aktivität *act* erstellt. Anschließend werden die Inter-Activity-Constraints zwischen den Startzeitpunkten der Zustände aller Kindaktivitäten von *act* hinzugefügt. Mit dem rekursiven Aufruf der Prozedur `Transform` wird das Punkt Algebra Netzwerk um die Zeitpunkte und Constraints der restlichen Aktivitäten vervollständigt.

Die Prozeduren aus dem Algorithmus 3.1 sind in Tabelle 3.1 definiert.

Algorithmus 3.1 Transformation eines BPEL Prozesses in Punkt Algebra

Table is a two-dimensional array indexed by time points, in which Table[i,j] holds the relation between time points i and j. Table[i,j] is initialized to the additive identity vector consisting of all relations; except for Table[i,i], which is intialized to (EQUALS).

```
procedure TransformProcess(P)
    act ← a ∈ P
    Transform(act)
    Closure
end procedure
procedure Transform(A)
    statesA ← GetStates(A)
    CreateConstraints(statesA, statesA)
    for all a ∈ A do
        statesC ← GetStates(a)
        for all s ∈ statesC do
            AddTimePointForState(s)
        end for
    end for
    for all a ∈ A do
        statesC ← GetStates(a)
        CreateConstraints(statesA, statesC)
        for all a2 ∈ A do
            if a ≠ a2 then
                statesC2 ← GetStates(a2)
                CreateConstraints(statesC, statesC2)
            end if
        end for
        Transform(a)
    end for
end procedure
procedure CreateConstraints(states1, states2)
    const ← GetConstraints(states1, states2)
    for all C_{i,j} ∈ const do
        AddConstraint(C_{i,j})
    end for
end procedure
procedure AddConstraint(C_{i,j})
    for all R_{i,j} ∈ C_{i,j} do
        Table[i, j] ← Table[i, j] + R_{i,j}
    end for
end procedure
```

`GetStates`	Definiert die Menge der Zustände für diese Aktivität gemäß dem Zustandsmodell aus Kap. 3.4
`AddTimePointForState`	Erstellt für einen Aktivitätenzustand einen Startzeitpunkt im Punkt Algebra Netzwerk
`GetConstraints`	Definiert die Constraints, die zwischen den Startzeitpunkten von zwei Mengen an Aktivitätenzuständen liegen.
`Closure`	Berechnet die Hülle des Punkt Algebra Netzwerkes. Diese Prozedur ist im Algorithmus von Allen in Algorithmus 2.1 definiert.

Tabelle 3.1: Prozeduren des Algorithmus 3.1

Nach Abschluss der Prozedur `TransformProcess`, wurden für alle Aktivitätenzustände Startzeitpunkte im Punkt Algebra Netzwerk erstellt. Zwischen den Startzeitpunkten der Aktivitätenzustände einer Aktivität wurden die nötigen Constraints definiert. Gemäß der Spezifikation von BPEL wurden für die strukturierenden Aktivitäten weitere Constraints erstellt, die den Kontrollfluss im Punkt Algebra Netzwerk modellieren. Durch die Berechnung der Hülle des Punkt Algebra Netzwerkes wurden die Auswirkungen der definierten Constraints auf alle weiteren Constraints (die noch die T-Relation enthalten) berechnet. Nach Abschluss der Berechnung der Hülle liegt ein konsistentes Punkt Algebra Netzwerk zur weiteren Analyse vor.

3.6 Vergleich von temporalen Aktivitätenzustandsnetzwerken

Liegt von zwei Prozessen eine Punkt Algebra Repräsentation mit berechneter Hülle vor, können die beiden Punkt Algebra Netzwerke anhand der Relationen verglichen werden.

Der Kontrollfluss von zwei BPEL Prozessen ist gleich, wenn die Basis-Aktivitäten der beiden Prozesse die gleichen Traces erfüllen. Diese Anforderung ist gegeben, wenn die folgenden Bindungen erfüllt sind:

1. Die temporalen Aktivitätenzustandsnetzwerke der beiden Prozesse enthalten korrespondierende Startzeitpunkte für alle $executing$-Zustände der Basis-Aktivitäten.
2. Zwischen allen $executing$ Aktivitätenzuständen der Basis-Aktivitäten eines BPEL Prozesses existieren in beiden temporalen Aktivitätenzustandsnetzwerken die gleichen Relationen.

Die Abbildung 3.4 zeigt den Vergleich des Kontrollflusses des Prozesses $P1$ mit den Prozessen $P2$ und $P2$. Von allen drei Prozessen liegen die temporalen Aktivitätenzustandsnetzwerke $PA(P1)$, $PA(P2)$ und $PA(P3)$ vor. Die Abbildung zeigt die Constraints der für den Vergleich relevanten Zustände $executing$ der Basis-Aktivitäten A, B, C, und D.

Der Kontrollfluss der Prozesse $P1$ und $P2$ ist nicht äquivalent. Die Bedingung 1 ist nicht erfüllt, da dem temporalen Netzwerk $PA(P2)$ der Startzeitpunkt des Zustandes $executing$ der Aktivität B fehlt. Die zweite Bedingung ist dagegen erfüllt, da die vorhandenen Constraints in beiden temporalen Netzwerken gleich sind.

Der Kontrollfluss der Prozesse $P1$ und $P3$ ist ebenfalls nicht äquivalent. Die Bedingung 1 ist erfüllt, da in beiden Prozessen jeweils äquivalente Startzeitpunkte für den Zustand $executing$ der Aktivitäten A bis D enthalten sind. Die Bedingung 2 ist nicht erfüllt, da Prozess $P3$ das Constraint $(\{<\}, C_{-exe}, D_{-exe}) \in PA(P3)$ von dem Constraint $(\{\|\}, C_{-exe}, D_{-exe}) \in PA(P1)$ abweicht.

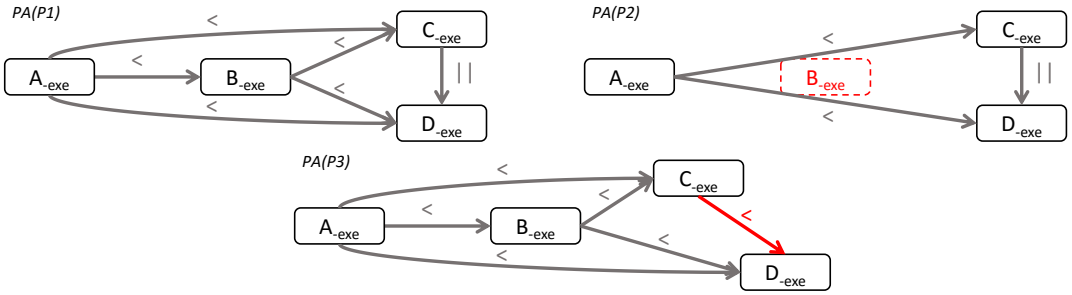


Abbildung 3.4: Beispiel für einen inkonsistenten Kontrollfluss. Das temporale Netzwerk $PA(P2)$ unterscheidet sich von dem temporalen Netzwerk $PA(P1)$ durch den fehlenden Zeitpunkt B_{-exe} vom temporalen Netzwerk $PA(P21)$. Das temporale Netzwerk $PA(P3)$ unterscheidet sich von dem temporalen Netzwerk $PA(P1)$ durch das Constraint $(\{<\}, C_{-exe}, D_{-exe}) \in PA(P3)$.

Der Algorithmus 3.2 beschreibt den Vergleich von zwei BPEL Prozessen, die bereits als temporales Aktivitätenzustandsnetzwerk vorliegen, auf Grundlage der in diesem Abschnitt definierten Bedingungen. Die Prozeduren des Algorithmus sind in Tabelle 3.2 definiert.

Zunächst werden die Matrizen *Table1* und *Table2* nach ihren Knoten sortiert und alle Knoten entfernt, die nicht den Zustand $executing$ referenzieren. Anschließend wird mit der Prozedur `CheckTimePoints` überprüft, ob in beiden temporalen Aktivitätenzustandsnetzwerken ausschließlich korrespondierende Zeitpunkte vorliegen. Ein Zeitpunkt korrespondiert in beiden Punkt Algebra Netzwerken, wenn ein Zeitpunkt den Startzeitpunkt der gleichen Aktivitätenzustände darstellt. Nach dieser Überprüfung der Startzeitpunkte wird überprüft, ob zwischen den Startzeitpunkten aller Aktivitätenzustände die gleichen Relationen vorliegen.

Algorithmus 3.2 Vergleich von zwei BPEL Prozessen als temporales Aktivitätenzustandsnetzwerk

Table1 is a two-dimensional array indexed by time points, in which Table1[i,j] holds the relation between time points i and j. Table1[i,j] contains a vector of relations. Table1[i,j] is initialized with relations created by a BPEL to temporal activity state network transformation of a process 1.
Table2 is a two-dimensional array indexed by time points, in which Table2[i,j] holds the relation between time points i and j. Table1[i,j] contains a vector of relations. Table2[i,j] is initialized with relations created by a BPEL to temporal activity state network transformation of a process 2.

```
procedure EQUALS
    Temp1 ← SORTANDFILTER(Table1)
    Temp2 ← SORTANDFILTER(Table2)
    tp1 ← GETTIMEPOINTS(Temp1)
    tp2 ← GETTIMEPOINTS(Temp2)
    if ¬CHECKTIMEPOINTS(tp1, tp2) then
        return false
    end if
    if ¬CHECKTIMEPOINTS(tp2, tp1) then
        return false
    end if
    for i ← 1, n do
        for j ← 1, n do
            if Temp1[i, j] ≠ Temp2[i, j] then
                return false
            end if
        end for
    end for
    return true
end procedure
procedure CHECKTIMEPOINTS(timepoints1, timepoints2)
    for all t1 ∈ timepoints1 do
        for all t2 ∈ timepoints2 do
            if ¬ CORRESPONDS(t1, t2) then
                return false
            end if
        end for
    end for
    return true
end procedure
```

`SortAndFilter`	Sortiert die Spalten und Zeilen einer Matrix nach den Bezeichnern der Zeitpunkte. Alle Zeitpunkte, die nicht den Zustand $executing$ oder eine strukturierende Aktivität referenzieren werden aus der Adjazenzmatrix entfernt.
`GetTimePoints`	Rückgabe aller Zeitpunkte, die in einer Matrix verwaltet werden.
`Corresponds`	Rückgabe von `true`, wenn zwei Startzeitpunkte für den gleichen Aktivitätenzustand sind.

Tabelle 3.2: Prozeduren des Algorithmus 3.2

3.7 Betrachtung der Gesamt-Komplexität

Dieser Abschnitt betrachtet die Komplexität für einen Prozess mit n Aktivitäten und m Links. Für jede Aktivität wird zur Erstellung des Punkt Algebra Netzwerkes ein Transformationsschritt ausgeführt. Für Kontrollflussabhängigkeiten wird jeweils ein Transformationsschritt pro Link ausgeführt. Daraus ergibt sich für die Transformation eine Komplexität von $O(n + m)$ für n Aktivitäten mit m Links. Die anschließend ausgeführte Berechnung der Hülle des Punkt Algebra Netzwerkes beträgt $O(n^3)$ [VK13]. Daraus folgt eine Komplexität von $O(n^3) + O(n + m) = O(n^4 + m)$.

4 Temporale Modelle

In Kapitel 3 wurde das Konzept eines temporalen Aktivitätenzustandsnetzwerkes für BPEL Prozesse beschrieben. In diesem Kapitel wird die Repräsentation von BPEL Aktivitäten durch temporale Aktivitätenzustandsnetzwerke und die Definition von Constraints zwischen Startzeitpunkten von Aktivitätenzuständen unterschiedlicher Aktivitäten beschrieben.

4.1 Temporale Aktivitätenzustandsnetzwerke

Die folgenden temporalen Modelle beschreiben die Zustände einer Aktivität als Punkt Algebra Netzwerk. Das Punkt Algebra Netzwerk enthält für jeden Zustand der Aktivität einen Zeitpunkt, der den Startzeitpunkt des Zustandes darstellt. Aus den Zustandsübergängen des Zustandsautomaten ergibt sich die temporale Reihenfolge der Startzeitpunkte der Zustände (in den Abbildungen durch blaue Relationen dargestellt).

Die Zustandsübergänge im Zustandsautomaten sind exklusiv. Das heißt, es wird immer nur ein Nachfolgezustand aktiviert. Aus der exklusiven Verzweigung im Zustandsautomaten ergeben sich mehrere mögliche Ausführungspfade. Das bedeutet, dass die Startzeitpunkte der Zustände nicht auf einer Zeitachse, sondern auf einem Zeitbaum angeordnet sind. Dieser Zeitbaum entspricht dem Konzept der Branching Time. Zwischen den Startzeitpunkten der Zustände, die über eine exklusive Verzweigung im Kontrollfluss aktiviert werden, ist im temporalen Aktivitätenzustandsnetzwerk eine unrelated Relation modelliert (lila Relationen in den Abbildungen).

4.1.1 Temporale Repräsentation einer Aktivität

Abbildung 4.1 zeigt ein temporales Aktivitätenzustandsnetzwerk, das aus dem Zustandsmodell für BPEL Aktivitäten aus Kap. 3.4 abgeleitet wurde. Dieses temporale Aktivitätenzustandsnetzwerk wird zur Modellierung der Startzeitpunkte der Zustände von Basis Aktivitäten, Flows und Fault Handler verwendet.

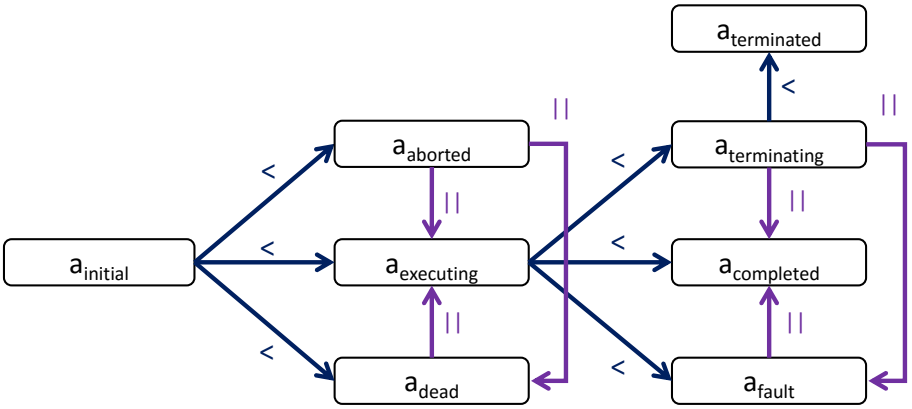


Abbildung 4.1: Temporales Aktivitätenzustandsnetzwerk für eine Aktivität

4.1.2 Temporale Repräsentation einer empty-Aktivität

Die empty-Aktivität implementiert keine Funktionalität. Die empty-Aktivität wird hauptsächlich zur Synchronisation von Kontrollfluss-Kanten genutzt. Abbildung 4.2 zeigt ein temporales Modell, dass vom temporalen Modell für eine Basisaktivität aus Abbildung 4.1 abgeleitet wurde.

Um die gleiche Menge der Startzeitpunkte in allen temporalen Modellen für Basis-Aktivitäten zur erhalten, enthält die empty-Aktivität einen abstrakten Zeitpunkt für den Start des Zustandes *executing*. Transferiert eine empty-Aktivität in den Zustand $executing$, transferiert die empty-Aktivität anschließend sofort in den Zustand $completed$. Daraus ergibt sich, dass die Startzeitpunkte der Zustände *executing* und *completed* einer empty-Aktivität a gleich sind: $(\{=\}, a_{-exe}, a_{-completed})$.

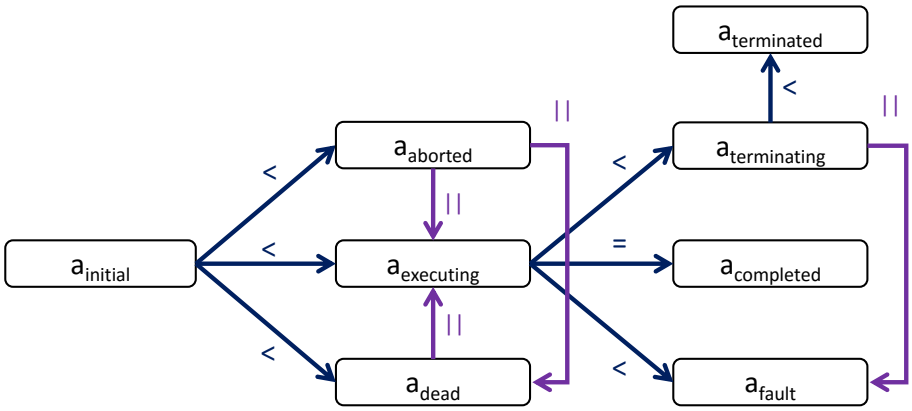


Abbildung 4.2: Temporales Aktivitätenzustandsnetzwerk für eine empty-Aktivität

Um die Widerspruchsfreiheit des temporalen Modells zu erhalten, sind keine unrelated Relationen mit dem Ziel des Startzeitpunktes des Zustandes *completed* nötig. Diese Relationen werden bereits durch die Constraints $(\{<\}, a_{-exe}, a_{-terminating})$ und $(\{<\}, a_{-exe}, a_{-fault})$ abgebildet und durch die Berechnung der Hülle propagiert.

4.1.3 Temporale Repräsentation einer throw-Aktivität

Die throw-Aktivität startet in übergeordneten Scopes zugeordnete Fault Handler durch die Signalisierung eines Fehlers. Abbildung 4.3 zeigt das temporale Aktivitätenzustandsmodell einer throw-Aktivität. Dieses temporale Modell wurden vom temporalen Modell für eine Basisaktivität aus Abbildung 4.1 abgeleitet. In einem geschichteten temporalen Netzwerk (siehe Kap. 3) wird ein Fehler durch den Startzeitpunkt des Zustandes *fault* modelliert. Da die Ausführung einer throw-Aktivität dem Beginn eines Fehler-Signals entspricht, transferiert diese zum Zeitpunkt der Ausführung in den Zustand *fault*. Daraus ergibt sich, dass die Startzeitpunkte der Zustände *executing* und *fault* einer throw-Aktivität a gleich sind: $(\{=\}, a_{-exe}, a_{-fault})$. Eine throw-Aktivität deshalb erreicht nie den Zustand $completed$, somit wird an dieser Stelle der Kontrollfluss eines Flow unterbrochen.

Analog zur empty-Aktivität entfallen unrelated-Relationen.

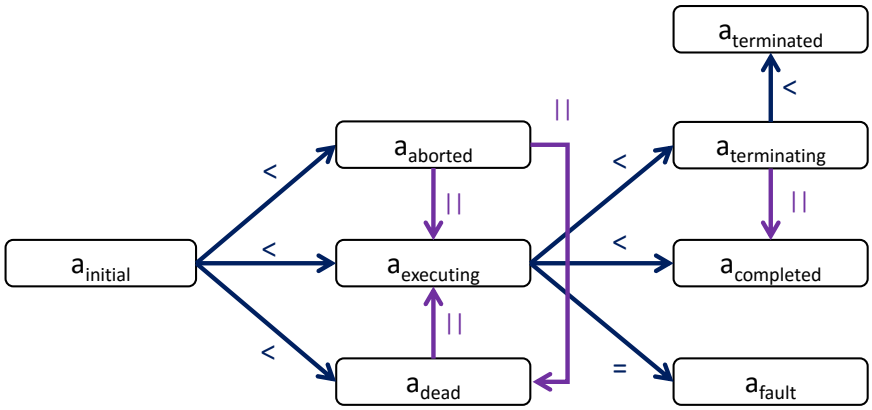


Abbildung 4.3: Temporales Aktivitätenzustandsnetzwerk für eine throw-Aktivität

4.1.4 Temporale Repräsentation einer Aktivität mit Fault Handler

Abbildung 4.4 zeigt das temporale Modell das für Scopes verwendet wird (Compansation Handler und Event Handler sind kein Teil dieser Arbeit). Das Netzwerk enthält einen Zeitpunkt zum Start des Fault Handlers und jeweils einen Zeitpunkt, indem entweder ein gefangener Fehler oder ein Fehler signalisiert wird.

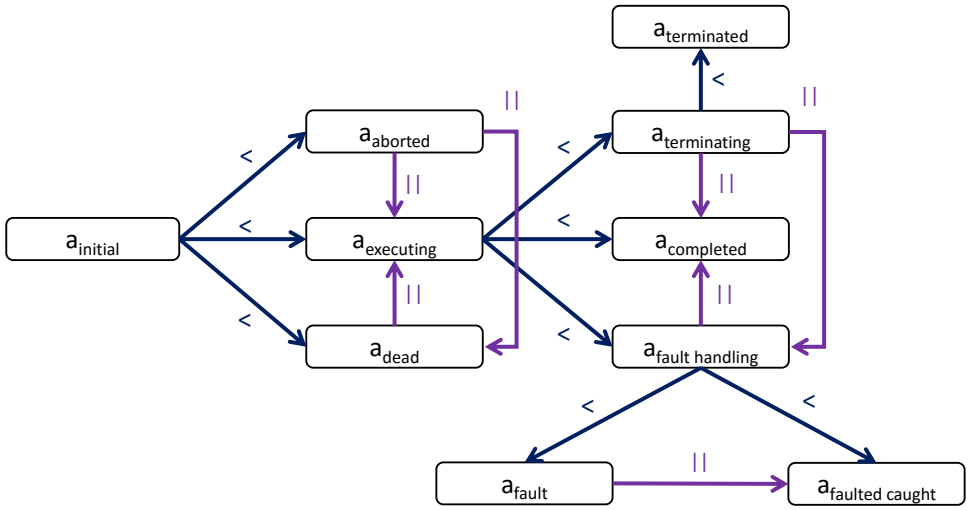


Abbildung 4.4: Temporales Aktivitätenzustandsnetzwerk für eine Aktivität mit Fault Handler

4.2 Temporale Repräsentation von strukturierenden Aktivtäten

Die strukturierenden Aktivitäten eines BPEL Prozesses beschreiben den Kontrollfluss im Standard- und Fehlerfall. Der Kontrollfluss hängt vom Zustand einer strukturierenden Aktivität und seinen Kindaktivitäten ab. Im temporalen Netzwerk wird der Kontrollfluss durch Relationen zwischen den Startzeitpunkten von Zuständen einer strukturierenden Aktivität und seinen Kindaktivitäten beschrieben.

Die folgenden Abschnitte beschreiben die temporale Verknüpfung der strukturierenden Aktivitäten Flow, Scope und Fault Handler anhand von Matrizen, die über die Startzeitpunkte der Aktivitätenzustände von zwei Aktivitäten aufgespannt sind. Leere Zellen der Tabelle bedeuten, dass für diese Kante kein Constraint definiert wird. Diese Constraints werden durch die Berechnung der Hülle bestimmt. Die temporalen Relationen wurden aus der Spezifikation von BPEL abgeleitet [OAS07].

4.2.1 Flow mit Aktivität

Die Repräsentation eines Flows besteht aus dem temporalen Aktivitätenzustandsnetzwerk einer Aktivität aus Abbildung 4.1 und der temporalen Verknüpfung mit den Startzeitpunkten der Aktivitätenzustände der Kindaktivität, beschrieben in Tabelle 4.1. Dazu gelten folgende temporalen Abhängigkeiten zwischen dem Flow und den Kindaktivitäten:

- Der Flow und seine Kindaktivitäten transferieren gleichzeitig in den Zustand *init*.

- Transferiert der Flow in den Zustand *aborted*, transferieren alle Kindzustände ebenfalls zum gleichen Zeitpunkt in den Zustand *aborted*.
- Transferiert der Flow in den Zustand *dead*, transferieren ebenfalls alle Kindzustände zum gleichen Zeitpunkt in den Zustand *dead*.
- Transferiert eine Kindaktivität in den Zustand *fault*, transferiert gleichzeitig der Flow in den Zustand *fault*. Kindaktivitäten transferieren in den Zustand *terminating*.
- Transferiert der Flow in den Zustand *terminating*, transferieren die Kindaktivitäten in den Zustand *aborted*, oder *terminating* (falls die Kindaktivität gerade ausgeführt wird).
- Eine Kindaktivität transferiert vor oder zum gleichen Zeitpunkt in den Zustand *terminated*, wie der Flow in den Zustand *terminated* transferiert.

		activity							
		init	aborted	executing	dead	fault	terminating	terminated	completed
flow	init	{=}							
	aborted		{=, ||}						
	executing								
	dead				{=}				
	fault					{=, ||}	{=, ||}		
	terminating		{=, ||}				{=, ||}		
	terminated							{>,=, ||}	
	completed								

Tabelle 4.1: Temporale Verknüpfung eines Flows mit den Kind-Aktivitäten

Für Kindaktivitäten, die keine eingehenden oder ausgehenden Links enthalten, gelten weitere Bedingungen:

1. Transferiert ein Flow f in den Zustand *executing*, transferieren die synchronisierten Kindaktivitäten a^{synch_1} ... a^{synch_n}, die keine eingehenden Links enthalten in den Zustand *executing*: $(\{=\}, f_{-exe}, a^{synch_i}_{-exe})$.
2. Der Flow f transferiert in den Zustand *completed*, zum gleichen Zeitpunkt oder nachdem eine Kindaktivität a^{synch_i}, die keine ausgehenden Links enthält, in den Zustand *completed* transferiert: $(\{>, =\}, f_{-exe}, a^{synch_i}_{-exe})$.

Die Aktivität a ist über ausgehende Links den Aktivitäten a^{pred_1} ... a^{pred_n} verbunden. Für Aktivitäten die im Kontrollfluss der Aktivität a nachfolgen, gelten weitere Bedingungen:

3. Transferiert eine Aktivität in den Zustand *completed*, transferieren Aktivitäten, die direkt durch einen ausgehenden Link verbunden sind, zum gleichen Zeitpunkt in den Zustand *executing*: $(\{=\}, a_{-completed}, a^{pred_i}_{-exe})$

4. Transferiert eine Aktivität in den Zustand *dead*, transferieren Aktivitäten, die direkt durch einen ausgehenden Link verbunden sind, ebenfalls zum gleichen Zeitpunkt in den Zustand *dead*: $(\{=\}, a_{-dead}, a^{pred_i}_{-dead})$
5. Transferiert eine Aktivität in den Zustand *fault*, transferieren Aktivitäten, die direkt durch einen ausgehenden Link verbunden sind, zum gleichen Zeitpunkt in den Zustand *aborted*: $(\{=\}, a_{-fault}, a^{pred_i}_{-aborted})$
6. Wird die Aktivität a^{pred_i} parallel mit den anderen nachfolgenden Aktivitäten von a ausgeführt, tritt diese Aktivität vor, gleichzeitig oder nach den anderen nachfolgenden Aktivitäten in den Zustand *executing* ein: $(\{<, =, >\}, a^{pred_i}_{-executing}, a^{pred_j}_{-executing})$
7. Wird die Aktivität a^{pred_i} exklusiv mit den anderen nachfolgenden Aktivitäten von a ausgeführt, ist der Startzeitpunkt in keiner linearen Relation zu den Startzeitpunkten des Zustandes *executing* der nachfolgenden Aktivitäten: $(\{\|\}, a^{pred_i}_{-executing}, a^{pred_j}_{-executing})$

4.2.2 Scope mit Aktivität

Die Repräsentation eines Scopes besteht aus dem temporalen Modell aus Abbildung 4.4 und der temporalen Verknüpfung mit dem temporalen Modell der Kindaktivität, beschrieben in Tabelle 4.2. Dazu gelten folgende temporalen Abhängigkeiten zwischen dem Scope und der Kindaktivität:

- Der Scope und seine Kindaktivität transferieren gleichzeitig in den Zustand *init*.
- Transferiert der Scope in den Zustand *aborted*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *aborted*.
- Transferiert der Scope in den Zustand *executing*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *executing*.
- Transferiert der Scope in den Zustand *dead*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *dead*.
- Transferiert die Kindaktivität in den Zustand *fault*, transferiert der Scope zum gleichen Zeitpunkt in den Zustand *fault handling*.
- Transferiert der Scope in den Zustand *terminating*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *terminating*.
- Transferiert der Scope in den Zustand *terminated*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *terminated*.
- Transferiert die Kindaktivität in den Zustand *fault caught* oder den Zustand *completed*, transferiert der Scope zum gleichen Zeitpunkt in den Zustand *completed*.

		activity									
		init	aborted	executing	dead	fault handling	fault	fault caught	terminating	terminated	completed
scope	init	{=}									
	aborted		{=}								
	executing			{=}							
	dead				{=}						
	fault handler						{=}				
	fault										
	fault caught										
	terminating								{=, ||}		
	terminated									{=, ||}	
	completed							{=, ||}			{=, ||}

Tabelle 4.2: Temporale Verknüpfung von Scopes mit der Kind-Aktivität

4.2.3 Scope mit Fault Handler

Fault Handler eines Scopes werden durch das temporale Aktivitätenzustandsnetzwerk aus Abbildung 4.1 modelliert. Die Tabelle 4.3 beschreibt die temporale Verknüpfung der Startzeitpunkte der Zustände des Scopes, mit den Startzeitpunkten der Zustände des Fault Handlers. Dazu gelten folgende temporalen Abhängigkeiten zwischen dem Scope und dem Fault Handler:

- Der Scope und ein Fault Handler transferieren gleichzeitig in den Zustand *init*.
- Transferiert der Scope in den Zustand *aborted*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *aborted*.
- Transferiert der Scope in den Zustand *fault handling*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *executing*.
- Transferiert der Scope in den Zustand *dead*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *dead*.
- Transferiert der Fault Handler in den Zustand *fault*, transferiert der Scope zum gleichen Zeitpunkt in den Zustand *fault*.
- Transferiert der Fault Handler in den Zustand *completed*, transferiert der Scope zum gleichen Zeitpunkt in den Zustand *fault caught*.
- Transferiert der Scope in den Zustand *terminating*, transferiert der Fault Handler in den Zustand *aborted*, oder *terminating* (falls der Scope sich bereits in Ausführung befindet).
- Transferiert der Scope in den Zustand *completed*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *dead*, da der Fault Handler nicht mehr ausgeführt werden kann.

		fault handler									
		init	aborted	exe-cuting	dead	fault handling	fault	fault caught	termi-nating	termi-nated	comp-leted
scope	init	{=}									
	aborted		{=}								
	executing										
	dead				{=}						
	fault handler			{=}							
	fault						{=, ‖}				
	fault caught										{=, ‖}
	terminating				{=, ‖}				{=, ‖}		
	terminated										
	completed				{=, ‖}						

Tabelle 4.3: Temporale Verknüpfung von Scopes mit dem Fault Handlern

4.2.4 Fault Handler mit Aktivität

Die Repräsentation eines Fault Handlers besteht aus dem temporalen Modell aus Abbildung 4.1 und der temporalen Verknüpfung der Startzeitpunkte der Aktivitätenzustände der Kindaktivität, beschrieben in Tabelle 4.4. Dazu gelten folgende temporalen Abhängigkeiten zwischen dem Fault Handler und der Kindaktivität:

- Der Fault Handler und seine Kindaktivität transferieren gleichzeitig in den Zustand *init*.
- Transferiert der Fault Handler in den Zustand *aborted*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *aborted*.
- Transferiert der Fault Handler in den Zustand *executing*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *executing*.
- Transferiert der Fault Handler in den Zustand *dead*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *dead*.
- Transferiert die Kindaktivität in den Zustand *fault*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *fault*.
- Transferiert der Fault Handler in den Zustand *terminating*, transferiert die Kindaktivität zum gleichen Zeitpunkt in den Zustand *terminating*.
- Transferiert die Kindaktivität in den Zustand *fault caught* oder in den Zustand *completed*, transferiert der Fault Handler zum gleichen Zeitpunkt in den Zustand *completed*.

		activity									
		init	aborted	executing	dead	fault handling	fault	fault caught	terminating	terminated	completed
fault handler	init	{=}									
	aborted		{=}								
	executing			{=}							
	dead				{=}						
	fault						{=, ||}				
	terminating								{=, ||}		
	terminated										
	completed							{=, ||}			{=, ||}

Tabelle 4.4: Temporale Verknüpfung von Fault Handlern mit der Kind-Aktivität

5 Umsetzung

In den Kapiteln 3 und 4 wurde das Konzept und die temporalen Modelle für eine Repräsentation eines BPEL Prozesses in Punkt Algebra beschrieben. Dieses Kapitel beschreibt eine Framework-Implementierung des Konzepts von temporalen Aktivitätenzustandsnetzwerken für BPEL Prozesse. Zunächst werden Anforderungen an eine Implementierung formalisiert. Eine Beschreibung des Designs der Implementierung erläutert die Module des Frameworks. Anschließend wird das generische Datenmodell, das Punkt Algebra Datenmodell, sowie die Implementierung der Transformation von BPEL2Punkt Algebra und der Vergleich von zwei temporalen Aktivitätenzustandsnetzwerken erläutert.

Mögliche Schnittstellen zur Erweiterung werden vorgestellt und der Einsatz der Implementierung als Konsolenanwendung und Java-Klassenimport beschrieben.

Der Source Code der Implementierung die im Rahmen dieser Arbeit entstanden wurde auf GitHub veröffentlicht[1].

5.1 Anforderungen an die Implementierung

Neben den konzeptuellen Anforderungen aus Abschnitt 3.2 werden weitere Anforderungen an eine Implementierung gestellt:

- Implementierung in Java unter Verwendung eines bestehenden EMF Meta-Modells für BPEL[2]
- Performantes Datenmodell mit schnellen Operationen auf dem Graphen
- Schnittstellen zur einfachen Erweiterung der Transformation um weitere Aktivitäten
- Erstellung der Constraints im Punkt Algebra Netzwerk ohne Abfrage von Aktivitäten-spezfiefischen Sonderfällen im Transformations-Algorithmus.
- Einsatz als Konsolenanwendung und Java-Klassenimport
- Robuste Identifizierung der Zeitpunkte durch Referenzen zu Instanzen von Aktivitäten und Enums.

[1] GitHub Veröffentlichung der Implementierung: https://github.com/wagnerse/bpel-equivalence
[2] EMF Meta-Modell für BPEL https://eclipse.org/bpel/

5.2 Design der Implementierung

Die Implementierung des Konzepts zur Repräsentation von BPEL Prozessen durch temporale Aktivitätenzustandsnetzwerke teilt sich in verschiedene Module auf (siehe Abb. 5.1). Grundlage bildet ein Modul, das ein generisches Datenmodell für verschiedene temporale Algebren zur Durchführung von CSP Algorithmen bietet. Das zweite Modul nutzt dieses generische Modul zur Realisierung eines Punkt Algebra Datenmodells mit der Funktionalität zur Berechnung der Hülle. Das dritte Modul enthält die Logik zur Transformation von BPEL Prozessen in ein Punkt Algebra Netzwerk. Das vierte Modul vergleicht zwei Punkt Algebra Netzwerke hinsichtlich der Äquivalenz der temporalen Aktivitätenzustände. Das EMF BPEL Datenmodell wurde vom BPEL Eclipse Projekt übernommen.

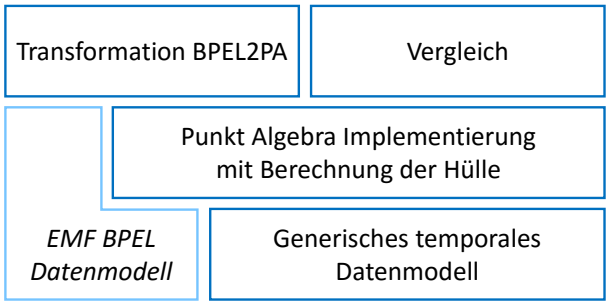


Abbildung 5.1: Die Implementierung gliedert sich in die Module Generische CSP Implementierung, Punkt Algebra CSP Implementierung, Transformation und Vergleich.

5.3 Generisches Datenmodell für temporale Algebren

Grundlage der Implementierung ist eine generisches Datenmodell für temporale Algebren. Dieser Abschnitt erläutert das Datenmodell anhand des Klassendiagramms in Abbildung 5.2. Das Datenmodell enthält eine abstrakte Klasse `CSPNetwork`, die Variablen und Constraints eines temporalen Netzwerkes enthält.

Die Trennung der Implementierung des temporalen Netzwerkes von der Punkt Algebra Implementierung separiert die Umsetzung der Graph-Implementierung von der spezifischen Implementierung der Punkt Algebra.

Dieses Vorgehen bietet eine große Flexibilität hinsichtlich der zukünftigen Wartbarkeit dieser Implementierung. Es ist zum einen möglich auch eine temporale Repräsentation von BPEL Prozessen in Intervall Algebra zu realisieren, die von dem generischen Datenmodell für temporale Algebren erbt. Ebenso ist es möglich das generische Datenmodell für temporale Algebren gegen

eine Open Source Implementierung auszutauschen. Auf den Einsatz des *Meta-CSP Frameworks*[3] wurde zugunsten einer besseren Performance verzichtet.

Die Klasse `CSPNetwork` enthält eine Liste aller Variablen des temporalen Netzwerkes. Die Methode `addVariable` zum Hinzufügen neuer Variablen ist nur für die abgeleiteten Klassen sichtbar. Diese reduzierte Sichtbarkeit stellt sicher, dass nur über die abgeleiteten Klassen und deren spezifisches Verhalten zum Handling von Variablen neue Variablen erzeugt und hinzugefügt werden können. Im Beispiel der Punkt Algebra Implementierung werden Variablen von der Klasse `PANetwork` erzeugt und gleichzeitig hinzugefügt. So ist sichergestellt, dass alle Variablen auch Teil des temporalen Netzwerkes sind. Neben Methoden zum Hinzufügen und Löschen von Constraints enthält die `CSPNetwork` Klasse die Methode `reduceConstraint` mit dieser Methode werden die Relationen eines Constraints eingeschränkt. Ein Constraint, dass alle Relationen einer Algebra enthält, die sogenannte T-Relation kann mit der abstrakten Methode `newTConstraint` erstellt werden.

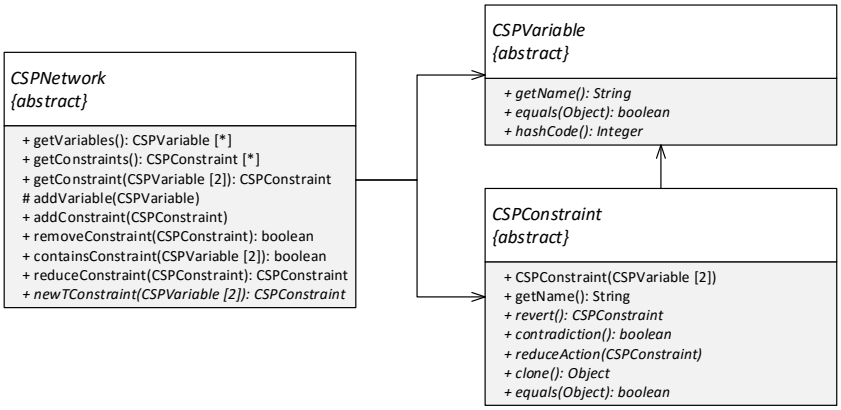


Abbildung 5.2: Das generische Datenmodell für temporale Algebren besteht aus der Klasse `CSPNetwork`, die einen Graph aus Variablen und Constraints enthält. Abstrakte Methoden in der Constraint-Klasse definieren Schnittstellen für Zustände und Verhalten einer Algebra.

Die Klasse `CSPVariable` stellt einen Knoten im temporalen Netzwerk dar. Die Klasse enthält abstrakte Methoden zur Berechnung der Äquivalenz sowie von Hash-Werten um die Klasse als Schlüssel in einer HashMap verwenden zu können.

Die Klasse `CSPConstraint` beschreibt temporale Relationen zwischen zwei Instanzen der Klasse `CSPVariable`. Diese Klasse enthält ebenfalls abstrakte Methoden die eine Schnittstelle für das Verhalten und den Zustand des Constraints bilden. Die Methode `contradiction` gibt

[3]The Meta-CSP Framework: `http://federicopecora.github.io/meta-csp-framework/`

einen boolschen Zustand zurück, ob das Constraint einen Widerspruch darstellt. Da eine Instanz der Klasse `CSPConstraint` eine gerichtete Kante darstellt, kann über die abstrakte Methode `revert` eine Kopie des Constraints in invertierter Richtung erzeugt werden. CSP-Algorithmen basieren darauf, dass zur Berechnung der Hülle Constraints weiter eingeschränkt werden. Die abstrakte Methode `reduceAction` schränkt ein Constraint basierend auf einem Constraint mit gleichen Start- und Ziel-Knoten ein.

5.4 Datenmodell Punkt Algebra

Das generische Datenmodell für temporale Algebren das im vorherigem Abschnitt beschrieben wurde, bildet die Grundlage für das Punkt Algebra Datenmodell, das von den abstrakten Klassen `CSPNetwork`, `CSPVariable` und `CSPConstraint` ableitet. Das Datenmodell für Punkt Algebra Netzwerke ist in Abbildung 5.3 modelliert.

Ein Punkt Algebra Netzwerk wird von der Klasse `PANetwork` verwaltet und enthält einen Solver zur Berechnung der Hülle, der über die Methode `probagate` gestartet werden kann. Weitere Aufgabe der Klasse `PANetwork` ist die Erstellung von neuen Zeitpunkten der Klasse `PAVariable`. Wird ein neuer Zeitpunkt erstellt, wird dieser dem Punkt Algebra Netzwerk hinzugefügt und ein Constraint mit der T-Relation zwischen dem neuen Zeitpunkt allen anderen Zeitpunkten des Punkt Algebra Netzwerkes erstellt.

Eine Instanz der Klasse `PAVariable` stellt einen Zeitpunkt dar. Ein Zeitpunkt in einem temporalem Aktivitätenzustandsnetzwerk wird durch eine Aktivität, einem Zustand der Aktivität und der Information ob der Zeitpunkt ein Start- oder Endzeitpunkt ist identifiziert. In dieser Arbeit werden allerdings nur die Startzeitpunkte von Zuständen modelliert. Die Informationen zur Identifizierung einer Instanz der Klasse `PAVariable` sind in ein Datenobjekt `TimePointDesc` gekapselt. Zur Sicherstellung der eindeutigen Bezeichnung von `PAVariable`-Instanzen sind alle Informationen zur Identifizierung in Objekten oder Enums beschrieben.

Ein Punkt Algebra Constraint ist durch die Klasse `PAConstraint` implementiert. Der abstrakten Implementierung fügt diese Klasse noch die Relationen der Punkt Algebra mit Branching Time hinzu. Das Constraint enthält eine Liste von `RelationEnum` und die im Kap. 2.3.1 beschriebenen Operationen von Constraints. Das Enum `RelationEnum` implementiert die Operationen der Punkt Algebra auf Relationen.

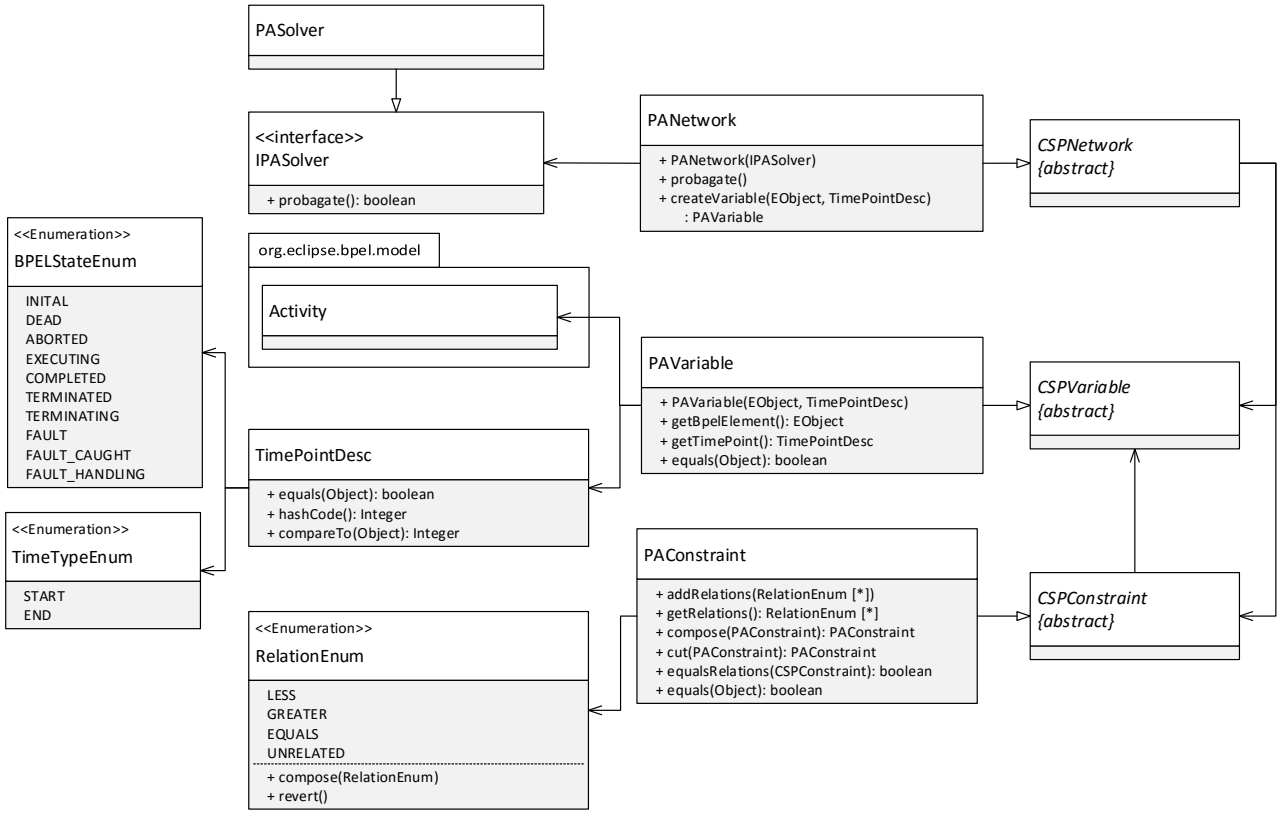


Abbildung 5.3: Die Implementierung der Punkt Algebra erbt von dem generischen Datenmodell für temporale Algebren. Eine Variable wird identifiziert durch eine BPEL Aktivität, einem Aktivitätenzustand und der Information über Start- oder Endzeitpunkt. Das Punkt Algebra Netzwerk enthält einen Solver zur Berechnung der Hülle des Punkt Algebra Netzwerkes.

5.4.1 Berechnung der Hülle eines Punkt Algebra Netzwerkes

Nachdem ein Constraint in einem Punkt Algebra Netzwerk geändert wurde, wird durch die Berechnung der Hülle die Auswirkungen auf andere Constraints berechnet. In der Implementierung wird der Algorithmus von Allen [All83] zur Berechnung der Hülle eingesetzt, der in Kap. 2.5 beschrieben ist.

Die Berechnung der Hülle eines Punkt Algebra Netzwerkes ist in eine Implementierung des Interface `IPASolver` gekapselt, um flexibel weitere Implementierungen zur Berechnung der Hülle integrieren zu können. Eine Instanz der Klasse `PANetwork` wird mit der Instanz einer Implementierung des Interfaces `IPASolver` initialisiert.

Die Klasse `PASolver` implementiert den Algorithmus von Allen [All83]. Der Solver wird nicht nach jedem Hinzufügen eines Constraints ausgeführt, sondern erst nachdem für alle BPEL Aktivitäten das temporale Aktivitätenzustandsnetzwerk aufgebaut wurde. Dazu fügt der Solver vor der Ausführung alle Variablen der Warteschlange hinzu. Wird während der Ausführung des Solvers ein Widerspruch erkannt, wird die Ausführung des Solvers mit einer `IllegalStateException` beendet.

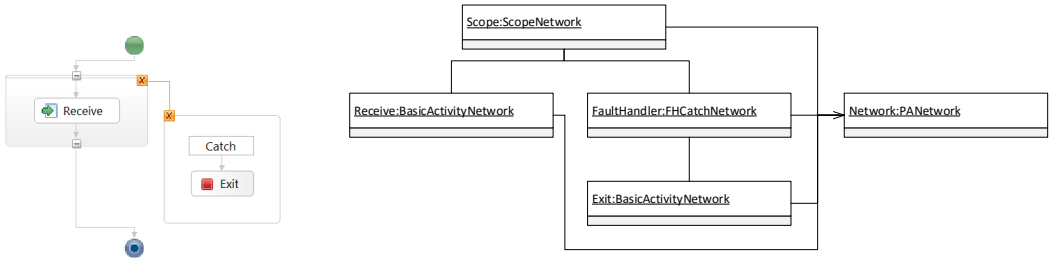
5.5 Transformation BPEL2Punkt Algebra

Die Transformation eines BPEL Prozesses in ein temporalen Aktivitätenzustandsnetzwerk bildet die Grundlage für die Implementierung des Vergleiches von zwei BPEL Prozessen. Die Implementierung basiert auf dem Punkt Algebra Datenmodell. Die Implementierung setzt den Algorithmus aus Kap. 3.5 um. Dieser Algorithmus beschreibt eine rekursive Tiefensuche über die Aktivitäten des BPEL Prozesses.

Da diese Arbeit nicht alle BPEL Aktivitäten und keine Choreographien umfasst, ist für dieses Modul ein Klassen-Design nötig, dass eine flexible und einfache Erweiterung ermöglicht. Diese Erweiterungsfähigkeit wird über eine abstrakte Basisklasse für alle Transformationen realisiert.

Die rekursive Tiefensuche über die einzelnen Aktivitäten des BPEL Prozesses ist in einer abstrakten Klasse `AbstractActivityNetwork` realisiert. Für jedes Aktivität existiert eine Implementierung dieser abstrakten Klasse, die eine Beschreibung der einzelnen Startzeitpunkte, der Intra-Activty-Constraints und der Inter-Aktivity-Constraints für diesen Aktivitäten-Typ enthält. Zur Laufzeit der Transformation wird für jede Aktivität eine passende Instanz einer Implementierung der Klasse `AbstractActivityNetwork` erstellt, die die Transformation für diese Aktivität ausführt.

Abbildung 5.4 zeigt einen Beispiel-Prozess dieser besteht aus einem Scope mit einer Aktivität und einem Fault Handler, der ebenfalls eine Aktivität enthält. Das Objektmodell modelliert die Netzwerk-Objekte, die zur Laufzeit der Transformation erstellt werden. Für jede Aktivität wird ein Netzwerk-Objekt erzeugt, dessen Klasse von der Klasse `AbstractActivityNetwork` erbt. All diese Netzwerk-Objekte enthalten eine Referenz auf das gleiche Punkt Algebra Netzwerk Objekt der Klasse `PANetwork`. Zur Laufzeit erzeugen die Netzwerk-Objekte der Reihe nach in dem Punkt Algebra Netzwerk für ihre Aktivität die Variablen mit Intra-Activity-Constraints und verknüpfen diese Variablen durch Inter-Activity-Constraints mit den Variablen des Netzwerk-Objektes dessen Aktivität im BPEL Prozess hierarchisch übergeordnet ist. In diesem Beispiel werden z. B. die Variablen des Objektes *Exit* mit den Variablen des Objektes *FaultHandler* durch Inter-Aktivity-Constraints verknüpft.



(a) BPEL Prozess **(b)** Objektmodell

Abbildung 5.4: Gegenüberstellung eines BPEL Prozesses und den Netzwerk-Objekten, die die Transformation der einzelnen Aktivitäten in ein temporales Aktivitätenzustandsnetzwerk durchführen. Alle Klassen der Objekte erben von der Klasse `AbstractActivityNetwork`.

Der nun folgende Abschnitt beschreibt die abstrakte Klasse `AbstractActivityNetwork`, die die Basis-Funktionalität der Transformation enthält und den Transformationsalgorithmus aus Kap. 3.5 implementiert.

5.5.1 Klassen-Design einer Transformation

Die Transformation einer Aktivität eines BPEL Prozesses ist in jeweils einer Klasse pro Aktivitätentyp definiert. Basis dieser Transformationsklassen ist die abstrakte Klasse `AbstractActivityNetwork` (siehe Abbildung 5.5), die in diesem Abschnitt beschrieben wird.

Der Konstruktor der Klasse `AbstractActivityNetwork` erwartet ein Objekt der Klasse `PANetwork` und das Netzwerk-Objekt dessen Aktivität im BPEL Prozess hierarchisch übergeordnet ist.

Der Ablauf der Transformation einer Aktivität ist in der Methode `linkActivityNetworkLayer` der Klasse `AbstractActivityNetwork` implementiert. Für jeden Zustand der Aktivität wird in dem Punkt Algebra Netzwerk eine Instanz der Klasse `PAVariable` als Startzeitpunkt festgelegt. Die abstrakte Methode `getEObject` hat die Instanz dieser Aktivität als Rückgabewert. Zur Validierung enthält die abstrakte Methode `getSupportedEClass` eine Referenz auf eine Meta-Klasse des EMF Meta-Modells von BPEL. Die Meta-Klassen des Meta-Modells können z. B. `Scope`, `Flow`, `Exit` usw. sein. Die Startzeitpunkte werden zusammen mit den Intra-Activity-Constraints erstellt (siehe 4.1 für die Definition der Intra-Activity-Constraints) und werden über die abstrakte Methode `getLocalConstraints` ermittelt.

Die Klasse `AbstractActivityNetwork` enthält eine Hash Map, die alle möglichen Relationen zwischen den Startzeitpunkten der Zustände der eigenen Aktivität und den Startzeitpunkten der Zustände der Kindaktivität enthält. Die Hash Map wird individuell durch den Konstruktor der der Klasse gefüllt, die die Klasse `AbstractActivityNetwork` implementiert. Dazu kann

die Methode `putConstraint` genutzt werden. Sind Relationen zwischen den Startzeitpunkten der Zustände der Kindaktivitäten definiert, sind diese ebenfalls Teil dieser Hash Map. Im Beispiel der Transformation eines Flows, enthält die Hash Map die Relationen zwischen den Startzeitpunkten der Zustände des Flows und seiner Kindaktivitäten, sowie weitere Relationen zwischen Startzeitpunkten der Zustände von Aktivitäten, die durch Links verbunden sind.

Über die Methode `getActivityConnector` und das Interface `IActivityConnector` sind die Variablen der Netzwerk-Objekte der Kindaktivitäten verfügbar. Netzwerk-Objekte der Kindaktivitäten werden erstellt, bevor nun die Inter-Activity-Constraints erstellt werden.

Zur Erstellung der Inter-Activity-Constraints wird über die Variablen der Aktivität und der Kindaktivität iteriert. Sind für solch ein Variablen-Paar Relationen in der Hash Map hinterlegt, wird ein Constraint mit diesen Relationen zwischen den Variablen erstellt. Anschließend wird auch über die Variablen der Kindaktivitäten untereinander iteriert um auch die Constraints zwischen den Variablen der Kindaktivitäten zur erstellen, falls für diese Variablen Relationen in der Hash Map hinterlegt sind.

Dieses Vorgehen der allgemeinen Definition von Relationen in einer Hash Map und der anschließenden konkreten Anwendung auf Aktivitäten von denen nur einen Übermenge an Zuständen bekannt ist ermöglicht eine generische Implementierung ohne explizite Abfrage von Sonderfällen und ermöglicht eine schlanke flexible Implementierung.

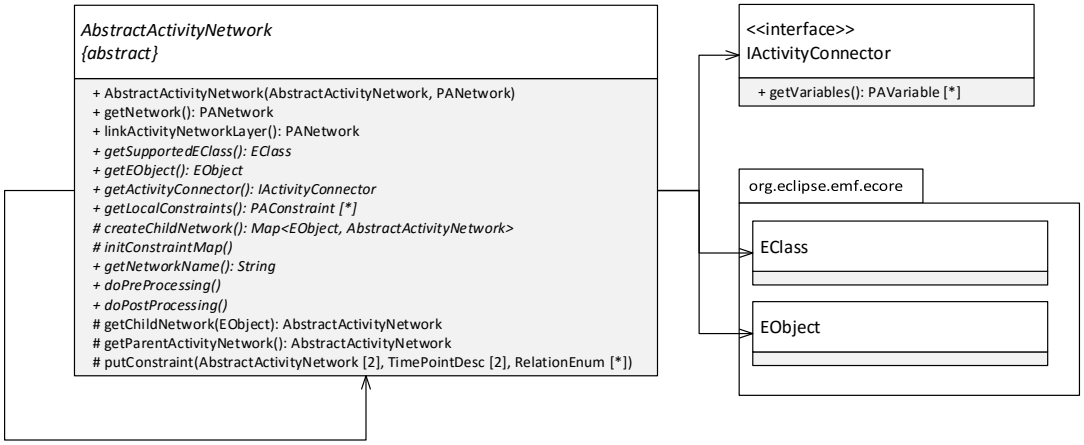


Abbildung 5.5: Die abstrakte Klasse `AbstractActivityNetwork` ist Basisklasse für jede Transformationsimplementierung einer Aktivität. Eine Instanz enthält zur Identifizierung des Objektes und zur Validierung eine Referenz zu einer Instanz einer Aktivität und dem zugehörigem Meta-Objektes im ecore Meta-Modell.

Die abstrakten Methoden `doPreProcessing` und `doPostProcessing` werden vor bzw. nach der Erstellung der Inter-Activity-Constraints ausgeführt und können von einer Implemen-

tierung der Klasse `AbstractActivityNetwork` verwendet werden um an diesen Stellen des Transformationsablaufes einer Aktivität spezifisches Verhalten zu implementieren, ohne die Basisklasse zu verändern.

Nach der Erstellung aller Constraints wird für alle Netzwerk-Objekte der Kindaktivitäten der nächste Rekursionsschritt durch den Aufruf der Methode `linkActivityNetworkLayer` auf den jeweiligen Objekten durchgeführt.

5.5.2 Erweiterbarkeit durch ein Factory-Repository

Im vorangegangenen Abschnitt wurde die abstrakte Basisklasse `AbstractActivityNetwork` beschrieben. Für jeden unterstützten Aktivitätentyp existiert somit eine Ableitung dieser Basisklasse. Diese Ableitungen werden in in der Basisklasse `AbstractActivityNetwork` instanziiert. Zur Kapselung dieser Instanziierung wurde das Factory-Pattern angewendet und die Ausführung des Factory Pattern nochmals in ein Factory-Repository ausgelagert, wie Abbildung 5.6 dargestellt. Für jede Implementierung der Klasse `AbstractActivityNetwork` existiert eine Ableitung des Interface `IActivityNetworkFactory`, die die Instanziierung der Netzwerk-Objekte übernimmt.

Um eine Ableitung der Klasse `AbstractActivityNetwork` in einer Ausführung der Transformation zur Anwendung zur bringen, genügt es eine Instanz der zugehörigen Factory-Klasse im Factory-Repository zu registrieren. Wird dem Factory-Repository mit der Methode `createElementNetwork` eine Instanz einer Aktivität als `EObject` übergeben, wird aus dem Factory-Repository anhand des `EClass` Objektes die passende Factory-Instanz ausgewählt und ein Netzwerk-Objekt erstellt.

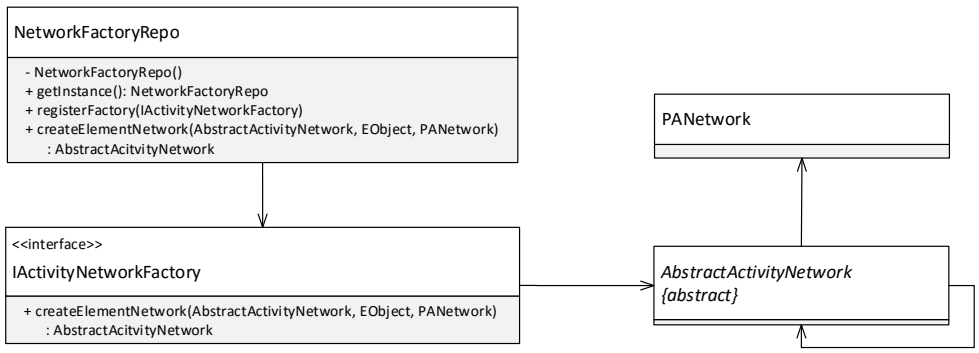


Abbildung 5.6: Im `NetworkFactoryRepo` werden Factory-Klassen registriert, die für eine Aktivität die passende Netzwerk-Klasse zur Transformation der Aktivität in ein temporales Aktivitätenzustandsnetzwerk erstellen.

5.5.3 Implementierung einer Transformation

Im den vorangegangenen Abschnitten wurde die abstrakten Klasse `AbstractActivityNetwork` und das Factory-Repository beschrieben. Dieser Abschnitt beschreibt die Implementierung und Integration einer weiteren Transformation einer Aktivität. Die Implementierung einer Transformationsklasse gliedert sich in drei Schritte.

1. Ableitung der Klasse `AbstractActivityNetwork`
2. Implementierung der Klasse `IActivityNetworkFactory`
3. Registrierung einer Instanz der Factory im Factory-Repository

Die Ableitung der Klasse `AbstractActivityNetwork` umfasst die Implementierung der abstrakten Methoden, deren Anforderungen in der Tabelle 5.1 beschrieben sind.

Methode	**Anforderung**
`getNetworkName`	Rückgabe eines String zur Benennung des Objekts. Diese Benennung muss nicht eindeutig sein. z. B. Name der Aktivität.
`getSupportedEClass`	Rückgabe des Meta-Objektes der zu transformierenden Aktivität. Das Meta-Objekt ist Teil des BPEL Meta-Modells.
`getEObject`	Rückgabe der Objekt-Instanz der konkret zur transformierenden Aktivität. Die Objekt-Instanz der Aktivität wird in der Regel dem Konstruktor der zu implementierenden Klasse übergeben.
`getLocalConstraints`	Rückgabe eines Arrays mit den Constraints zwischen den Variablen, die die Startzeitpunkte der Zustände der zu transformierenden Aktivität darstellen.
`getActivityConnector`	Rückgabe einer Instanz einer Implementierung der Klasse `IActivityConnector`. Dieses Data-Transfer-Object enthält die Variablen, die die Startzeitpunkte der Zustände der zu transformierenden Aktivität darstellen.
`initConstraintMap`	Rückgabe einer Hash Map, bestehend aus Relationen zwischen den Variablen unterschiedlicher Netzwerk-Objekte. Aus diesen Relationen werden zur Laufzeit der Transformation die Inter-Activity-Constraints gebildet.
`createChildNetworks`	Rückgabe einer Hash Map, die die Netzwerk-Objekte für alle Kindaktivitäten enthält. Dazu kann die Hilfsmethode `createChildNetwork` der Basisklasse genutzt werden.

Tabelle 5.1: Methoden zur Implementierung der Transformation einer Aktivität

Zur Initialisierung der neu implementierten Transformationsklasse wird eine Factory-Klasse erstellt. Die Factory-Klasse erstellt eine Instanz der neu implementierten Transformationsklasse. Die Tabelle 5.2 beschreibt die Anforderungen an die Implementierung des Interface `IActivityNetworkFactory`.

Methode	**Anforderung**
`getSupportedEClass`	Rückgabe des Meta-Objektes zur dem die Factory-Implementierung ein passendes Netzwerk-Objekt zurückgibt. Das Meta-Objekt ist Teil des ecore Meta-Modells.
`createElementNetwork`	Rückgabe eines Netzwerk-Objekts zur Transformation.

Tabelle 5.2: Methoden zur Implementierung des Interface `IActivityNetworkFactory`

Nachdem ein passende Factory-Klasse implementiert wurde, kann diese dem Factory-Repsitory hinzugefügt werden. Die Klasse `NetworkFactoryRepo` wurde nach dem Singelton-Pattern implementiert und ist somit statisch referenzierbar. Mit der Methode `registerFactory` kann eine Instanz einer Factory dem Repository hinzugefügt werden. Der Aufbau des Repositories ist im Konstruktor der Klasse `BpelEquivalence` realisiert, die gleichzeitig auch Schnittstelle zur Umsetzung des Konzepts zur Punkt Algebra basierten Repräsentation und dem Vergleich des Kontrollflusses von BPELProzessen ist.

5.6 Vergleich von temporalen Aktivitätenzustandsnetzwerken

Liegen von zwei BPEL Prozessen jeweils eine Repräsentation als temporales Aktivitätenzustandsnetzwerk vor, können diese beiden Punkt Algebra Netzwerke verglichen werden. Dazu wird überprüft, ob in beiden Punkt Algebra Netzwerken, gleiche Constraints vorliegen. Der Algorithmus und die Äquivalenzbedingungen des Vergleiches von zwei temporalen Aktivitätenzustandsnetzwerken ist in Kapitel 3.6 beschrieben. Um die Aktivitäten in den beiden BPEL Prozessen zuordnen zu können, müssen gleiche Aktivitäten den gleichen Namen haben.

Die Klasse `ProcessEquals`, die in Abbildung 5.7 modelliert ist, enthält eine Methode zum Vergleich. Der Vergleich muss über die Klasse `EqualsConfiguration` parametrisiert werden, die folgende Vergleichsparameter enthält:

- Liste mit den Namen der Aktivitäten, die verglichen werden sollen
- Zustandsmenge die verglichen werden soll

Diese Möglichkeit der Konfiguration bietet den Vorteil den Vergleichsraum an Aktivitäten einzuschränken und bereits im Vorfeld False-Positive-Fehler zu reduzieren. Die Einschränkung der Zustandsmenge dient dazu spezielle Use-Cases zu validieren. Ebenso ist es möglich den Zustandsraum der verglichen wird einzuschränken. Wird z. B. der Zustand *executing* in die Konfiguration mit aufgenommen, werden die Constraints zwischen den Startzeitpunkten der *executing* Zustände der Aktivitäten verglichen. Diese Beispiel-Konfiguration entspricht dem Vergleich der Ausführungsreihenfolge der Aktivitäten.

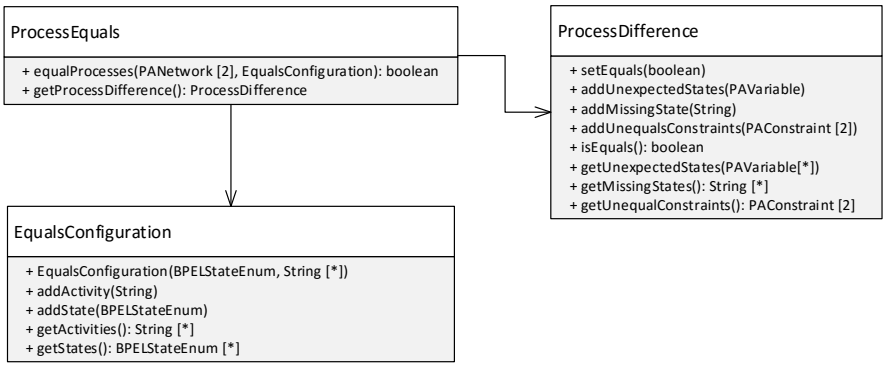


Abbildung 5.7: Die Klasse `ProcessEquals` vergleicht anhand der Konfiguration, die in dem Data-Transfer-Object `EqualsConfiguration` beschrieben ist, zwei temporale Aktivitätenzustandsnetzwerke. Das Ergebnis liegt in einer Instanz der Klasse `ProcessDifference` vor.

Die Klasse `ProcessDifference` hält ein detailliertes Ergebniss des Vergleichs als Data-Transfer-Objekt vor. Eine Instanz der Klasse `ProcessDifference` beschreibt den Unterschied von zwei BPEL Prozessen anhand der Kriterien die in der Tabelle 5.3 beschrieben sind.

`istEquals`	Gleichheit der Prozesse
`getUnexpectedSTates`	Aktivitätenzustände, die Teil der Konfiguration sind, aber in einem Prozess fehlen.
`getMissingStates`	Aktivitätenzustände, die Teil der Konfiguration sind, aber in beiden Prozessen fehlen.
`getUnequalsConstraints`	Constraints die in den beiden Punkt Algebra Netzwerken unterschiedlich sind.

Tabelle 5.3: Repräsentation des Ergebnis eines Vergleiches

5.7 Einsatz der Implementierung

Die Implementierung die im Rahmen dieser Arbeit entstanden ist, kann als Konsolenanwendung und als Java-Klassenimport genutzt werden. Dieser Abschnitt beschreibt den Einsatz der Konsolenanwendung, eine Klasse als Import für neue Projekte und die Markierung von exklusiven und parallelen Kontrollflüssen in BPEL Prozessen.

5.7.1 Konsolenanwndung

Sollen zwei BPEL Prozesse mit der Konsolenanwendung verglichen werden, müssen dem Aufruf Parameter übergeben werden. Folgende Parameter müssen der Konsolenanwendung übergeben werden:

```
<path process 1> <path process 2> [-s]{<state>} [-a]{<activity name>}
```

Listing 5.1: Parameter der Konsolenanwendung

Die Konsolenanwendung erwartet die Pfade der beiden Prozesse, optional eine Liste von Zuständen, die Grundlage des Vergleichs sind und eine Liste der Aktivitäten, die in beiden Prozessen verfügbar sein sollen. Werden keine Zustände angegeben, wird die Konfiguration um den Zustand *executing* ergänzt.

5.7.2 Klassenimport

Zum Einsatz dieser Implementierung in weiteren Java-Projekten kann ein Import der Klasse `BpelEquivalence` durchgeführt werden, die in Abbildung 5.8 dargestellt ist. Diese Klasse enthält die Methode `createNetwork` zur Transformation eines BPEL Prozesses in ein temporales Aktivitätenzustandsnetzwerk. Der überladene Parameter von Typ boolean gibt an, ob der Rückgabewert auch im Fall von Widersprüchen im Punkt Algebra Netzwerk dieses Netzwerk zurück gibt. Alternativ wird die Transformation mit einer `IllegalStateException` beendet. Der Vergleich von zwei temporalen Aktivitätenzustandsnetzwerken als Punkt Algebra Netzwerk ist mit der Methode `checkBpelEquivalence` möglich. Zum einen kann der Kontrollfluss der *executing* Zustände überprüft werden, wenn eine Liste von Aktivitäten-Namen übergeben wird. Eine Überladung der Methode ermöglicht auch eine genauere Konfiguration des Vergleichs durch ein Objekt der Klasse `EqualsConfiguration`.

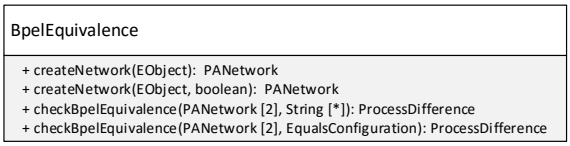


Abbildung 5.8: Die Klasse `BpelEquivalence` enthält eine Programmierschnittstelle zur Transformation und dem Vergleich von BPEL Prozessen

5.7.3 Markierung von Kontrollflüssen in BPEL Prozessen

Die Analyse der Kontrollflüsse ist kein Teil dieser Arbeit. Daher ist es nötig bei Aktivitäten, die mehrere ausgehende Links enthalten, den Typ der Kontrollflussverzweigung zu markieren. Diese Marker sind über BPEL Extensions realisiert. Dazu wird zunächst der Namespace der Extension und die Extension in der BPEL Datei definiert. In dem Sources-Element einer Aktivität wird mit einem `Equivalenceanotation`-Element angegeben, ob die ausgehenden Kanten *exclusive* oder *parallel* sind. Listing 5.2 zeigt die Definition des Namespace und der Extension, sowie eine Beispielverwendung der Extension in einer Aktivität.

```
xmlns:ext="http://iaas.uni-stuttgart.de/bpel/equivalence/extension/exclusive"

<extensions>
      <extension
          namespace="http://iaas.uni-stuttgart.de/bpel/equivalence/extension/exclusive"
          mustUnderstand="no" />
</extensions>

[...]

<bpel:empty name="start">
      <bpel:sources>
            <ext:equivalenceanotation>exclusive</ext:equivalenceanotation>
            <bpel:source linkName="link1"></bpel:source>
            <bpel:source linkName="link3"></bpel:source>
      </bpel:sources>
</bpel:empty>
```

Listing 5.2: Markierung von exklusiven und parallelen Kontrollflüssen

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Repräsentation des Kontrollflusses von BPEL Prozessmodellen durch temporale Aktivitätennetzwerke realisiert. Grundlage sind Arbeiten über die Zustände von BPEL Aktivitäten, Punkt Algebra, Branching Time und der Algorithmus von Allen zur Berechnung der Hülle von temporalen Netzwerken.

Aus dem Zustandsmodell für BPEL Aktivitäten wurde ein temporales Aktivitätenzustandsnetzwerk in Punkt Algebra für verschiedene Aktivitäten abgeleitet. Zwischen den Startzeitpunkten der Aktivitätenzustände von strukturierenden Aktivitäten und deren Kindaktivitäten wurden Relationen definiert, um den Kontrollfluss, basierend auf der BPEL Spezifikation, für einen BPEL Prozess in ein Punkt Algebra Netzwerk zu transformieren. Durch die Berechnung der Hülle eines Punkt Algebra Netzwerkes können die temporalen Relationen zwischen den Aktivitätenzuständen der Basis-Aktivitäten bestimmt werden. Somit steht ein temporales Modell zur Verfügung, das transparent von den Aktivitätenzuständen der strukturierenden Aktivitäten die temporale Relation der Zustände der Basis-Aktivitäten beschreibt. Für solch ein Punkt Algebra Netzwerk, das ein temporales Aktivitätenzustandsnetzwerk eines BPEL Prozesses enthält, wurden Äquivalenzbedingungen definiert, um Traces von zwei Prozessen vergleichen zu können. Die nötigen Algorithmen zur Transformation und dem Vergleich wurden beschrieben.

Die im Rahmen dieser Arbeit entstandene Implementierung ermöglicht den Vergleich von Traces von zwei BPEL Prozessen ohne Prozess-Instanzen zur Ausführung zu bringen. Durch den Fokus des Implementierungsentwurfes auf mögliche Erweiterungen ist eine schnelle Erweiterung der Transformationsfunktionalität möglich.

Ausblick

Die in dieser Arbeit entwickelte temporale Repräsentation von BPEL Modellen, sowie die Implementierung der Äquivalenzbedingungen bieten eine Grundlage für weitere Arbeiten im Bereich der Behavioural Profiles und der Prozess Konsolidierung. Eine weitere Anwendungsmöglichkeit der temporalen Repräsentation von BPEL Modellen ist der Vergleich von aufgezeichneten Traces mit den Traces, die aus dem temporalen Aktivitätenzustandsnetzwerk gewonnen werden. Durch einen Vergleich von Traces, können Traces ermittelt werden, die in der realen Ausführung nicht zur Anwendung kommen oder Sicherheits- und Compliance-Schwachpunkte des modellierten Prozesses darstellen können.

Diese Arbeit hat sich nicht mit der kompletten Menge an Aktivitäten von BPEL beschäftigt. Eine Entwicklung von temporalen Modellen für sequenzielle Aktivitäten, sowie ein Konzept zur Integration von BPEL Erweiterungen wie BPEL4People oder Choreographien, würden weitere Anwendungsmöglichkeiten bieten. Die Integration von Analysen von Kontrollflussbedingungen ermöglichen eine direkte Analyse von BPEL Modellen ohne manuelle Vorbereitung.

Die Implementierung kann um die Funktionalität erweitert werden, ein Mapping von korrespondierenden Aktivitäten zu definieren. Somit können Prozesse verglichen werden, deren Aktivitäten nicht die gleichen Bezeichner haben. Daraus würde sich auch die Möglichkeit ergeben Aktivitäten als korrespondierend zu bezeichnen, die nicht von dem gleichen Aktivitäten-Typ sind. Somit könnten Analysen durchgeführt werden, ob sich die Traces eines Prozesses durch die Änderung des Typs einer Aktivität verändern.

Literaturverzeichnis

[All83] J. F. Allen. „Maintaining Knowledge About Temporal Intervals“. In: *Commun. ACM* 26.11 (Nov. 1983), S. 832–843.

[Bro01] M. Broxvall. „The point algebra for branching time revisited“. In: *KI 2001: Advances in Artificial Intelligence*. Springer, 2001, S. 106–121.

[FBS04] X. Fu, T. Bultan und J. Su. „Analysis of interacting BPEL web services“. In: *Proceedings of the 13th international conference on World Wide Web*. ACM. 2004, S. 621–630.

[Fer04] A. Ferrara. „Web services: a process algebra approach“. In: *Proceedings of the 2nd international conference on Service oriented computing*. ACM. 2004, S. 242–251.

[FGV05] R. Farahbod, U. Glässer und M. Vajihollahi. „A Formal Semantics for the Business Process Execution Language for Web Services.“ In: *WSMDEIS*. 2005, S. 122–133.

[FR14] J. Freund und B. Rücker. *Praxishandbuch BPMN 2.0*. Carl Hanser Verlag GmbH Co KG, 2014.

[HSS05] S. Hinz, K. Schmidt und C. Stahl. „Transforming BPEL to Petri nets“. In: *Business Process Management*. Springer, 2005, S. 220–235.

[Ivo99] S. M. Ivo Düntsch Hui Wang. „Relations algebras in qualitative spatial reasoning“. In: *Fendamenta Informaticae*. 1999, S. 229–248.

[KEvL+11] O. Kopp, L. Engler, T. van Lessen, F. Leymann und J. Nitzsche. „Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus.“ In: *S-BPM ONE 2010 - the Subjectoriented BPM Conference (CCIS)*. Bd. 138. Communications in Computer and Information Science. Springer-Verlag, 2011.

[KHK+11] O. Kopp, S. Henke, D. Karastoyanova, R. Khalaf, F. Leymann, M. Sonntag, T. Steinmetz, T. Unger und B. Wetzstein. „An event model for WS-BPEL 2.0“. In: *Online Publikationen der Universität Stuttgart* (2011). URL: http://dx.doi.org/10.18419/opus-2951.

[LLN11] T. v. Lessen, D. Lübke und J. Nitzsche. *Geschäftsprozesse automatisieren mit BPEL*. 1. Aufl. Heidelberg: dpunkt-Verl., 2011, XIV, 259 S.

[Loh07] N. Lohmann. „A feature-complete Petri net semantics for WS-BPEL 2.0“. In: *Web Services and Formal Methods*. Springer, 2007, S. 77–91.

[Mac77] A. K. Mackworth. „Consistency in networks of relations“. In: *Artificial intelligence* 8.1 (1977), S. 99–118.

[OAS07] OASIS. „Web services business process execution language version 2.0“. In: *2007434 - I* (2007). URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[OVV+07] C. Ouyang, E. Verbeek, W. M. Van Der Aalst, S. Breutel, M. Dumas und A. H. Ter Hofstede. „Formal semantics and analysis of control flow in WS-BPEL“. In: *Science of Computer Programming* 67.2 (2007), S. 162–198.

[Rei94] A. J. Reich. „Intervals, Points, and Branching Time“. In: *TIME*. 1994, S. 121–133.

[RW04a] M. Ragni und S. Wölfl. „Branching Allen - Reasoning with Intervals in Branching Time“. In: *Spatial Cognition IV. Reasoning, Action, Interaction*. Springer, 2004, S. 323–343.

[RW04b] M. Ragni und S. Wölfl. „Branching Allen - Reasoning with Intervals in Branching Time - The composition tables“. In: *Spatial Cognition IV. Reasoning, Action, Interaction*. Springer, 2004, S. 323–343.

[VK06] F. Van Breugel und M. Koshkina. *Models and Verification of BPEL*. 2006.

[VK13] M. B. Vilain und H. A. Kautz. „Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report“. In: *Readings in Qualitative Reasoning about Physical Systems* (2013), S. 373.

[VKv89] M. Vilain, H. Kautz und P. van Beek. „Constraint Propagation Algorithms for Temporal Reasoning: a Revised Report“. In: *Readings in Qualitative Reasoning about Physical Systems*. Hrsg. von J. de Kleer D. S. Weld. San Mateo, CA: Morgan Kaufmann, 1989, S. 373–381.

[W85] R. W. *Petri Nets An Introduction*. Bd. 164. Springer, 1985.

[WKL12] S. Wagner, O. Kopp und F. Leymann. „Towards Verification of Process Merge Patterns with Allen's Interval Algebra“. Englisch. In: *Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012)*. Bamberg: CEUR Workshop Proceedings, März 2012, S. 1–8. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-10&engl=0.

[WMW11] M. Weidlich, J. Mendling und M. Weske. „Efficient consistency measurement based on behavioral profiles of process models“. In: *IEEE Transactions on Software Engineering* 37.3 (2011), S. 410–429.

[WRK+13] S. Wagner, D. Roller, O. Kopp, T. Unger und F. Leymann. „Performance Optimizations for Interacting Business Processes“. Englisch. In: *Proceedings of the first IEEE International Conference on Cloud Engineering (IC2E 2013)*. IEEE Computer Society, März 2013, S. 1–7.

Alle URLs wurden zuletzt am 13. 08. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift