FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Kontrollflusstransformation von BPMN zu Asbru

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Tschach
Matrikelnummer 0525381

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. Horst Eidenberger
Mitwirkung: Dr. Katharina Kaiser

Wien, 20.08.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

Thomas Tschach, Langegasse 16, 7210 Mattersburg

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.“

Mattersburg, am 20.08.2013

_____

(Unterschrift Verfasser)

Danksagung

Diese Arbeit wäre ohne die Hilfe von Betreuern, Familie und Freunden nicht möglich gewesen. Besonders bedanken möchte ich mich bei meinem Betreuer Herrn Prof. Dr. Horst Eidenberger und meiner Betreuerin Frau Dr. Katharina Kaiser für die stets sehr gute Unterstützung während der Erstellung der Diplomarbeit.
Weiterer Dank gilt meinen Eltern Sieglinde und Hans Tschach, die mich nicht nur finanziell unterstützt haben, sondern auch immer ein offenes Ohr für meine Probleme und Sorgen hatten.
Außerdem möchte ich mich bei meinen Freundinnen und Freunden bedanken, die für die nötige Ablenkung vom Studium sorgten, als dies notwendig war.

Kurzfassung

BPMN ist eine weitverbreitete und gut verständliche, grafische Beschreibungssprache für Geschäftsprozesse und wird von einer Vielzahl von Modellierungswerkzeugen unterstützt, während Asbru eine XML-basierte Beschreibungssprache für ausführbare, medizinische Leitlinien ist, die nur durch wenige grafische Modellierungswerkzeuge unterstützt wird.
Durch Definieren einer Modelltransformation zwischen BPMN und Asbru könnte die durch ausgereifte Werkzeuge gut unterstützte, standardisierte und wohlbekannte Notation herangezogen werden, um medizinische Leitlinien und Behandlungspläne in Asbru zu modellieren. Da die beiden Modelle unterschiedliche Paradigmen zur Repräsentation des Kontrollflusses nutzen, müssen einige Probleme im Bezug auf deren Transformation gelöst werden.
Ziel dieser Arbeit ist es, diese Probleme aufzuzeigen und Lösungsansätze in verwandten Bereichen zu finden. Daher werden (1) Transformationsstrategien diskutiert, die Lösungen für diese Transformationsprobleme auf Basis von Übersetzungen zwischen graforientierten Prozessbeschreibungssprachen und BPEL aufzeigen. Des Weiteren wird (2) beschrieben, wie und welche Kontrollflussmuster von BPMN und Asbru unterstützt werden, um etwaige Diskrepanzen aufzuzeigen. Anschließend wird (3) ein Mapping von BPMN-Elementen auf Asbru-Elemente definiert, und (4) die Eignung der oben angesprochenen Transformationsstrategien für die Umsetzung in einem BPMN-zu-Asbru Transformationssystem überprüft. Abschließend wird (5) das implementierte Transformationssystem anhand der Übersetzung der einzelnen Kontrollflussmuster beschrieben.
Die gewonnen Erkenntnisse und gezogenen Parallelen können verwendet werden, um eine Transformation zwischen BPMN und Asbru zu realisieren, wie dies am implementierten Prototypen gezeigt wurde.

Abstract

BPMN is a well known and understandable, graphical process description language for business processes and is supported by a variety of modelling tools, while Asbru is an XML based description language for executable clinical guidelines, which is supported only by a few graphical modelling tools.
Through defining a model transformation between BPMN and Asbru, the standardized and through matured tools good supported and well known BPMN notation could be used to model clinical guidelines or treatment plans in Asbru. Since both notations use different paradigms to represent the control flow, some problems have to be considered to get a correct transformation system.
The aim of this thesis is to show these problems and find solutions in similar scientific areas. So we (1) discus transformation strategies, which solve these transformation problems on concrete transformation processes between a variety of graph-oriented process description languages to BPEL. Another topic covered here is (2) which and how control-flow patterns are supported by BPMN and Asbru, to discover possible mismatches. Then a (3) mapping is defined to map BPMN elements to semantic equivalent Asbru elements. Afterwards (4) an appropriate transformation strategy out of the strategies introduced above is selected to be implemented in a BPMN-to-Asbru transformation system. Then the (5) implemented transformation system is described based on translation of the control-flow patterns.
The achieved results could be used to implement a transformation process between BPMN and Asbru, as it is done in the described prototype.

Inhaltsverzeichnis

1 Einführung

Medizinische Leitlinien (clinical guidelines) sind Dokumente, die eingesetzt werden, um die Qualität der Patientenbetreuung zu verbessern und durch Erhöhen der Effizienz Kosten einzusparen [1]. Sie liegen traditionell als textuelle Beschreibungen, Entscheidungsbäume oder in Form von Prozessdiagrammen vor [2]. Medizinische Leitlinien versuchen Diagnose- und Behandlungsprozesse in standardisierten, wohldefinierten Abläufen durchzuführen, um systematische Fehler in den Abläufen zu vermeiden und effiziente Behandlungen nach hohen Qualitätsstandards zu erreichen.

In den letzten beiden Dekaden wurde erheblicher Aufwand betrieben, um Informationssysteme zu schaffen, die medizinische Leitlinien automatisiert ausführen können. Diese Systeme sollen unter anderem das medizinische Personal bei der Auswahl der relevanten Leitlinien, der Anwendung einer ausgewählten Vorschrift und der Evaluierung der damit erzielten Resultate unterstützen [3].
Im Zuge des Asgaard Projekts wurde ein Informationssystem geschaffen, das die o.a. Aufgaben bewältigen soll und deshalb Asbru als die Beschreibungssprache für medizinische Leitlinien und Behandlungspläne hervorbrachte [3]. Mit Asbru können Abfolgen diagnostischer Abläufe oder Therapiepläne definiert werden. In ähnlichen Bereichen werden Prozessmodelle ebenfalls eingesetzt, z.B. in der Geschäftswelt, um Geschäftsabläufe zu definieren.

Die Business Process Model Notation (BPMN) ist eine grafische Notation, um Prozessabläufe zu beschreiben [4]. BPMN ist weit verbreitet und wird häufig verwendet, um Geschäftsprozesse zu beschreiben und wird bereits von einer Vielzahl an Tools unterstützt. BPMN dient lediglich zur Spezifikation von Prozessen. Es gibt jedoch keine Interpreter die BPMN-Modelle direkt ausführen können [5].

Die grundsätzliche Idee ist nun, die gute Unterstützung durch Modellierungswerkzeuge und die breite Akzeptanz von BPMN zu nützen, um Asbru-Modellinstanzen von medizinischen Leitlinien zu erstellen. Es soll daher eine Modelltransformation definiert werden, die BPMN-Modellinstanzen in Asbru-Modellinstanzen übersetzt. Da BPMN nicht die Ausdruckskraft hat, medizinischen Leitlinien vollständig zu beschreiben, ist dies auch nur teilweise möglich. Der Fokus der Transformation liegt auf dem Kontrollfluss, der im Idealfall vollständig aus dem Quellmodell übernommen werden soll.
Da BPMN den Kontrollfluss graforientiert ausdrückt, und Asbru dem blockorientierten Paradigma unterliegt, ist eine Transformation nicht immer trivial.

Wie eben erwähnt, gibt es zwei Paradigmen, den Kontrollfluss eines Prozesses und somit die Abfolge, in der Aktivitäten ausgeführt werden sollen, auszudrücken. Es gibt das graforientiert und das blockorientierte Paradigma:

(1) Graforientierte Modelle benutzen gerichtete Kanten, um die Abfolge von Aktivitäten und deren Abhängigkeiten zu definieren (Beispiel siehe Abb. 1). Vertreter dieser Sprachen sind z.B. Business Process Model Notation (BPMN) [6],UML Activity Diagrams [7], Petri Nets [8], Event-Driven Process Chains (EPC) [9], YAWL [10] und andere.

(2) Blockorientierte Modelle stellen strukturierte Konstrukte zu Verfügung, um die Abfolge von Aktivitäten darzustellen. Verschachtelt man diese, werden komplexe Prozesse geschaffen (Beispiel siehe Abb. 3). Vertreter dieses Paradigmas sind die Business Process Execution Language for Web Services (BPEL4WS oder BPEL) [11], XLANG oder auch Asbru [12].

Das verwendete Kontrollflussparadigma ist auch ein Kriterium, anhand dessen man die verschiedenen Prozessbeschreibungssprachen unterscheiden kann.

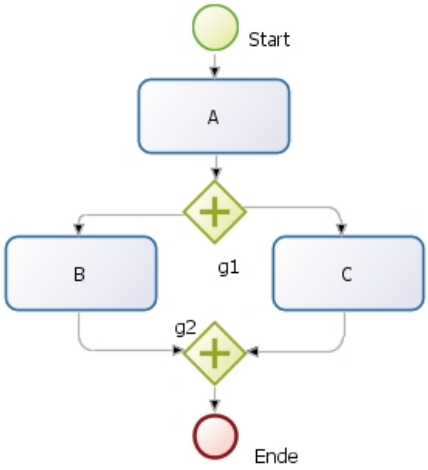


Abb. 1: Beispielprozess in graforientierter BPMN-Notation (grafische Darstellung)

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <process id="Prozess1" name="Prozess1">
    <startEvent id="Start" name="Start"/>
    <endEvent id="Ende" name="Ende"/>
    <parallelGateway id="g1" name="g1"/>
    <parallelGateway id="g2" name="g2"/>
    <task id="A" name="A"/>
    <task id="C" name="C"/>
    <task id="B" name="B"/>
    <sequenceFlow sourceRef="Start" targetRef="A" ... />
    <sequenceFlow sourceRef="A" targetRef="g1" ... />
    <sequenceFlow sourceRef="g1" targetRef="C" ... />
    <sequenceFlow sourceRef="C" targetRef="g2" ... />
    <sequenceFlow sourceRef="g1" targetRef="B" ... />
    <sequenceFlow sourceRef="B" targetRef="g2" ... />
    <sequenceFlow sourceRef="g2" targetRef="Ende" ... />
  </process>
</definitions>
```

Abb. 2: Beispielprozess aus Abb. 1 in standardisiertem BPMN XML-Format

Abb. 1 zeigt die grafische Darstellung eines Prozesses in BPMN Notation, in dem zuerst Aktivität A ausgeführt wird, und danach die Aktivitäten B und C gleichzeitig gestartet und parallel ausgeführt werden. Der Prozess endet, wenn beide Aktivitäten B und C beendet wurden. Abb. 2 zeigt den BPMN Prozess im dazugehörigen standardisierten BPMN XML-Format. Die Knoten werden durch die *startEvent*, *endEvent*, *task* und *parallelGateway* Elemente repräsentiert. Die Kanten werden durch das *sequenceFlow* Element dargestellt, die die Knoten per *sourceRef* und *targetRef* Attribut miteinander verbinden.

```
<process name="Beispielprozess">...
  <sequence>
  <invoke operation="A"…/>
  <flow>
    <invoke operation="B"…/>
    <invoke operation="C"…/>
  </flow>
  </sequence>
</process>
```

Abb. 3: Beispielprozess aus Abb. 1 im blockorientierten BPEL XML-Format

Im blockorientierten BPEL-Code (Abb. 3) wird dies durch die Verschachtelung des *flow* Elements im *sequence* Element ausgedrückt.

Graforientierte Modelle sind ausdruckstärker, da sie durch die Verkettung von Knoten mit Hilfe gerichteter Kanten Abfolgen von Aktivitäten definieren können. So können z.B. Schleifen durch zyklische Kantenfolgen konstruiert werden, die mehrere Ein- und Ausstiegspunkte haben. Dies lässt sich mit strukturierten Konstrukten, wie sie blockorientierte Sprachen verwenden, nicht darstellen. Da strukturierte Abläufe besser verständlich und weniger fehleranfällig als unstrukturierte sind, sollten unstrukturierte Abläufe möglichst vermieden werden [13].

Abb. 4 zeigt eine Schleife (A, B) in graforientierter BPMN-Notation mit einem Einstiegspunkt g1, über den die Schleife aktiviert wird, und den Ausstiegspunkten g2 u. g3, über die die Schleife verlassen werden kann.

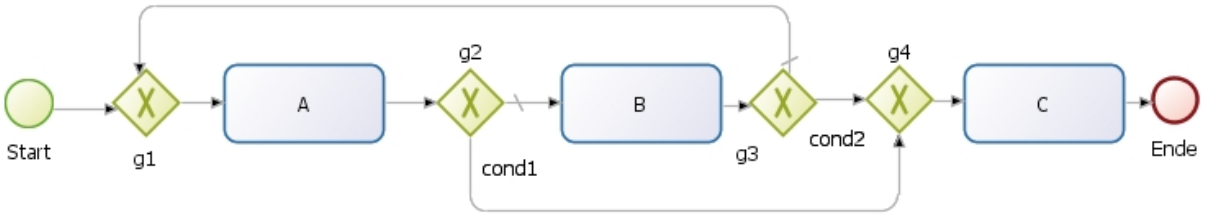


Abb. 4: unstrukturierte Schleife mit zwei Schleifenausstiegspunkten (g2, g3)

In blockorientierten Modellen gibt es dafür meist ein strukturiertes Element, in BPEL ist es z.B. das *while* Element, mit dem die Schleifenbedingung und der auszuführende Schleifenkörper definiert werden. Die Schleifenbedingung wird entweder vor bzw. nach dem Abarbeiten des Schleifenkörpers überprüft, um zu entscheiden, ob die Schleife verlassen oder eine weitere Iteration durchgeführt wird. Mehrere Ein- oder Ausstiegspunkte sind dabei nicht vorgesehen.
Somit kann die in Abb. 4 gezeigte Schleife nicht mit strukturierten Elementen blockorientierter Modelle dargestellt werden.

Der Beitrag dieser Arbeit besteht darin, Wissen und Erfahrungen aus bestehenden Transformationen von verschieden graforientierten Prozessmodellen zu BPEL aufzuarbeiten und auf die BPMN-zu-Asbru Transformation anzuwenden. Dabei werden bestehende Transformationsstrategien erläutert, auf Eignung für die BPMN-zu-Asbru Transformation geprüft, und ein Mapping zwischen BPMN und Asbru-Elementen definiert. Auf dieser Basis wurde ein Prototyp erstellt, der mit Hilfe von XSLT-Stylesheets eine geeignete Transformationsstrategie und das Mapping praktisch umsetzt. Das konzeptuelle Vorgehen des Transformationssystems wird mit Hilfe der Kontrollflussmuster als Vergleichssystem beschrieben.

Daher werden folgende Themen behandelt:
Zuerst wird dem Leser grundlegendes Wissen über das Quell- (BPMN) und Zielmodell (Asbru) und über die verwendete Umwandlungstechnologie (XSLT) der Transformation gegeben (Abschnitt 3.1).
Anschließend wird jeweils ein formales Modell für graf- und blockorientierte Beschreibungssprachen abstrahiert und definiert, um damit Modelleigenschaften und bereits bekannte Transformationsstrategien zu erläutern (Abschnitt 3.2).
Des Weiteren wird eine Methode diskutiert, die unstrukturierte Schleifen des Quellmodells in eine strukturierte Schleife umwandeln kann, wenn bestimmte Einschränkungen erfüllt sind (Abschnitt 3.3).
Für unstrukturierte Komponenten kommen andere Transformationsstrategien zum Einsatz als für strukturierte. D.h. es muss eine effiziente Methode gefunden werden, um das Eingangsmodell der Transformation in Komponenten zu teilen, und dies zu klassifizieren. Eine solche Methode wird in Abschnitt 3.4 erläutert.
Probleme bei der Modelltransformation entstehen dort, wo das Zielmodell Kontrollflussmuster (control-flow pattern) nicht unterstützt, die im Quellmodell modelliert werden können. Um im Vorfeld abzuklären wo Diskrepanzen auftreten, wird für BPMN und Asbru aufgelistet, wie und welche Kontrollflussmuster unterstützt werden (Abschnitt 0).
Abschnitt 4.2 definiert ein Mapping, das versucht BPMN-Elementen semantisch äquivalente Asbru-Elemente zuzuordnen.
Die in Abschnitt 3.2.5 vorgestellten Transformationsstrategien, die graforientierte in blockorientierte Modelle umformen, werden geprüft, ob und welche davon sich für die BPMN-zu-Asbru Umwandlung eignen, und wie die Strategie hierfür gestaltet werden muss (Abschnitt 4.3).

Die aus den vorrangegangenen Abschnitten gewonnenen Erkenntnisse werden genützt, um einen Prototypen zu implementieren, der ein BPMN-zu-Asbru Transformationssystem praktisch umsetzt. Das konzeptionelle Vorgehen des Transformationssystems wird im Kontext der Kontrollflussmuster erörtert (Abschnitt 4.4).
Kapitel 5 beschäftigt sich mit der Evaluierung des implementierten Software-Prototyps. In Abschnitt 6 werden die erreichten Ergebnisse zusammengefasst und ein Ausblick auf weiterführende Studien gegeben.

2 Methode

Ziel dieser Arbeit ist es, ein Transformationssystem zu schaffen, das, unter etwaigen Einschränkungen, den Kontrollfluss eines BPMN-Modells in ein äquivalentes Asbru-Modell übersetzt, und in einem Software-Prototyp umzusetzen.
Um dieses Transformationssystem zu erzielen ergeben sich vier wissenschaftliche Fragen, die in dieser Arbeit beantwortet werden:

1) Welche Probleme treten bei der Umwandlung von graf-basierten in blockorientierte Beschreibungssprachen auf?
2) Welche BPMN-Entitäten werden auf welche Asbru-Entitäten abgebildet?
3) Welche Kontrollflussmuster werden von beiden Modellen unterstützt? Welche nur von BPMN? Welche Diskrepanzen entstehen dadurch?
4) Welche generische Transformationsstrategie ist für die BPMN-zu-Asbru Transformation geeignet, und wie wird sie in XSLT implementiert?

Dieser Abschnitt beschreibt nun die Methoden, mit der die Antworten auf diese Fragen gefunden und die benötigten Ergebnisse erzeugt werden, um das Transformationssystem zu implementieren.

Zuerst wird die grundsätzliche Problematik beschrieben, wenn Modelle des graforientierten Paradigmas in Modelle, die dem blockorientierten Paradigma folgen, übergeführt werden, und warum eine Übersetzung eines BPMN-Modells in ein Asbru-Modell, aufgrund der unterschiedlichen Repräsentationsparadigmen nicht trivial ist. Dies beantwortet die erste wissenschaftliche Fragestellung und stellt einen Teil des State-of-the-Art Berichts der Arbeit dar. Er basiert auf Literaturrecherche in Arbeiten, die sich mit ähnlichen Transformationssystemen beschäftigen, hauptsächlich mit BPMN-zu-BPEL Transformationen.

Des Weiteren werden Transformationsstrategien beschrieben, die sich bei der Übersetzung von diversen graforientierten Prozessmodellen zu BPEL-Modellen bewährt haben. Diese Strategien werden ebenfalls durch Literaturrecherche ermittelt, im State-of-the-Art Bericht zusammengefasst und später auf Anwendbarkeit im Prototyp des BPMN-zu-Asbru Übersetzungssystems geprüft.

Als Rahmen für den Vergleich ob und wie BPMN bzw. Asbru Kontrollflussmuster unterstützt, werden die in [14] definierten Kontrollflussmuster (workflow control-flow patterns) herangezogen. Der Vergleich von BPMN und Asbru bezüglich der Kontrollflussmuster, wird im State-of-the-Art Bericht ausgeführt und basiert auf Literaturrecherche in Arbeiten, die analysieren, welche Kontrollflussmuster und wie sie durch BPMN bzw. Asbru unterstützt werden. Dies beantwortet die dritte wissenschaftliche Fragestellung dieser Arbeit.

Unerlässlich für die Transformation eines Modells in ein anderes ist die Identifizierung semantisch äquivalenter Konstrukte im Quell- (BPMN) und Zielmodell (Asbru). Dieses Mapping wird verwendet, um korrekte Konstrukte bei der Transformation im Zielmodell, entsprechend den Vorgaben des Quellmodells, zu generieren und entsteht aus der Evaluierung der Spezifikationen beider Prozessmodelle. Dies beantwortet die zweite wissenschaftliche Fragestellung dieser Arbeit.

Vor der Implementierung des Prototyps des Transformationssystems, wird eine Transformationsstrategie festgelegt, die im Software-Prototyp umgesetzt werden soll. Die im State-of-the-Art Bericht vorgestellten Transformationsstrategien, von diversen graforientierten Prozessmodellen zu BPEL-Modellen, werden anhand der Asbru-Spezifikation evaluiert und auf Anwendbarkeit geprüft. Dies beantwortet den ersten Teil der vierten wissenschaftlichen Fragestellung dieser Arbeit.

Der zweite Teil der vierten wissenschaftlichen Frage, wird durch die Implementierung des Transformationssystems beantwortet. Der implementierte Prototyp setzt das definierte Mapping zwischen BPMN und Asbru und die ausgewählte Umwandlungsstrategie in einem ausführbaren Transformationssystem um, und benützt als Transformationstechnologie XSLT. Dadurch soll die Umsetzbarkeit und Korrektheit des Transformationssystems praktisch nachgewiesen werden und kann an Beispielen nachvollzogen werde. Bei der Evaluierung werden aussagekräftige Testbeispiele durch das Transformationssystem übersetzt, das entstandene Asbru-Modell mit der Asbru-DTD syntaktisch geprüft. Die semantische Äquivalenz wird durch manuelle Prüfung ermittelt, dabei wird als Hilfsmittel ein Tool zur grafischen Darstellung von Asbru-Plänen verwendet.

Im folgenden Abschnitt werden dem Leser Grundlagen über BPMN, Asbru und die Transformationstechnologie XSLT vermittelt, die die für diese Arbeit wesentlichen Aspekte zusammenfassen.

3 State-of-the-Art

In diesem Abschnitt werden allgemeine theoretische Grundlangen zu BPMN, Asbru und Transformationssprachen, im speziellen XSLT vermittelt (Abschnitt 3.1). Außerdem werden Transformationsstrategien und Transformationsmethoden besprochen, die sich bei der Übersetzung zwischen vergleichbaren Prozessmodellen bewährt haben (3.2). Des Weiteren werden Methoden vorgestellt, um bestimmte unstrukturierte Schleifen in strukturierte umzuwandeln (Abschnitt 3.3) und Komponenten zu klassifizieren (Abschnitt 3.4).

3.1 Grundlagen

In diesem Abschnitt wird versucht, grundlegendes Wissen über die in dieser Arbeit relevanten Prozessbeschreibungssprachen (BPMN und Asbru) bzw. Transformationstechnologien (XSLT) zu vermitteln.

3.1.1 Business Process Model Notation (BPMN)

BPMN ist ein graforientiertes Prozessmodell, das entwickelt wurde, um Geschäftsprozesse zu beschreiben und Abläufe grafisch darzustellen, mit dem Schwerpunkt auf den Kotrollfluss des Prozesses. BPMN wird durch eine Vielzahl von Modellierungswerkzeugen unterstützt, BPMN-Modelle sind jedoch nicht ausführbar [4].

Abb. 5 zeigt die wichtigsten BPMN Elemente, die zur Verfügung stehen, um einen Prozess zu beschreiben.

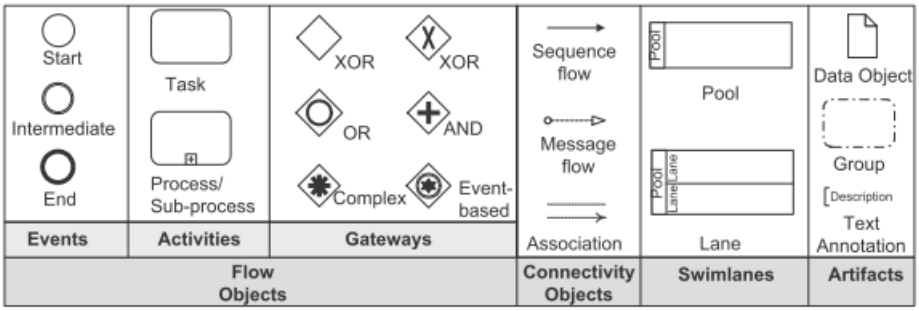


Abb. 5: BPMN Kernelemente [15]

Das zentrale Element eines BPMN-Prozesses ist der Task, der eine nicht weiter zerlegbare Aktivität symbolisiert. Mit Subprozessen (Sub-process) kann ein großer Prozess in kleinere unterteilt bzw. aus bestehenden Prozessteilen ein Gesamtprozess zusammengesetzt werden und ruft aus einem Prozess heraus einen anderen Prozess auf.
Die Abfolge, in der die Tasks ausgeführt werden, wird über die Kanten (Sequence flow) bestimmt, mit denen die Flow Objects (Knoten) miteinander verbunden werden. Der Beginn und das Ende eines BPMN-Prozesses werden durch Start- und End-Events bestimmt. Um den Kontrollfluss zu verzweigen oder zu vereinigen, werden Konnektoren (Gateways) eingesetzt. Es gibt unterschiedlich Arten von Konnektoren, um exklusive (XOR), inklusive (OR) und parallele (AND) Verzweigungen zu modellieren. Aber auch komplexere Verzweigungsbedingungen sind möglich.
Um darzustellen, wer für die Ausführung eines Tasks im Geschäftsprozess verantwortlich ist, stellt BPMN Pools und Lanes zur Verfügung. Tasks, die im Bereich eines Pools oder einer Lane definiert sind, fallen in dessen Verantwortungsbereich. Pools können mehrere Lanes enthalten und stehen hierarchisch über den Lanes. Dadurch lassen sich hierarchische Verantwortungsbereich wie z.B. Unternehmen/Abteilungen oder Abteilung/Mitarbeiter darstellen [16].

Mit diesen BPMN Kernelementen lässt sich nun der Kontrollfluss eines Prozesses modellieren.
Ein Beispiel eines BPMN-Prozesses und dessen XML-Serialisierung finden sind in, Abschnitt 1.1

3.1.2 Asbru Grundlagen

Asbru ist eine Prozessbeschreibungssprache, um ausführbare medizinische Leitlinien darzustellen. Asbru charakterisiert sich dadurch, dass die Ausführung von Aktivitäten auch von zeitlichen Bedingungen (*time annotation*) abhängig gemacht und auf die Daten eines beliebigen Patienten angewandt werden kann.

Asbru ist ein DTD-spezifizierte XML-Sprache (aktuelle Versions 7.3h). Eine Ausführungseinheit, der s.g. „Asbru-Interpreter“ [17] hat einen Großteil der Asbru-Spezifikation implementiert. Da sich der Interpreter noch in Entwicklung befindet, sind jedoch nicht alle Elemente der Spezifikation realisiert.

Das Top-Level-Element einer Asbru-Leitlinie ist das *plan-library* Element. Darin werden Definitionen, wie Datentypen, Parameter, Variablen, Konstante, komplexe Zeitdefinitionen, etc., auf Domänen- bzw. Library-Ebene (*domain-defs* bzw. *library-defs*) erstellt und Pläne (*plans*) definiert. Datentypen, Parameter, Variablen, etc. können auf Library-Ebene definiert werden und sind somit in allen Plänen des *plan-library* Elements gültig und sichtbar. Die vorhin genannten Elemente können aber auch auf Domänen-Ebene definiert werden und beschreiben so eine bestimmte Umgebung in der ein Plan ausgeführt werden kann. Domänenspezifische Definitionen sind unabhängig von der Plan Library und können in verschiedene Plan Libraries angewandt werden [12].

In einer Plan Library kann jeder Plan *Preferences, Intentions, Conditions, Effects*, und einen *Plan Body* enthalten. Es können aber auch Default-Einstellungen, Parameter, Variablen auf Planebene und Rückgabewerte definiert werden [2], [12].
In den Preferences können Kriterien definiert werden, die für die Selektierung eines Plans von Belang sein können. Kriterien könnten sein z.B. die Kosten oder die entstehende Ressourcenbelegungen (z.B. Gerät A ist für zwei Stunden belegt), die bei der Anwendung des Plans entstehen
Mit den Intentions werden die Ziele auf hohem Level ausgedrückt, die durch den Plan erreicht werden sollen. Dies ist wichtig, um die Resultate des Plans mit den Zielen zu vergleichen und die Effizienz des Plans zu ermitteln. Ein Ziel könnte z.B. sein, den Blutzuckerwert eines Patienten zu senken.
In den Effects können mathematische Zusammenhänge zwischen Eingangsparameter und messbaren Größen beschrieben werden, die auch mit Wahrscheinlichkeiten ausgestattet werden können. So könnte eine Funktion definiert werden, mit der der Zusammenhang der verabreichten Insulinmenge mit dem angestrebten Blutzuckerwert beschrieben wird.

Mit Plänen werden strukturierte Abläufe definiert, indem sie ineinander verschachtelt werden, aber sie können auch Aktionen darstellen, die nicht weiter zerlegt werden können.
Ob ein Plan ausgeführt wird, wird von der Erfüllung mehrerer Bedingungen ausgelöst. Je nachdem welche Bedingung erfüllt ist, befindet sich eine Planinstanz in verschiedenen Zuständen (siehe Abb. 6). Diese Bedingungen werden in den Conditions eines Plans definiert [12].

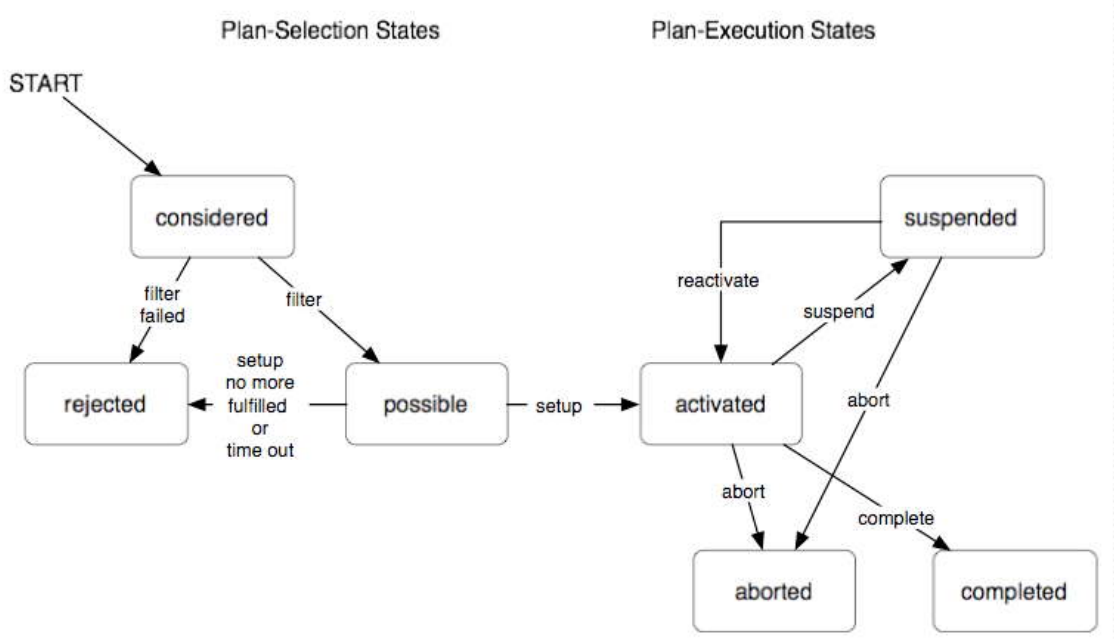


Abb. 6: mögliche Zustände und Übergangsbedingungen eines Asbru Plans vor und bei der Ausführung [12]

Damit ein Plan aktiviert wird, muss die filter- und die setup-precondition erfüllt sein. Ist ein Plan einmal aktiviert, wird er durch Erfüllen der complete-condition beendet, und der Nachfolgeplan entsprechend der Prozessbeschreibung aufgerufen. Aber er kann auch vorübergehend deaktiviert (suspend-condition), wieder reaktiviert (reactivate-condition) oder komplett abgebrochen (abort-condition) werden, wenn die entsprechenden Bedingungen erfüllt sind.
Die filter- und die setup-precondition können noch mit einem Attribut (*confirmation-required*="yes") versehen werden, dass eine manuelle Bestätigung erforderlich macht. D.h. wenn die filter- bzw. die setup-precondition erfüllt ist, wird erst durch das Quittieren eines Benutzers der Übergang in den nächsten Zustand (possible bzw. ready) aktiviert.

Um komplexe Abläufe darzustellen, müssen Pläne verschachtelt werden. Dies erfolgt im Plan Body eines Plans, in dem die Aktionen bzw. Abfolgen von Plänen definiert werden. Abhängig vom Plan-Typ (PT) des *subplans* Elements werden die in einem Plan verschachtelten Teilpläne entweder in sequenziell (PT: *sequential*), parallel (PT: *parallel*), in beliebiger Abfolge hintereinander (PT: *any-order*) oder in beliebiger, auch paralleler Abfolge (PT: *unordered*) ausgeführt.
Um Schleifen auszudrücken, die einen Planabschnitt abhängig von einer Abbruchbedingung (termination-condition) iterativ wiederholen, stellt Asbru den Iterativen Plan zu Verfügung, mit dem das Verhalten ausgedrückt werden kann [12].

Sind in einem Plan Teilpläne verschachtelt (*subplans* Element), dann gilt der Plan als beendet, wenn alle Teilpläne beendet wurden, und die *complete-condition* erfüllt ist. Dieses Verhalten kann jedoch über die *Fortsetzungsbedingung* (*continuation condition*) des Plans beeinflusst werden. Mit der Fortsetzungsbedingung, ausgedrückt durch das *wait-for* Element, kann spezifiziert werden, wie viele bzw. welche Teilpläne erfolgreich beendet sein müssen, damit der Plan, der sie verschachtelt, beendet werden kann.
Es gibt insgesamt sieben mögliche Modi, die Fortsetzungsbedingung zu konfigurieren (siehe auch [12]). In dieser Arbeit werden jedoch nur die dafür relevanten Modi „*one*" und „*all*" erläutert.

- One: Damit wird ausgedrückt, dass ein einzelner beliebiger Teilplan erfolgreich beendet sein muss, damit der verschachtelnde Plan beendet werden kann.
- All: Der verschachtelnde Plan kann erst beendet werden, wenn alle Teilpläne erfolgreich beendet wurden.

Mit der Fortsetzungsbedingung wird festgelegt, wie viele Subpläne zwingend erfüllt sein müssen, damit der verschachtelnde Plan beendet werden kann. Oft ist es jedoch so, dass mehr Subpläne bei der Ausführung aktiviert werden als zwingend nötig wären, um die Fortsetzungsbedingung zu erfüllen. Das Default-Verhalten besagt, dass nur auf die Beendigung der benötigten Anzahl an Subplänen gewartet wird, z.B. bei wait-for=one auf einen einzelen Subplan. Das Default-Verhalten kann jedoch durch das *wait-for-optional-subplans* Attribut des *subplans* Elements beeinflusst werden.
Wird das Attribut mit dem Wert *yes* belegt (*wait-for-optional-subplans*="yes"), wird auch auf die Beendigung optionaler Subpläne gewartet, solange keinen zeitlichen Bedingungen widersprochen wird [12].

Für die Bedingungen können unter anderem Variablen verwendet werden, die über Vergleichsoperatoren miteinander oder mit Konstanten verglichen werden. Variablen gibt es mit verschiedenen Sichtbarkeitsbereichen (Scope) [12]:

- Globale Variablen: Sie sind in jedem Plan in der gesamten Plan Library sichtbar und müssen im *library-defs* Element definiert werden.
- Lokale Variablen: Sie sind nur in dem Plan sichtbar, in dem sie definiert wurden und werden im *value-defs* Element eines Plans deklariert.
- Kontext-Variablen: Sind Teil eines Kontexts, der einen Teil der Umgebung repräsentiert und müssen in einem *domain* Element einer Plan Library definiert werden.

Lokalen und globalen Variablen können mit Hilfe des *variable-assignment* Elements neue Werte zugewiesen werden, während Kontext-Variablen neue Werte mit dem *set-context* Element erhalten.
In diese Arbeit sind jedoch nur die globalen Variablen von Belang. Mehr dazu in Abschnitt 4.3.4.

Ein weiteres Element, welches zur Formulierung einer Bedingung verwendet werden kann, ist das *plan-state-constraint* Element. Damit lassen sich Bedingungen beschreiben, die erfüllt sind, wenn ein referenzierter Plan einen definierten Zustand erreicht. Dadurch können z.B. Pläne aktiviert werden, wenn der Vorgängerplan beendet (completed-state) wurde [12]. Genaueres dazu siehe Abschnitt 0.
Ein Beispiel für eine Asbru Plan Library ist zur Veranschaulichung in, Abschnitt 1.2 zu finden.

3.1.3 Transformationssprachen und XSLT

Allgemein dienen Modelltransformationen dazu, ein bzw. mehrere Eingangsmodell(e) anhand von Transformationsregeln in ein bzw. mehrere Ausgangsmodell(e) zu transformieren. Diese Transformationsregeln werden definiert und können von einer Ausführungseinheit automatisiert durchgeführt werden. Es gibt hier drei Ansätze wie diese Transformationsregeln definiert werden können [18]:

1) Direkte Modell Manipulation (direct model manipulation): Bei diesem Ansatz wird ein API zur Verfügung gestellt, mit dem durch allgemeine Programmiersprachen (general-purpose languages) auf das Modell zuggegriffen werden kann, um es zu manipulieren. Beispiele bzgl. XML direkt zu manipulieren sind Document Object Model (DOM) und Simple API for XML (SAX).
2) Transformationssprache (transformation language support): Hier wird eine Sprache definiert, mit der Transformationsregeln ausgedrückt, zusammengesetzt und angewandt werden können. Beispiele hierfür sind Query/Views/Transformations (QVT [19]) und die Atlas Transformation Language (ATL [20]) [21].
3) Zwischenformat Repräsentation (intermediate representation): Dieser Ansatz basiert darauf, das Quellmodell in ein anderes, standardisiertes Zwischenformat, meist XML, zu exportieren

und dieses Zwischenformat mit einem externen Werkzeug ins Zielmodell zu transformieren. Z.B. haben UML Werkzeuge meist die Möglichkeit Modelle in XMI[1] zu exportieren und mittels XSLT ins Zielmodell zu transformieren.

Da XSLT in dieser Arbeit zur Umsetzung des Transformationssystems eingesetzt wird, wird im folgenden Abschnitt näher darauf eingegangen.

3.1.3.1 XSLT

XSLT steht für Extensible Stylesheet Language Transformtion [22] und ist neben Extensible Stylesheet Language Formating Objects (XSL-FO) Teil der Extensible Stylesheet Language (XSL [23]) [24]. XSLT ist in der Version 2.0 seit Jänner 2007 eine Empfehlung des World Wide Web Consortium (W3C) bezüglich XML Transformationen und nützt zur Navigation die XML Navigation Language (XPath [25]) 2.0, ein weiterer W3C Standard [22].

XSLT definiert ein XML-Format, mit dem Regeln festgelegt werden, die es erlauben, ein XML-Dokument in ein anderes textbasiertes Dokument (z.B. XML, HTML oder unstrukturierter Text) zu transformieren. Ein XSLT-Prozessor verwendet dabei das XSLT-Stylesheet, in dem die Transformationsregeln festgelegt sind, und das zu transformierende XML-Dokument und generiert daraus, entsprechend dem Stylesheet, das ausgehende XML-Dokument (siehe Abb. 7) [24].

Ein XSLT-Stylesheet besteht aus mehreren Template-Regeln. Eine Template-Regel besteht aus einem Muster und dem Template (siehe Abb. 8). Der XSLT-Prozessor liest das Eingangsdokument ein und traversiert über den entstandenen Eingangsbaum. Passt ein Muster einer Template-Regel auf Knoten des Eingangbaumes, wird das entsprechende Template auf die Knoten angewandt. Das Template definiert dabei Werte, die in den Ausgangsbaum geschrieben werden, der dem Ausgangsdokument entspricht. Das Muster entspricht einem Ausdruck in XPath 2.0 und identifiziert so die Knoten, auf die das Template angewendet wird [22], [24].

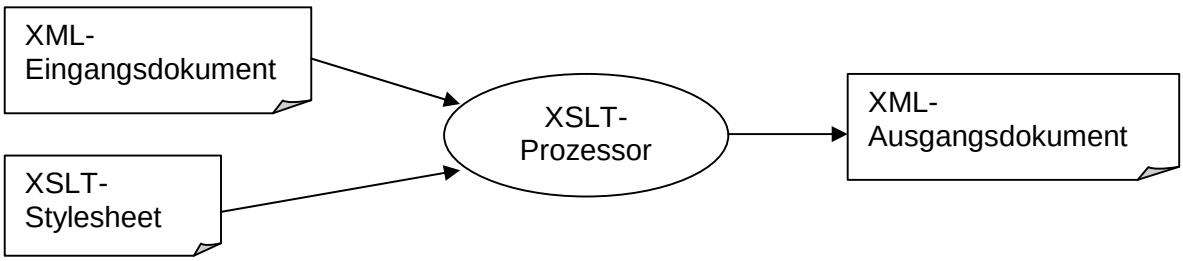


Abb. 7: Ablauf einer Transformation mittels XSLT-Prozessors

```
<xsl:template match="Muster">
        Template
</xsl:template>
```

Abb. 8: Grundsätzlicher Aufbau einer Template-Regel eines Stylesheets

Das Template kann neben normalem Text auch weitere XSLT-Anweisungselemente enthalten, die der XSLT-Prozessor bei der Transformation interpretiert, mit dem statischen Text des Templates kombiniert und ins Ausgangsdokument schreibt.
Für weitere und vertiefende Informationen über XSLT sei auf [22], [24], [26] verwiesen.

[1] XMI ist ein XML-basierter Standard, um UML-Modelle zu exportieren und in anderen UML-Werkzeugen zu importieren

3.2 Transformationen von Workflow-Modellen

Verschiedenste Sprachen wurden entwickelt, um Business Process Models (BPM) bzw. Workflow-Models (WfM) zu beschreiben. Die Zielsetzungen dieser Sprachen unterscheiden sich und sind vielfältig. Man verwendet sie u.a. zur Dokumentation, zur formalen Analyse von Geschäftsprozessen oder um Prozesse zu erstellen, in denen die Abfolge von Aufrufen bestehender Services definiert werden (Komposition, Choreografie) [27].

Workflows sind Instanzen eines WfM und beschreiben im Wesentlichen Aktivitäten und die Abfolge, wie sie ausgeführt werden, um einen Geschäftsprozess darzustellen. Bei der Definition der Abfolge der auszuführenden Aktivitäten spricht man auch vom Kontrollfluss (control flow) des Prozesses. Dient der Workflow nur zur Dokumentation, spricht man von einem abstrakten Prozess (abstract process). Ist der Workflow auch ausführbar, spricht man von einem ausführbaren Prozess (executable process) [28].

In den letzten Jahrzehnten wurde versucht, die Vielfalt an BPM Sprachen in einem Standard zu bündeln. Diese Bemühungen scheiterten jedoch hauptsächlich an den unterschiedlichen, domänenspezifischen Konstrukten der verschiedenen Sprachen und an den ungleichen Paradigmen, wie der Kontrollfluss dargestellt wird [29].
Wie schon in Abschnitt 0 angeführt gibt es zwei Paradigmen, wie man den Kontrollfluss repräsentieren kann, nämlich das graforientierte und das blockorientiert Kontrollflussparadigma. Beim graforientierten Kontrollflussparadigma werden gerichtete Kanten verwendet, um die logischen und zeitlichen Abhängigkeiten zwischen den Knoten des Workflows zu modellieren. Dagegen wird bei BPM Sprachen, die dem blockorientierten Kontrollflussparadigma folgen, strukturierte Konstrukte benutzt um sequentielle, alternative, parallele Abläufe und Schleifen zu modellieren [29].

Modelle sind abstrahierte, vereinfachte Darstellungen des zu beschreibenden Objekts, im Fall von WfM eben Workflows. In der Softwareentwicklung waren Modelle meist Entwürfe für das zu implementierende System und dienten als Vorlage für den Programmierer, der die entsprechende Implementierung realisierte.
Model-Driven-Development (MDD) geht einen Schritt weiter und generiert auf Basis des Modells ausführbaren Code. Manchmal sind davor noch Modelltransformationen notwendig, um die Code-Generierung zu optimieren. Dabei wird das Quellmodell nicht in ausführbaren Code, sondern in ein anderes, das Zielmodell, transformiert.
Modelltransformationen werden auch verwendet, um Modelle an das Eingangsformat eines anderen Tools oder eines Standards anzupassen, da der Entwicklungsprozess möglicherweise von mehreren Werkzeugen unterstützt wird, und ein Modell unter ihnen ausgetauscht werden soll.
Des Weiteren können Modeltransformationen eingesetzt werden, wenn das Quellmodell analysiert werden soll und es dafür keine entsprechenden Formalismen besitzt, das Zielmodell jedoch schon [29].
Wird ein WfM nur unzureichend durch vorhandene Werkzeuge unterstützt, kann eine Modelltransformation eine Lösung darstellen. Das gewünschte Modell wird in einem anderen, etablierten Modell mit guter Modellierungsinfrastruktur modelliert, und anschließend in das eigentlich gewünschte Workflow-Modell umgewandelt.

Abgesehen von den Gründen, warum ein Modell in ein anderes umgewandelt wird, gibt es generelle Schlüsselanforderungen an diese Transformationen, die wie folgt lauten [4]:

- Vollständigkeit (completness): Jede mögliche Instanz des Quellmodells soll in eine entsprechende Instanz des Zielmodells umgewandelt werden können.

- Automatisierung (automation): Die Instanz des Zielmodells soll automatisch generiert werden und manuelle Transformationsschritte vermieden werden.
- Lesbarkeit (readability): Die generierte Zielmodellinstanz soll gut lesbar und auch ohne weitere Analyseschritte für menschliche Betrachter verständlich sein.

Diese Anforderungen zu erfüllen ist nicht trivial und auch nicht immer vollständig erreichbar. Vor allem wenn das Quellmodell weniger restriktiv als das Zielmodell ist und Konstrukte möglich sind, die im Zielmodell nicht unterstützt werden, leidet die Vollständigkeit der Umwandlung.
In den nächsten Abschnitten wollen wir uns mit allgemeinen Definitionen und Eigenschaften von graf- und blockorientierten Beschreibungssprachen beschäftigen und einige Transformationsstrategien aufzeigen.

3.2.1 Definition graforientierter Beschreibungssprachen

Graforientierte Beschreibungssprachen für Workflow-Modelle beschreiben eine Instanz durch einen Prozessgrafen (process graph, PG), der aus Knoten und Kanten besteht. Knoten können untrennbare Aktionen, Konnektoren, Start- und Endzustände sein. Diese Knoten werden mit Kanten verbunden und definieren so die zeitlichen und logischen Abhängigkeiten zwischen den Knoten [29]. Kanten können auch eine Bedingung (guard) haben, die boolsche Ausdrücke sind. Ist die Kantenbedingung erfüllt, dann wird der Zielknoten der Kante aktiviert.
Konnektoren verzweigen bzw. vereinigen den Kontrollfluss und können anhand ihres Verhaltens in AND-, OR- oder XOR-Splits bzw. -Joins unterteilt werden.
In [29] wird eine Prozessbeschreibungssprache, namens *Process Graph,* definiert, die die Kernelemente von YAWL [10] und EPC [9] abstrahiert. Die Process Graph Definition enthält genau jene Elemente, um den Kontrollfluss mit Hilfe von Kanten und Knoten darzustellen. Die meisten graforientierten Modelle enthalten diese Elemente als Kern ihrer Spezifikation, aber bauen diese auch mit zusätzlichen Elementen aus, die die Ausdruckkraft erhöhen.
Die folgende formale Spezifikation von *Process Graph* (Definition 1) kann erweitert werden, um den Anforderungen anderer graforientierter Modelle gerecht zu werden. Für BPMN z.B., müsste die u.a. Definition um das Konzept der Events erweitert werden.

Die nachfolgende Definition 1 stellt die formale Definition der Sprache Process Graph dar. Wir verwenden diese Spezifikation zur Darstellung eines Prozessgrafen, um dessen Bestandteile zu erläutern und die Transformationsstrategien in Abschnitt 3.2.5 verständlicher zu machen.

Definition 1: Ein Prozessgraf (PG) ist ein Tupel $PG = \{S, E, F, C, l, A, g\}$, das aus vier disjunkten Mengen S, E, F, C, einer Funktion $l: C \rightarrow \{AND, OR, XOR\}$, einer binären Relation $A \subseteq (S \cup F \cup C) \times (E \cup F \cup C)$ und einer Funktion $g: A \rightarrow expr$ besteht [29]. Weiters muss gelten:

- S ist die Menge an Startzuständen. $|S| \geq 1$und $\forall s \in S: |succ^2(s)| = 1 \wedge |pred^3(s)| = 0$
- E ist die Menge der Endzustände. $|E| \geq 1$ und $\forall e \in E: |succ(e)| = 0 \wedge |pred(e)| = 1$
- F ist die Menge an Aktivitäten. $\forall f \in F: |succ(f)| = 1 \wedge |pred(f)| = 1$
- C ist die Menge an Konnektoren. $\forall c \in C: |succ(c)| = 1 \wedge |pred(c)| > 1 \vee |succ(c)| > 1 \wedge |pred(c)| = 1$

[2] succ(n) gibt die Menge der Nachfolgeknoten eines Knotens n an: $succ(n) = \{x \in N | (n, x) \in A\}$
[3] pred(n) gibt die Menge der Vorgängerknoten eines Knotens n an: $pred(n) = \{x \in N | (x, n) \in A\}$

- Funktion l spezifiziert den Typ eines Konnektors $c \in C$ als AND, OR oder XOR. $l: C \rightarrow \{AND, OR, XOR\}$

- A definiert die Menge der gerichteten Kanten, die die Knoten im Graf verbinden. Es gibt keine Kanten mit identen Start- und Zielknoten (reflexiv) und keine paarweise identen Kanten (mehrfach Kanten). $A \subseteq (S \cup F \cup C) \times (E \cup F \cup C)$

- Die Funktion g gibt für eine Kante $a \in A$ die Kantenbedingung, also den boolschen Ausdruck $expr$, zurück. Gehen die Kanten von einem Konnektor des Types XOR-Split aus, dann müssen alle Kantenbedingungen so gestaltet sein, dass exakt eine Kantenbedingung erfüllt ist (mutually exclusiv). Kantenbedingungen können nur auf Kanten definiert werden, die von einem XOR- oder OR-Split ausgehen $((c, n) \in A | \, l(c) \neq AND \wedge n \in E \cup F \cup C)$. Die Bedingungen aller anderen Kanten, also Kanten von AND-Splits und Sequenzen, sind immer erfüllt.

Mit der anschließenden Bestimmung der transitiven Hülle, kann die Erreichbarkeit zweier Knoten innerhalb des PG dargestellt werden.

Definition 2: Transitive Hülle (transitiv closure) A* eines Prozessgrafen $PG = \{S, E, F, C, l, A, g\}$ ist die transitive Hülle über die Kantenmenge A. D.h. ist $(n1, n2) \in A^*$, dann gibt es einen Pfad von n1 nach n2 über eine Folge von Kanten aus A [29].

3.2.2 Definition blockorientierter Beschreibungssprachen

Blockorientierte Beschreibungssprachen definieren den Kontrollfluss über strukturierte Aktivitäten. Durch deren Verschachtelung können komplexe Abläufe erstellt werden.
Es gibt Element für sequentiellen (sequence), alternativen (switch), parallelen Kontrollfluss (flow), so wie Schleifen (while) und alternative Startbedingungen (pick). Über Links können Aktivitäten verbunden werden, und komplexere Synchronisationen erzielt werden.

BPEL ist ein Beispiel für ein blockorientiertes Workflow-Model. Es eignet sich gut zur Veranschaulichung von Transformationsstrategien, da man in der Literatur viele verschiedene Transformationsprozesse findet, die Übersetzungen von oder zu BPEL implementieren. Es können verschiedene Übersetzungsansätze mit BPEL umgesetzt werden, da es kein rein blockorientiertes Prozessmodell darstellt, sondern auch graforientierte Elemente (Links) aufweist. Außerdem besitzt BPEL interessante Konstrukte, wie Event-Handler, mit denen ein Kontrollfluss simuliert werden kann. Aus diesem Grund eignet sich BPEL gut, um mit ihm Transformationsstrategien von und zu blockorientierten Prozessmodellen zu veranschaulichen.

In [29] werden die Kernkonzepte von BPEL abstrahiert, und daraus das Workflow-Model namens BPEL Control Flow (BCF) definiert. BCF kann jedoch auf jede andere blockorientierte Prozessbeschreibung umgelegt werden.
Da BCF eine vereinfachte Darstellung von BPEL ist, in der die für die Repräsentation des Kontrollflusses relevanten Elemente definiert werden, und die nachfolgenden Strategien auf BPEL Mechanismen zurückgreifen, vereinfacht ein grundlegendes Wissen über BPEL das Verständnis für die nachfolgenden Abschnitte.

3.2.2.1 Business Process Execusion Language (BPEL) Grundlagen

BPEL ist eine weitverbreitete Sprache im Bereich der Web Service Orchestration. Dabei werden Geschäftsprozesse auf Basis bestehender Web Services (WS) erstellt und selbst als Web Service verfügbar gemacht [27]. BPEL wird in einem XML-Format dargestellt und ist weitgehend blockorientiert und ausführbar.

BPEL kombiniert Aktivitäten zu einem Workflow. Man kann zwei Arten von Aktivitäten unterscheiden, nämlich *Basisaktivitäten* (*basic activities*) und *strukturierte Aktivitäten* (*structured activities*) [4].
Basisaktivitäten führen einfache Aktionen aus, wie Aufruf eines Web Services (*invoke*), warten auf einen Aufruf (*receive*) und eine Nachricht als Rückgabewert auf einen Aufruf an den Aufrufer zurückschicken (*reply*) [27], [4]. Weiters gibt es Basisaktivitäten *exit*, *wait* und *empty*, mit denen der gesamte Prozess beendet, eine spezifizierte Zeitspanne gewartet oder nichts gemacht wird [27], [5].

Über strukturierte Aktivitäten wird in BPEL der Kontrollfluss des Prozesses definiert, durch deren Verschachtelung komplexe Abfolgen erzeugt werden können. Zu ihnen zählen *sequence*, um sequentielle Abläufe, *switch*, um alternative Abläufe, und *flow*, um parallele Abläufe zu beschreiben. Mit *while* werden Schleifen ausgedrückt, und mit *pick* werden alternative Startbedingungen für eine Aktivität definiert. *Scope* gruppiert Aktivitäten und begrenzt den Gültigkeitsbereich von *event-* und *exception-handler* [5].

Event-Handler können mit einem Scope assoziiert werden und stellen Event-Aktions-Regeln (*event-action rules*) dar. D.h. wird ein Prozess gerade ausgeführt und befindet sich innerhalb eines Scopes mit dem ein Event-Handler assoziiert ist und es tritt ein passender Event auf, dann wird die entsprechende Aktivität des Handlers ausgeführt [4].
BPEL kann den Kontrollfluss nicht nur blockorientiert festlegen, sondern hat auch graforientierte Elemente, die *Links*. Innerhalb eines flow Elements können Aktivitäten mit Links verbunden werden. Mit Hilfe von transition-conditions und join-conditions kann der Kontrollfluss festgelegt werden [28].

Geschäftsprozesse werden häufig mit Hilfe von BPEL und Web Services realisiert. BPEL stellt zwar einen de-facto Standard im Hinblick auf die Realisierung da, eignet sich jedoch nicht zum Modellieren auf technisch unabhängigem Abstraktionsniveau und wird deshalb nur selten von Analysten eingesetzt. BPMN oder UML Activity Diagrams eignen sich besser dafür. Auf Grund der zunehmenden Relevanz und um das Business-IT-Gap zu schließen, ist die Transformation von diversen WfM zu BPEL häufig Thema wissenschaftlicher Arbeiten.

Nach dieser Einführung in BPEL folgt nun die formale Definition von BCF. Definition 3 gibt die Bestandteile eines Prozesses in BCF an und definiert somit ein repräsentatives, formales Modell einer blockorientierten Beschreibungssprache, das die Beschreibung der Transformationsstrategien in Abschnitt 3.2.5 erleichtert.

Definition 3: Ein BPEL Control Flow (BCF) ist ein Tupel
$BCF = \{Seq, Flow, Switch, While, Pick, Scope, Basic, Empty, Terminate, Link, de, jc, tc\}$. Die Mengen $Seq, Flow, Switch, While, Pick, Scope, Basic, Empty, Terminate, Link$ sind dabei paarweise disjunkt. $Str = Seq \cup Flow \cup Switch \cup While \cup Pick, \cup Scope$ vereinigt die strukturierten Aktivitäten, während die Basisaktivitäten unter $Bas = Basic \cup Empty \cup Terminate$ zusammengefasst werden. Die Vereinigung der strukturierten und der Basisaktivitäten ergeben alle Aktivitäten des BCF $Act = Str \cup Bas$. Wobei folgende Definitionen gelten:

- Seq, Flow, Switch, While, Pick, Scope, Empty, Terminate, Link ($Link \subseteq Act \times Act$) enthalten jeweils die dazugehörigen BPEL Elemente.
- Basic enthält alle BPEL Basisaktivitäten, ausgenommen Empty und Terminate.
- de: ist eine Dekompositionsrelation und bildet eine strukturierte Aktivität auf die Menge der darin geschachtelten Aktivitäten ab. $de: Str \rightarrow \mathbb{P}(Act) \backslash \emptyset$

- jc: stellt die Aktivierungsbedingung (join condition) der Aktivitäten dar. $jc: Act \rightarrow expr$
- tc: hier wird die Übergangsbedingung (transition condition) der Links definiert. $tc: Link \rightarrow expr$

Definition 4 Aktivierungsbedingung: *Aktivierungsbedingungen* von BPEL-Aktivitäten sind zusammengesetzte Ausdrücke und stellen eine konjunktive, disjunktive oder antivalente (xor) Verknüpfung der Übergangsbedingung der in die Aktivität eingehenden Links dar. Mit AND, OR oder XOR wird, in verkürzter Form, stellvertretend von folgenden Ausdrücken gesprochen:

$AND: jc(x) = tc(y_1, x) \wedge tc(y_2, x) \wedge \ldots \wedge tc(y_n, x)$,
$OR: jc(x) = tc(y_1, x) \vee tc(y_2, x) \vee \ldots \vee tc(y_n, x)$,
$XOR: jc(x) = tc(y_1, x) \oplus tc(y_2, x) \oplus \ldots \oplus tc(y_n, x)$,
wobei succ(x)={y, y, … y} gilt.

Definition 5 Teilbaumfragment: Die Verschachtelung strukturierter Aktivitäten stellt ein baumartiges Konstrukt dar, auf der eine Teilbaumrelation definiert werden kann. Wenn $Strukt \subseteq Act \times Act$ ist, dann ist $(a1, a2) \in Strukt$ genau dann, wenn $a2 \in de(a1)$ gilt und $Strukt^*$ ist die transitive Hülle von $Strukt$ [29].

Nach der formalen Beschreibung der beiden Modellansätze werden verschiedene strukturelle Eigenschaften besprochen, die die Beschreibung der nachfolgenden Strategien erleichtert.

3.2.3 Herausforderungen bei der Transformation

Transformationen können problemlos durchgeführt werden, wenn alle Entitäten des Quellmodells in Konstrukte des Zielmodells umgewandelt werden können. Ist dies nicht der Fall, treten Diskrepanzen auf. Werden nun Workflows eines weniger restriktiven Metamodells in einen Workflow eines restriktiveren Metamodells übergeführt, kann es zu Informationsverlusten kommen. Abhängig vom zu übersetzenden Prozess, ist der resultierende Prozess möglicherweise nicht mehr semantisch äquivalent.
Grundsätzlich können drei Klassen von Diskrepanzen unterschieden werden, die bei der Transformation zwischen unterschiedlichen BPM-Sprachen auftreten können [30]:

(1) Diskrepanzen domänenspezifische Entitäten auszudrücken: Modelle sind meist vereinfachte Darstellungen der realen Welt, und besitzen unterschiedliche Möglichkeiten, Konstrukte der realen Welt und ihre Eigenschaften auszudrücken. Kommen Modelle aus unterschiedlichen Anwendungsbereichen und haben unterschiedliche Ausdruckskraft, geht bei der Transformation des einen Modells in das andere die Semantik verloren. Dies ist jedoch zu vermeiden bzw. zu minimieren.
(2) Diskrepanzen Kontrollflussmuster auszudrücken: Kontrollflussmuster beschreiben immer wiederkehrende Ablaufsituationen in Workflows. Nicht unterstützte Muster im Zielmodell können nicht transformiert werden und verhindern eine vollständige Modelltransformation. In [31] wurden 20 Kontrollflussmuster definiert, die als Vergleichssystem herangezogen werden können, um nicht unterstützte Muster zu identifizieren.
(3) Diskrepanzen in der Repräsentation des Kontrollflusses: Hier entsteht der Unterschied der Modelle dadurch, wie der Kontrollfluss ausgedrückt wird - also entweder graf- oder blockorientiert.

In dieser Arbeit wird angenommen, dass Diskrepanzen, domänenspezifische Entitäten auszudrücken, nicht auftreten. Betrachtet man graforientierte Prozessbeschreibungssprachen fällt auf, dass die Ausdruckkraft durch die Darstellung des Kontrollflusses durch Kanten und Knoten stärker ist. Dadurch sind Transformationen von block- zu graforientierten Modellen meist unproblematischer als

in die entgegengesetzte Richtung. Dies manifestiert sich in den beiden folgenden Beobachtungen, variiert aber abhängig vom betrachteten WfM:
In graforientierten WfM können

- Schleifen mit mehreren Ein- und/oder Ausstiegspunkten konstruiert werden.
- Blöcke konstruiert werden, deren Split- und Join-Konnektor nicht vom gleichen Typ ist. Somit sind auch Kombinationen wie z.B. AND-Split/XOR-Join möglich.

3.2.3.1 Schleifen

Schleifen werden in blockorientierten WfM über strukturierte Aktivitäten dargestellt, die nur einen Ein- (g1) und einen Ausstiegspunkt (g2) definieren (Abb. 9). In BPEL z.B. wird dies über das *while* Element ausgedrückt oder in Asbru über einen zyklischen Plan [1], [32].

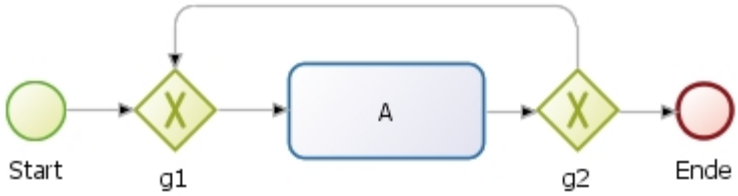


Abb. 9: strukturierte Schleife mit einem Ein- (g1) und Ausstiegspunkt (g2)

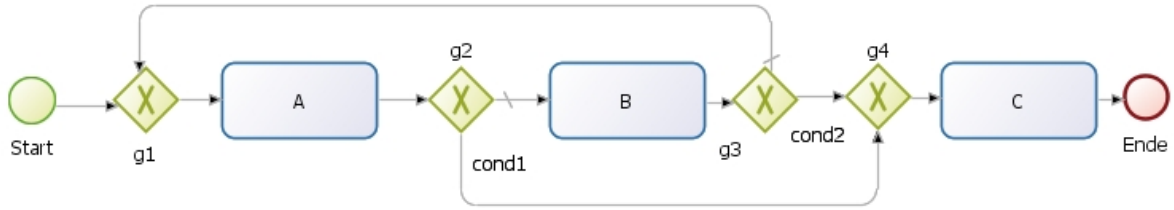


Abb. 10: unstrukturierte Schleife mit einem Einstiegspunkt (g1) und zwei Ausstiegspunkten (g2, g3)

Abb. 10 zeigt eine Schleife mit einem Einstiegspunkt g1 und zwei Ausstiegspunkten g2 und g3. Dies kann weder mit einem While-Element in BPEL, noch mit einem zyklischen Plan in Asbru dargestellt werden.

3.2.3.2 Blöcke

Blöcke, wie z.B. And-Blöcke (Abb. 11) oder XOR-Blöcke, werden in blockorientierten WfM über strukturierte Aktivitäten repräsentiert. Das Verhalten am Synchronisationspunkt (Join-Konnektor) ist meist durch das jeweilige Konstrukt vorgegeben und lässt sich nicht beeinflussen.
Blöcke mit unterschiedlichen Konnektortypen (Abb. 12) lassen sich deshalb meist nicht umsetzen. Der in Abb. 12 gezeigte unstrukturierte Block lässt sich weder in BPEL noch mit Asbru ausdrücken. In BPEL kann entweder ein strukturierter And-Block mit dem *Flow* Element bzw. ein strukturierter XOR-Block mit dem *Switch* Element dargestellt werden [32]. Asbru hat für den strukturierten And-Block den parallelen Plan bzw. den If-Then-Else Plan für den strukturierten XOR-Block zum Ausdrücken zu Verfügung [1].

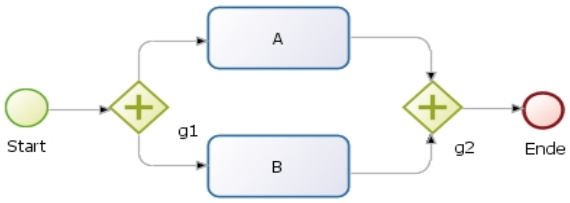


Abb. 11: strukturierter AND-Block

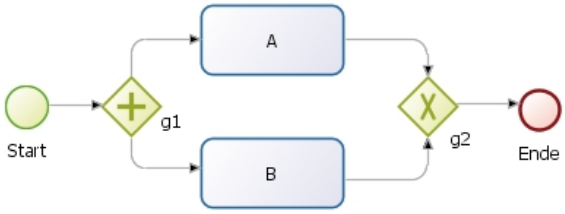


Abb. 12: unstrukturierter Block AND-Verzweigung (g1) u. XOR-Vereinigung (g2)

Wenn man von strukturierten Komponenten spricht, dann sind dies Teile des Prozesses im Quellmodell, die auf Konstrukte im Zielmodell abgebildet werden können. Ob eine Komponente strukturiert ist oder nicht, ist deshalb auch vom Quell- und Zielmodell abhängig und deshalb keine absolute Eigenschaft.

3.2.4 Strukturierte Prozessgrafen und andere Eigenschaften

Prozessgraf-, wie auch BCF-Modelle, werden als Eingaben für den Transformprozess betrachtet. Einige Strategien erfordern restriktivere Einschränkungen des Eingangsmodells, damit die Transformation fehlerfrei durchgeführt werden kann. Die im Zuge der hier vorgestellten Transformationsstrategien relevanten Modelleigenschaften werden wir in diesem Abschnitt erläutern.

Definition 6 zyklische Prozessgrafen: Ein Prozessgraf kann zyklische Kantenfolgen enthalten. D.h. wenn es eine oder mehrere Folgen gerichteter Kanten gibt, die von einem Knoten n zu n zurückführt, dann ist der PG zyklisch [29]. Formal gilt $\exists n \in F \cup C$: $(n, n) \in A^*$. Andernfalls ist der Prozessgraf kreisfrei.

Definition 7 strukturierte BPEL Control Flows (BCFs): Ein BPEL Control Flow ist strukturiert, wenn es kein Link-Element innerhalb des beschriebenen Prozesses gibt [29]. Also, $Link = \emptyset$. Sonst gilt der BCF als unstrukturiert.

Definition 8 strukturierte Prozessgrafen (PG): Ein Prozessgraf ist strukturiert, wenn er mit Hilfe der nachfolgenden Reduktionsregeln zu einer einzigen Aktivität vereinfacht werden kann. Die folgenden Reduktionsregeln beschreiben eine Komponente des PG, und wie sie durch eine Funktion ersetzt werden [29]:

- Sequenz Reduktion: Aktivitäten $f_1, f_2, \dots f_n \in F$, die sequenziell miteinander verbunden sind $(f_1, f_2), (f_2, f_3), \dots, (f_{n-1}, f_n) \in A$, werden durch ein Funktion f_C ersetzt. $F := (F \cup \{f_C\}) \setminus \{f_1, f_2, \dots f_n\}$, $A := (A \cup \{(x, f_C) | x \in pred(f_1)\} \cup \{(f_C, x) | x \in succ(f_n)\} \setminus (f_1, f_2), (f_2, f_3), \dots, (f_{n-1}, f_n)$

- Konnektorpaar Reduktion: Ein Konnektorpaar setzt sich aus dem ersten Konnektor $c_1 \in C$, der den Kontrollfluss verzweigt $|pred(c_1)| = 1$, und dem zweiten Konnektor $c_2 \in C$, der den

Kontrollfluss wieder vereinigt $|succ(c_2)| = 1$, zusammen. Die Menge der direkten Nachfolgeknoten des ersten Konnektors und die Menge der direkten Vorgängerknoten des zweiten Konnektors müssen ident sein $succ(c_1) = pred(c_2) \in F$, und beide Konnektoren müssen denselben Typ (AND, OR, XOR) haben $l(c_1) = l(c_2)$. Ist dies erfüllt, kann das Konnektorpaar und die darin eingeschlossenen Aktivitäten durch die Aktivität fc ersetzt werden. $F \coloneqq (F \cup \{f_C\}) \backslash succ(c_1),\ A \coloneqq (A \cup \{(x, f_C) | x \in pred(c_1)\} \cup \{(f_C, x) | x \in succ(c_2)\}) \backslash (\{(x, c_1) | x \in pred(c_1)\} \cup \{(f_C, c_2) | x \in succ(c_2)\}),\ C \coloneqq C \backslash \{c_1, c_2\}$

- Schleifen Reduktion: ein Schleife beginnt mit einem XOR-Konnektor $c_1 \in C$ (Einstiegspunkt in die Schleife), der den Kontrollfluss vereinigt $|succ(c_1)| = 1$, der über den vorwärtsgerichteten Zweig mit einem XOR-Konnektor$c_2 \in C$ verbunden ist. c_2 (Ausstiegspunkt der Schleife), der den Kontrollfluss splittet $|pred(c_1)| = 1$, ist über den rückführenden Zweig mit dem Einstiegspunkt verbunden. Je nachdem ob im vorwärtsgerichteten, im rückführenden, in beiden oder in keinem Zweig eine Aktivität vorhanden ist, ergeben sich vier Schleifenvarianten. Unabhängig von der Schleifenvariante kann der Ein- u. Ausstiegspunkt und die eingeschlossene(n) Aktivität(en) $succ(c1) \cap pred(c2), succ(c2) \cap pred(c1) \subseteq F$ durch eine Aktivität f_c ersetzt werden. $F \coloneqq (F \cup \{f_C\}) \backslash \big(succ(c1) \cap pred(c2)\big) \cup (succ(c2) \cap pred(c1)),\ A \coloneqq (A \cup \{(x, f_C) | x \in pred(c_1) \setminus succ(c_2)\} \cup \{(f_C, x) | x \in succ(c_2) \backslash pred(c1)\}) \setminus (\{(x, c_1) | x \in pred(c_1)\} \cup \{(c_1, x) | x \in succ(c_1)\} \cup \{(x, c_2) | x \in pred(c_2)\} \cup \{(c_2, x) | x \in succ(c_2)\}),\ C \coloneqq C \backslash \{c_1, c_2\}$

- Startblock Reduktion: Ein Startblock besteht aus XOR-Join Konnektor $c \in C$, dessen Vorgängerknoten der Menge der Startzustände $S = pred(c)$ entsprechen. Ein solcher Block kann durch eine Aktivität fc ersetzt werden. $S \coloneqq \emptyset,\ F \coloneqq F \cup \{f_C\},\ A \coloneqq (A \cup \{(f_c, x) | x \in succ(c)\}) \backslash (\{(x, c) | x \in pred(c)\} \cup \{(c, x) | x \in succ(c)\})$, C:=C\{c}

- Endblock Reduktion: Ein Endblock besteht aus XOR-Split Konnektor $c \in C$, dessen Nachfolgeknoten der Menge der Endzustände $E = succ(c)$ entsprechen. Ein solcher Block kann durch eine Aktivität fc ersetzt werden. $E \coloneqq \emptyset,\ F \coloneqq F \cup \{f_C\},\ A \coloneqq (A \cup \{(x, f_c) | x \in pred(c)\}) \backslash (\{(x, c) | x \in pred(c)\} \cup \{(c, x) | x \in succ(c)\})$, C:=C\{c}

3.2.5 Transformationsstrategien

Nachdem die Definitionen sowohl für die graf- als auch die blockorientierten Workflow-Models und deren strukturellen Eigenschaften vorhanden sind, kann nun zum Kern dieses Abschnitts, den Transformationsstrategien vorgestoßen werden. Es werden hier jedoch nur jene Strategien erläutert die einen graforientierten Workflow in einen blockorientierten umwandeln. Jene Umwandlungsstrategien, die blockorientierte Workflows in graforientierte transformieren, sind für diese Arbeit nicht relevant. Der interessierte Leser findet sie jedoch unter [29].
Um ein besseres Verständnis für die in folgenden Unterabschnitten angeführten Transformationen zu bekommen, werden noch weitere Konzepte erklärt.

Definition 9: Wird eine Funktion $f: A \to B$ auf eine Menge von Elementen angewandt, dann wird die Funktion auf alle Elemente einzeln angewandt und die daraus abgebildeten Elemente zu einer Ergebnismenge zusammengefasst [29]. $f(X) = \cup_{x \in X} f(x), X \subseteq A$

Definition 10 Attributierter Prozessgraf: Ein attributierter Prozessgraf (APG) ist ein Tupel $APG = \{S, E, F, C, l, A, B\}$, wobei S, E, F, C, l wie in Definition 1 definiert sind [29].

- A ist die Menge an Kanten, zwischen den Knoten des APGs. $A = (S \cup F \cup C \cup B) \times (E \cup F \cup C \cup B)$
- B ist die Menge aller PG-Knoten, die ein Attribut mit der BCF-Übersetzung enthalten. Somit ist ein Element in B ein bereits übersetzter Knoten des Prozessgrafs.

Definition 11 APG Knoten Mapping: M stellt eine Mapping-Funktion dar, die einen Knoten des APGs in BCF-Aktivitäten übersetzt. $M: S \cup E \cup F \cup C \cup B \rightarrow Basic \cup Empty \cup Terminate \cup B$

$$M(x) = \begin{cases} Empty(x), if\ x \in C; \\ Basic(x), if\ x \in F \cup S \\ Terminate(x), if\ x \in E; \\ x, if x \in B \end{cases}$$

Die grundsätzliche Idee ist Startzustände und Aktivitäten auf Basic-Elemente, Konnektoren auf Empty-Elemente und Endzustände auf Terminate-Elemente abzubilden [29].

Die in den folgenden Unterabschnitten vorgestellten Übersetzungsstrategien nützen für die Modelltransformation unterschiedliche Mechanismen des Zielmodells. Als Zielmodell dient hier BPEL, da es einerseits repräsentativ für das blockorientierte Paradigma ist, andererseits sind Transformationen von verschiedenen graforientierten Modellen zu BPEL ein wissenschaftlich ausführlich erörtertes Thema (siehe [4], [5], [27], [32], um nur einige Quellen zu nennen).

In dieser Arbeit werden 5 Strategien diskutiert, manche davon kombinieren andere Strategien zu einer eigenständigen Strategie und manche bauen aufeinander auf und verfeinern den ursprünglichen Ansatz.
Folgende Ansätze werden erörtert.

(1) Element-Preservation: Hier werden Aktivitäten entsprechend einer Mapping-Funktion (Definition 11) in Basic-Aktivitäten umgewandelt. Der Kontrollfluss wird dann mit Hilfe von BPEL-Links definiert.
(2) Element-Minimization: verfeinert das Ausgangsmodell der Element-Preservation Strategie, indem die Empty-Elemente eliminiert werden, die die Konnektoren darstellen.
(3) Structure-Identifikation: Der zu übersetzende Eingangsgraf wird in Komponenten geteilt, die in strukturiert BPEL-Aktivitäten übersetzt werden können. Dies eignet sich jedoch nur für einen strukturierten Prozessgrafen, da unstrukturierte Komponenten nicht übersetzt werden können.
(4) Structure-Maximization: Strukturierte Komponenten des Eingangsgrafs werden mit der Structure-Identifikation Strategie transformiert, während unstrukturierte Komponenten mit Hilfe der Element-Preservation bzw. -Minimization Strategie übersetzt werden.
(5) Event-Condition-Action-Rules: Der Eingangsgraf wird mit der Structure-Identifikation Strategie so weit wie möglich transformiert. Aktivitäten innerhalb unstrukturierter Komponenten werden in BPEL-Event-Handler umgewandelt, die den entsprechenden Event der nachfolgenden Aktivität (Event-Handlers) aufrufen.

Die folgenden Unterabschnitte enthalten die detaillierten Beschreibungen der oben überblicksmäßig angeführten Strategien.

3.2.5.1 Strategie 1: Element-Preservation

BPEL ist keine rein blockorientierte Beschreibungssprache, sondern besitzt auch grafbasierte Konzepte, nämlich Links. Mit Links können logische Abhängigkeiten modelliert werden, vergleichbar mit Kanten in Prozessgrafen. BPEL-Links können nur innerhalb eines BPEL Flow-Elements eingesetzt werden, und die damit definierten Kantenfolgen müssen kreisfrei sein. Eine weitere Einschränkungen ist, dass Links nie aus einer bzw. in eine Schleife führen dürfen [29].

Der generelle Gedanke dieser Strategie ist, dass jeder Knoten des PG, entsprechend der Mapping-Funktion (Definition 11), in eine BPEL-Basisaktivität übersetzt wird, und die Kanten durch entsprechende Links dargestellt werden [29].
Der entstandene BPEL-Prozess besitzt genau ein Flow-Element. In diesem Flow-Element sind die Links und die übersetzten Aktivitäten des PG enthalten. In den Aktivitäten sind source- bzw. target-Elemente definiert, die die Quell- bzw. Zielaktivität des Links darstellen.
Über alle target-Elemente einer Aktivität kann eine Aktivierungsbedingung angegeben werden. Und für jeden Link kann, über das source-Element, eine Übergangsbedingung spezifiziert werden.

Diese Strategie kann nur auf kreisfreie PGs angewendet werden, die keine zyklischen Pfade spezifizieren. Der Vorteil liegt im geringen Implementierungsaufwand, und dass für jeden Knoten im PG eine Aktivität im BCF gebildet wird. Da auch ein Empty-Element für jeden Konnektor im PG gebildet wird, enthält der BCF mehr Aktivitäten als eigentlich nötig wären, was ein klarer Nachteil dieser Strategie ist [29].

3.2.5.2 Strategie 2: Element-Minimization

Diese Strategie setzt bei dem Nachteil an, dass in Strategie 1 Konnektoren explizit in Empty-Elemente übersetzt werden und vereinfacht das Ergebnis aus Strategie 1.
D.h. die Empty-Elemente des in Strategie 1 entstandenen BCF werden eliminiert, indem die Links um die Empty-Elemente herumgeleitet werden und die Aktivierungsbedingung der nachfolgenden Aktivitäten angepasst werden. Danach werden die Empty-Elemente und die darin hinein- und hinausführenden Links entfernt [29].

Zuerst wird der PG mit Strategie 1 übersetzt. Im Anschluss wird über alle Empty-Elemente $x \in X$ iteriert, die keinen anderen Konnektor als direkten Vorgängerknoten im PG haben.
Nun wird ein Link von jeder eingangsverlinkten Aktivität von x zu jeder ausgangsverlinkten Aktivität gebildet. Die angepasste Aktivierungsbedingung $jc'(y)$ der ausgangsverlinkten Aktivität $y \in Y$ wird in der Form angepasst, dass die ursprüngliche Aktivierungsbedingung von y, $jc(y)$, mit der Aktivierungsbedingung des Empty-Elements x, $jc(x)$, konjunktiv verknüpft wird. $jc'(y) = jc(y) \wedge jc(x)$. Im Anschluss werden alle Empty-Elemente x und die mit ihnen verbundenen Links entfernt. $M^{-1}(Empty) \cap C = \emptyset,\ Links = (Links \cup \{(y1, y2) | y1 \in pred(x) \wedge y2 \in succ(x)\}) \setminus (\{(x, y) | y \in succ(x)\} \cup \{(y, x) | y \in pred(x)\})$

Der Vorteil der Element-Minimization Strategie ist, dass der generierte BCF nur noch Elemente enthält, die zum Ausdrücken des BPEL-Prozesses notwendig sind.
Der Nachteil liegt darin, dass die korrespondierenden Knoten in PG und BCF nicht mehr so intuitiv erkennbar sind, wie z.B. in Strategie1, da nicht mehr alle ursprünglichen Knoten vorhanden sind [29].

3.2.5.3 Strategie 3: Structure-Identification

Bei der Structure-Identification Strategie werden strukturierte Teile des Prozessgrafen identifiziert, um sie durch die Reduktionsregeln (Definition 8) zu vereinfachen. Nur werden die strukturierten Komponenten nicht durch eine Funktion, sondern durch einen Knoten eines attributierten Prozessgrafen ersetzt, dessen Attribut die BCF-Übersetzung der Komponente enthält [29].
Handelt es sich um einen strukturierten PG, kann mit dieser Strategie eine vollständige Reduktion erfolgen. Somit wird der PG in einen APG umgewandelt, dessen Serialisierung der Attribute den BCF-Prozess enthält. Zusammengefasst bedeutet das, dass der PG ausschließlich mit strukturierten BCF-Aktivitäten übersetzt wird.

Daraus kann gleich gefolgert werden, dass diese Strategie nur für strukturierte Prozessgrafen sinnvoll ist.

Definition 12 Übersetzungsregeln: Entsprechend der angewandten Reduktionsregeln werden folgende strukturierte BCF-Aktivitäten zur Übersetzung verwendet:

- Sequenz: Sequenzen eines PG wird mit einem Sequence-Element dargestellt, dessen Subaktivität den Knoten in der Reihenfolge des PG entsprechen.

- AND-Block: Ein AND-Block wird in ein Flow-Element umgewandelt, wobei jeder Zweig einer Subaktivität des Flow-Elements entspricht.

- OR-Block: Dafür wird ein Empty-Element in ein Flow-Element verschachtelt, und jeder Zweig des OR-Blocks wird als Subaktivität des Flow-Elements dargestellt. Zusätzlich werden Links, ausgehend vom Empty-Element zu den Subaktivitäten, erstellt, deren Übergangsbedingungen mit den Kantenbedingungen des OR-Splits übereinstimmen.

- XOR-Block: Ein XOR-Block wird in ein Switch-Element umgewandelt, wobei jeder Zweig einer Subaktivität des Switch-Elements entspricht.

- Schleifen: Die Übersetzung wird, entsprechend der Schleifenvariante, mit While-, Sequenc-, oder Empty-Elementen umgesetzt. Genauere Informtionen siehe [29].

- Start-Block: Der Start-Block wird mit einem Pick-Element umgesetzt, das für jeden Zweig ein Empty-Element als Subaktivität besitzt.

- End-Block: Wird entsprechend dem Eingangsmodell mit einem AND-, OR, oder XOR-Block, gefolgt von einem Terminate-Element für jeden Zweig, umgesetzt.

Bei der Übersetzung geht man wie folgt vor:
Zuerst wird ein attributierter Prozessgraf entsprechend dem Eingangsgrafen erstellt $APG = \{S, E, F, C, l, A, \emptyset\}$ [29]. Danach werden die Reduktionsregeln solange angewandt, bis der Graf auf einen Knoten reduziert ist.
Dabei wird bei jeder Iteration, nach einer passenden Komponente APG' gesucht, die durch Anwendung einer Reduktionregel (Definition 8) ersetzt werden kann. APG' wird mit Hilfe der Regeln in Definition 12 übersetzt, mit ihrer Übersetzung ersetzt und somit APG reduziert [29].

Der Vorteil liegt darin, dass der entstandene Prozess nur aus strukturierten Aktivitäten besteht und dadurch lesbarer ist. Als Nachteil kann man anführen, dass die Zuordnung zu den ursprünglichen Elementen des PG nicht mehr so offensichtlich erkennbar ist [29].

3.2.5.4 Strategie 4: Structure-Maximization

Die Idee dieser Übersetzung ist es, die Reduktionsregeln aus Structure-Identification Strategie so oft wie möglich anzuwenden. Erst wenn dies nicht mehr möglich ist, da unstrukturierte Teilgrafen nicht reduziert werden können, werden diese mit der Element-Preservation bzw. Element-Minimization Strategie weiter übersetzt [29].

Der Vorteil dieser Strategie ist, dass sie auch auf unstrukturierte Prozessgrafen, abgesehen von unstrukturierten Schleifen, angewandt werden kann. Da diese Strategie eigentlich zwei o.a. Strategien zur Umsetzung verwendet, ist nachteilig der erhöhte Implementierungsaufwand anzuführen [29].

3.2.5.5 Strategie 5: Event-Condition-Action-Rules

Hier ist die Idee ähnlich wie bei der vorangegangenen Strategie 4. D.h. es wird versucht, möglichst viele Teile des Eingangsgrafen mit Hilfe des Structure-Indentification Verfahrens zu übersetzen. Die unstrukturierten Teile des PG, die nicht mit Definition 8 reduziert werden können, werden jedoch mit Event-Condition-Action (ECA) Regeln im BCF-Prozess abgebildet. In BPEL können ECA Regeln mit Event-Handler implementiert werden [29].

Ein Event-Handler ist ein Konstrukt, das einen Event-Typ und eine Startbedingung mit einer Aktivität assoziiert. Tritt ein Event des assoziierten Typs auf und die Startbedingung des Event-Handlers ist erfüllt, wird die spezifizierte Aktivität ausgeführt. Durch Aufrufen des entsprechenden Event-Typs innerhalb des Prozesses können Event-Handler gestartet werden, und somit können auch unstrukturierte Teilgrafen realisiert werden [29].

Formal muss die Definition des BCF um folgende Elemente erweitert werden [29]:

- $Handler$ stellt die Menge der Event-Handler im Prozess dar und $handler(prc, Act)$ assoziiert die Menge der Startbedingungen mit einer Aktivität.
- $invoke(s)$ ist jene Aktivität, die einen Event-Typ s aufruft und so einen Event-Handler startet. $invoke(s) \in Invoke$
- $receive(s)$ ist jene Aktivität, die auf den Aufruf eines Event-Typs wartet und, bei dessen Empfang, die nachfolgenden Aktivitäten ausführt.

Zuerst wird der Prozessgraf mittels Structure-Identification so weit wie möglich reduziert. Danach wird jene Komponente, die beim Starten des Prozess als erstes ausgeführt wird herangezogen und folgende Sequenz gebildet:

$$receive(startProcess);$$
$$M\big(getFirstComponent(PG)\big);$$
$$invoke(firstCompleted)$$

Danach wird für jede weitere Komponente f ein Event-Handler erzeugt.

$$handler(getPreconditionSet(f), M(f); invoke(< i^4 > Completed)$$

Nachdem die erste Basisaktivität beendet wurde, ruft die erste Sequenz jene Event-Handler auf, die nach der ersten Aktivität ausgeführt werden und diese die darauf folgenden.

Der Vorteil dieser Strategie ist, dass es hier keine Einschränkungen gibt, und jeder beliebige PG transformiert werden kann.
Ein Nachteil ist hier, dass der erzeugte BPEL-Prozess komplizierter aufgebaut und schwerer zu lesen ist [29].

Die in Abschnitt 3.2.5 beschriebenen Transformationsstrategien haben ihren Schwerpunkt darauf, typische Probleme, die bei der Übersetzung von graf- zu blockorientierten Prozessmodellen auftreten, zu umgehen. Andere allgemeinere Ansätze, um Modelle zu transformieren werden im folgenden Abschnitt besprochen.

[4] i steht hier für die Nummer der jeweilige Komponente

3.2.6 Weitere Transformationsansätze

In diesem Abschnitt soll ein Überblick über verschiedene Ansätze gegeben werden, Modelltransformationen durchzuführen.
Geschäftsmodelle sind inzwischen integraler Bestandteil des Arbeitsalltags, und Unternehmen besitzen oft eine Vielzahl von modellierten Prozessen. Diese Modelle werden mit Geschäftspartnern ausgetauscht oder müssen durch Firmenfusionen in bestehende Systeme integriert werden. Durch diese und ähnliche Szenarien steigt der Bedarf an, Modelle zu transformieren [33].

Modelltransformation ist eigentlich ein Sammelbegriff, unter dem man ganz allgemein Transformationen von *Modell-zu-Modell*, *Modell-zu-Text* oder *Text-zu-Modell* verstehen kann. In dieser Arbeit ist immer die Transformation von Modell-zu-Modell gemeint, wenn von Modelltransformation gesprochen wird. Wird ein Modell in ein Modell derselben Abstraktionsstufe übersetzt, spricht man von einer horizontalen Transformation. Hat das Zielmodell eine andere Abstraktionsstufe, spricht man von einer vertikalen Transformation [34].

Bei Modell-zu-Modell Transformationen wird das Quellmodell der Transformation, welches konform zum Quellmetamodell modelliert wurde, durch Ausführen der Transformationsdefinition durch den Transformations-Engine in das Zielmodell der Transformation umgewandelt, welches konform zum Zielmetamodell ist (Abb. 13) [21].

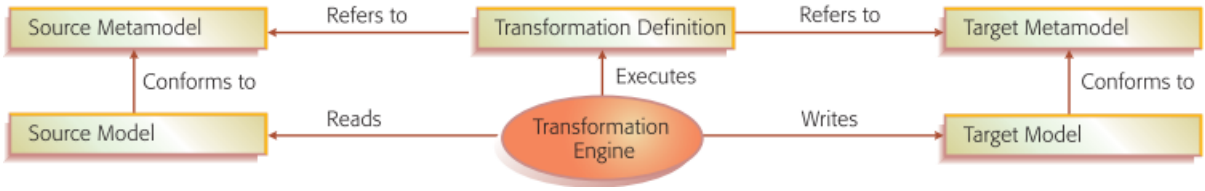


Abb. 13: Illustration d. Zusammenhänge zwischen Quell-/Ziel(meta)modell und d. Transformationsdefinition [35]

Es gibt verschiedene Ansätze diese Transformationsdefinition zu spezifizieren [35]:

- Direct Manipulation Approach: Hier wird das Quellmodell in eine interne Repräsentation umgewandelt, die über ein API manipuliert werden kann. Die Modelltransformation wird meist mit Hilfe einer imperativen Programmiersprache implementiert, die das API nützt.
- Structure-driven Approach: Dieser Ansatz charakterisiert sich dadurch, dass die Transformation zwei Phasen aufweist. Die erste Phase baut die Struktur des Zielmodells auf, während in der zweiten Phase die Attribute und Verbindungsreferenzen erstellt werden.
- Operational Approach: Er ähnelt dem Direct Manipulation Approach, allerdings wird dieser Ansatz um Formalismen erweitert, um auch komplexere Programme zu spezifizieren. Dadurch wird keine zusätzliche allgemeine Programmiersprache mehr benötigt. Dies könnte z.B. eine Abfragesprache mit imperativen Elementen, wie OCL [36], sein, die ein eigenständiges Programmiersystem bildet.
- Template-based Approach: Hier werden Modell Templates verwendet, die Teilmodelle mit eingebettetem Meta-Code darstellen. Der Meta-Code wird bei der Transformation ausgewertet und die Ergebnisse davon stattdessen eingefügt. Dadurch können die dynamischen Anteile mit einem statischen Teilmodell kombiniert werden.
- Graph-transformation-based Approach: Der Ansatz beruht auf Graftransformationen und kann auf typisierte, attributierte, beschriftete Grafen angewandt werden. Eine Transformationsregel besteht aus einer linken (LHS) und einer rechten Seite (RHS) mit jeweils einem Grafmuster. Die LHS beschreibt das Muster, das im Graf vorgefunden werden muss, um die Transformationsregel zu aktivieren. Die RHS beschreibt das Muster, durch das die LHS ersetzt wird, um den gewünschten Zielgrafen zu erhalten.

- Hybride Approach: Bei diesem Ansatz werden beliebige oben genannte Ansätze miteinander kombiniert, um die Transformationsdefinition zu spezifizieren.

Diese Aufzählung soll nur einen Überblick über mögliche Wege geben, wie Transformationsregeln definiert werden können. Für weitere Details sei auf [35] verwiesen.
Alle hier angeführten Ansätze haben jedoch gemeinsam, dass sie das Quell- und Zielmetamodell nützen, um die Transformationsdefinition zu spezifizieren. Dadurch ist Wissen über den Aufbau der Metamodelle eine Voraussetzung für den Transformationsdesigner.

Der im nächsten Abschnitt gezeigte Ansatz unterscheidet sich grundsätzlich von den bisher gezeigten und arbeitet mit einer anderen, benutzerfreundlichen Art der Transformationsspezifikation, mit der auch Personen arbeiten können, die mit den Metamodellen nicht vertraut sind.

3.2.6.1 Model Transformation by-Example (MTBE)

Modelle werden in unterschiedlicher Art dargestellt. Der Designer/Modellierer, der die Modelle meist mit einem grafischen Modellierungswerkzeug erstellt, arbeitet mit der grafischen Notation (auch *konkrete Syntax (concrete syntax, CS)* genannt) des Modells, in der jeder Entität des Metamodells eine grafisches Symbol zugeordnet ist. In einem UML Klassendiagramm z.B. werden Klassen als Rechtecke und Relationen zu anderen Klassen als Kanten zwischen den Klassensymbolen dargestellt [37].

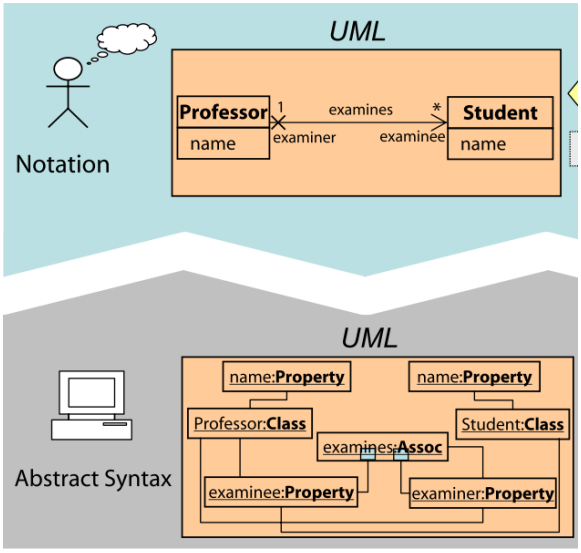


Abb. 14: UML Klassendiagramm oben in konkreter, unten in abstrakter Syntax [37]

Intern wird das Modell jedoch entsprechend dem Metamodell in *abstrakter Syntax (abstract syntax, AS,* Abb. 14 *oben)* abgebildet. Die oben in Abschnitt 3.2.6 angeführten Spezifizierungsmethoden arbeiten auf Ebene der abstrakten Syntax bzw. des Metamodells, indem Transformationsregeln unter Verwendung der Metamodellelemente definiert werden.

Dadurch ergeben sich zwei problematische Aspekte [37]:

(1) Die Art wie dem Benützer das Modell repräsentiert wird und wie er es sieht, unterscheidet sich von der elektronischen Repräsentationsform (Metamodell).
(2) Die Intention des Metamodell liegt darin, eine Modellierungssprache zu definieren, die effizient implementiert werden kann. Daher werden nicht alle Konzepte der Modellierungssprache explizit im Metamodell ausgedrückt, sondern verbergen sich in Attributen oder Relationen und müssen später rekonstruiert werden. Man spricht hier auch von versteckten Konzepten (*concept hiding)*.

Beide Aspekte erschweren die Spezifikation von Modelltransformationsregeln.

Die Idee von *Model Transformation by-Exampel (MTBE)* ist nun, ein Mapping auf der Ebene der konkreten Syntax zwischen zwei Modellen zu definieren und daraus Transformationsregeln auf Metamodellebene zu generieren.

Unter der Voraussetzung, dass Relationen für jede Modellierungssprache vorhanden sind, die Elemente der abstrakten in Elemente der konkreten Syntax abbilden, kann MTBE mit Hilfe der folgenden drei Schritte definiert werden [37]:

(1) Als erstes muss der Benutzer ein oder mehrere Quell- und Zielmodell (e) modellieren die dasselbe Problem beschreiben. Die definierten Modelle sollten dabei möglichst alle Konzepte der Modellierungssprache abdecken.

(2) Danach müssen die korrespondierenden Elemente in den Modellen verbunden und so ein Mapping zwischen den Modellelementen des Quell- und Zielmodells definiert werden. Dafür stehen nicht nur 1:1, sondern auch komplexere Mapping-Operatoren zu Verfügung [34].

(3) Im letzten Schritt wird das Mapping durch einen Modelltransformationsgenerator (Model Transformation Generator, MTGen) analysiert und so eine Transformationsdefinition generiert, die auf bewährten metamodellbasierten Transformationstechnologien basiert. Durch das Mapping etwaige ergebende Zweideutigkeiten, müssen durch den Transformationsdesigner aufgelöst werden.

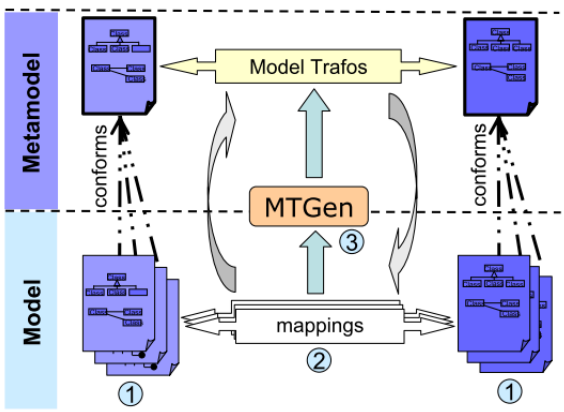


Abb. 15: MTBE Spezifizierungsschritte im Kontext der Architekturschichte [21]

Abb. 15 zeigt die einzelnen Spezifikationsschritte im Kontext der Modellierungsarchitekturschichten. Die für die Spezifikation der von MTBE benötigten Modelle und das benutzerdefinierten Mapping zwischen den Modellelementen befinden sich auch Modellebene (M1). Die von MTGen generierten Transformationsregeln (Model Trafos) befinden sich auch Metamodellebene (M2).

Da bei diesem Ansatz Transformationen auf der Modellebene (M1) definiert werden, muss man nicht mit im Metamodell versteckten Konzepten umgehen. Außerdem wird mit der graphischen Notation des Modells gearbeitet, mit der der Benützer bereits vertraut ist. Dadurch soll es einfach möglich sein Transformationsregeln zu erstellen, auch ohne mit dem Metamodell der Modellierungssprache vertraut zu sein [37].

Die bisher angeführten Transformationen beruhen darauf, das Elemente im Quellmodell in ein oder in mehrere Elemente des Zielmodells umgesetzt werden, und auch die Transformationsregeln werden entsprechend spezifiziert. Für Anwendungsfälle wo eine 1:1 bzw. 1:n Transformation nur schwer anwendbar ist, stellt der im nächsten Abschnitt vorgestellte Ansatz eine Alternative dar.

3.2.6.2 Musterbasierte Modelltransformation

Aktuelle Technologien für Modelltransformationen, wie QVT [19] oder ATL [20], beruhen auf der Transformation von Metamodellelementen. Dadurch sind Transformationen schwierig zu definieren, die nicht darauf basieren, ein Element im Quellmodell in ein oder mehrere Element(e) im Zielmodell abzubilden.
Diese komplexen Transformationen benötigen oft Informationen, die erst durch komplexe Abfragen zur Verfügung stehen, die jedoch fehleranfällig, schwer zu warten und kaum wiederverwendbar sind [33].

Abb. 16 zeigt einen Prozess in ADONIS [16] Notation. Die Besonderheit an ADONIS ist, dass hier kein explizites Element vorhanden ist, um die Vereinigung einer alternativen (XOR) Verzweigung

auszudrücken. Somit vereinigt das Parallel Join Element (in Abb. 16 rot eingekreist) nicht nur die parallele, sondern auch implizit die alternative Verzweigung.
Wird dieser Beispielprozess in ein Modell transformiert, welches ein explizites Element dafür vorsieht, muss erst durch eine komplexe Abfrage ermittelt werden, ob im Pfad der Vorgängerelemente eine alternative Verzweigung vorhanden ist. Da diese Abfragen fehleranfällig und kaum wiederzuverwenden sind, ist dieser Lösungsansatz unbefriedigend.

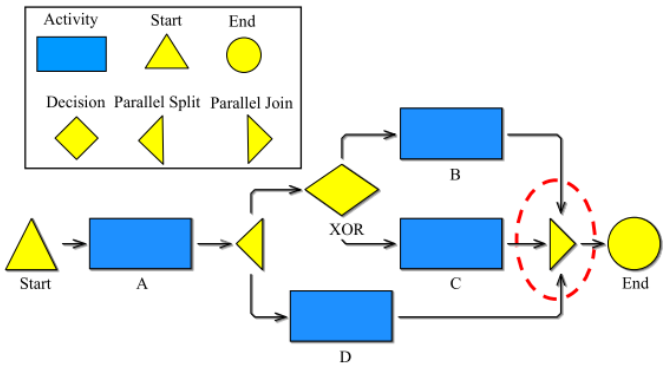


Abb. 16: Prozess in ADONIS Notation; Parallel Join Element (roter Kreis) schließt parallele u. alternative Verzweigung [33]

In [33] wird eine musterbasierte Herangehensweise erläutert, die Übersetzungsmuster (transformation pattern) für den Übersetzungsprozess nutzt. D.h. der zu übersetzenden Prozess wird in einer ersten Phase analysiert und durch eine Abfolge bzw. Verschachtelung von Übersetzungsmustern beschrieben. Diese Abfolge von Übersetzungsmustern wird danach in einer zweiten Phase in einen Zielprozess umgesetzt. Dabei werden die Transformationsmuster mit Elementen des Zielmodells synthetisiert und bauen so Muster für Muster den Zielprozess auf.

Dadurch, dass der zu transformierende Prozess durch Muster auf einem höheren Abstraktionslevel beschrieben wird, soll die Definition der Transformation erleichtert und die Wiederverwendung verbessert werden.

Für unseren Beispielprozess aus Abb. 16 bedeutet das, dass er ein Sequential-Path Pattern aus A und einem Parallel Pattern darstellt. Dieses Parallel Pattern aus einem Alternative Pattern und D und das Alternative Pattern aus B und C besteht.
Diese musterbasierte Zwischendarstellung des Quellprozesses kann im Zielmodell synthetisiert werden und baut so den Zielprozess auf. Dies funktioniert natürlich nur, wenn das Zielmodell diese Muster auch unterstützt [33].
Dies ist ein vielversprechender Transformationsansatz und eignet sich auch für die Transformation von graf- zu blockorientierten Prozessmodellen.

3.3 Strukturieren unstrukturierter Schleifen

Die Definition, was unter einer strukturierten Komponente zu verstehen ist, ist unterschiedlich und abhängig von den verwendeten Modellierungssprachen. Die in Definition 8 festgelegten Formen von Strukturiertheit beziehen sich auf Konstrukte in BPEL, dem Zielmodell.
Allgemein versteht man unter strukturierten Komponenten jene Muster im Quellmodell, die ein äquivalentes Konstrukt im Zielmodell haben und in dieses auch übersetzt werden können.

Z.B. sind in Definition 8 nur Split-Konnektoren mit Join-Konnektoren vom gleichen Typ (AND, XOR, OR) kombinierbar, damit dieses Konnektorpaar eine strukturierte Komponente bildet. Das hat den Grund, dass BPEL nur diese Konnektorpaare unterstützt. Würde BPEL auch das „multiple instances without synchronisation" Muster unterstützen, dann könnte auch ein And-Split- mit einem XOR-Join-Konnektor kombiniert werden und als strukturierte Einheit gelten.

Daraus kann gefolgert werden, dass unstrukturierte Komponenten kein semantisch äquivalentes syntaktisches Gegenstück im Zielmodell haben und deshalb nicht übersetzt werden können.
Es gibt jedoch auch quasi-strukturierte Komponenten [38]. Das sind Komponenten, die zwar unstrukturiert sind, jedoch durch Reorganisieren in eine strukturierte Komponente umgewandelt werden können, z.B. durch Hinzufügen eines Konnektors, etc..

Meist können Schleifen nur in blockorientierte Modellierungssprache übersetzt werden, wenn die Schleife genau einen Eingangspunkt und genau einen Ausgangspunkt hat. Schleifen die nun mehrere Ein- oder Ausgangspunkte haben und im Schleifenkörper keinen And-Konnektor (bzw. Or-Konnektor) enthalten, können in strukturierte Schleifen umgewandelt werden [32]. Meist müssen dafür Konnektoren hinzugefügt und/oder Schleifenbedingungen adaptiert werden [39].

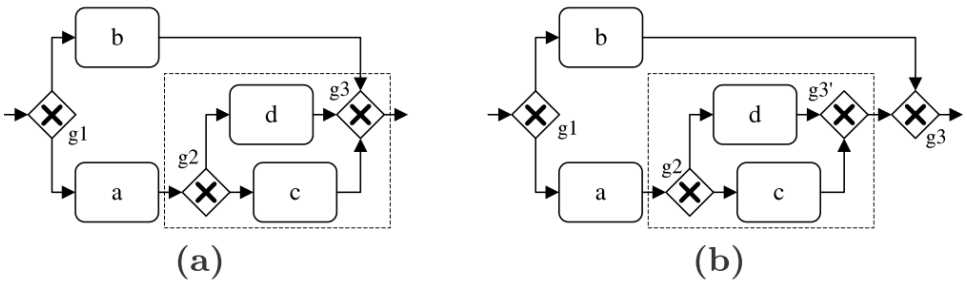


Abb. 17: a) unstrukturierte Komponente, da schließender Konnektor zu g1 fehlt b) reorganisierte Variante davon [38]

Abb. 17a zeigt eine Komponente, die deshalb unstrukturiert ist, da der Split-Konnektor g1 keinen entsprechenden Join-Konnektor besitzt. Durch Einfügen eines zweiten Join-Konnektors g3' wird der Prozess durch Reorganisation strukturiert (siehe Abb. 17b).

Im folgenden Unterabschnitt wird ein Verfahren beschrieben, mit dem unstrukturierte Schleifen, mit gewissen Einschränkungen, in ein blockorientiertes Modell übergeführt wird, welches nur strukturierte Schleifen enthält.

3.3.1 Reorganisation von Schleifen

Wie o.a. können unstrukturierte Schleifen, in deren Schleifenkörper nur XOR-Splits und XOR-Joins verwendet werden, in strukturierte Schleifen umgewandelt werden [32].

In [39] wird ein Verfahren vorgestellt, das Transformationen von Prozessmodellinstanzen mit unstrukturierten Komponenten in blockorientierte Prozessmodelle durchführt, welche nur strukturierte Schleifen unterstützen.

Zur Demonstration des Verfahrens wird ein Beispiel vorgestellt, das einen Bestellprozess darstellt (siehe Abb. 18). In diesem Bestellprozess kann ein Produkt ausgewählt, konfiguriert und in die Bestellung aufgenommen werden. Die Konfiguration des Produkts kann übersprungen werden oder erst nach dem Hinzufügen zur Bestellung erfolgen. Nachdem ein Produkt zur Bestellung hinzugefügt wurde, können weitere Produkte ausgewählt, konfiguriert und iterativ hinzugefügt werden. Die Bestellung kann anschließend abgeschickt oder verworfen werden. Des Weiteren können nach verschickter Bestellung die Konfigurationen eingesehen werden.
Der Prozess in Abb. 18 zeigt eine Schleife A, B, C, D mit mehreren Einstiegspunkten (Start, E und F) und mehrere Ausstiegspunkte (E und F), wodurch sich unstrukturierte Zyklen charakterisieren.

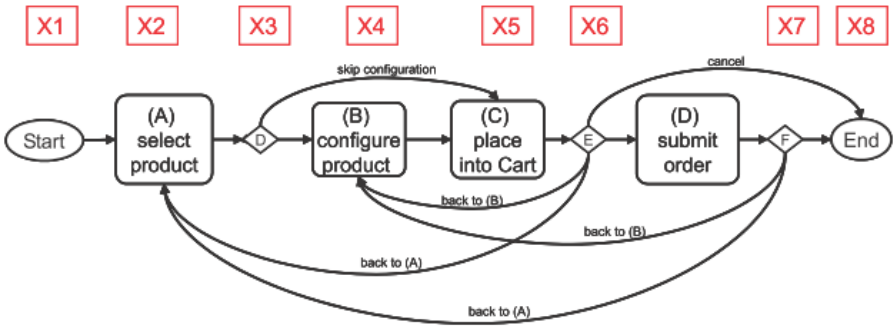


Abb. 18: Bestellprozess mit unstrukturierter Schleife u. dazugehörige Fortführungsvariablen (rote Rechtecke über den Komponenten) [39]

Das Verfahren basiert auf der Fortführungssemantik (Continuation semantics) und teilt den definierten Prozess in einen ausgeführten, in den aktuellen und den auszuführenden Prozessabschnitt oder Zustand [39]. Wenn (A) der aktuelle Zustand im Prozessgraf ist, dann ist Start der ausgeführte Zustand und (B) oder (C) der zukünftige Zustand.

Das Verfahren teilt sich in drei Teilschritte auf. Im ersten Schritt werden den einzelnen Komponenten Variablen Fortführungsvariablen (continuation variables) zugeteilt. Im zweiten Schritt wird für jede Fortführungsvariable eine Gleichung gebildet. Diese Gleichungen werden im dritten Schritt mit Reduktionsregeln solange vereinfacht, bis eine einzelne Gleichung über bleibt, die die strukturierte Schleife darstellt. Die nachfolgenden Sektionen beschreiben die einzelnen Schritte im Detail.

3.3.1.1 Schritt 1: Fortführungsvariablen

Im beschriebenen Verfahren werden zuerst Fortführungsvariablen (continuation variables) definiert. Dem Startknoten, dem Endknoten und jedem anderen Knoten mit mehr als einer eingehenden oder ausgehenden Kante wird eine eigene Variable zugeordnet.
In Abb. 18 werden diese Variablen X1–X8 oberhalb des assoziierten Knotens dargestellt. Somit erhalten alle Aktivitäten und Konnektoren eine Variable zugeteilt, mit Ausnahme von Aktivität D. Da Aktivität D nur jeweils eine aus- und eingehende Kante besitzt, wird ihr keine Variable zugewiesen [39].

3.3.1.2 Schritt 2: Fortführungsgleichungen

Nachdem jeder geeignete Knoten eine Variable besitzt, wird als zweiter Schritt für jede Variable eine Fortführungsgleichung (continuation equation) aufgestellt.

Auf der linken Seite (left-hand side, LHS) der Gleichung steht der Variablenname z.B. X1.
Die rechte Seite der Gleichung (right-hand side, RHS) enthält entweder eine andere Variable oder Aktivität. Sie drückt die Möglichkeiten aus, wie der Prozess im Anschluss an die Variable fortgeführt wird [39].
Eine Aktivität wird mit *invoke A* notiert und kann mit dem Sequenzoperator „ ; “ zu einer linearen Abfolge von Aktivitäten verknüpft werden.
Ein XOR-Split wird durch einen If-Ausdruck der Form *if <Cond> then x* dargestellt. Die If-Bedingung <Cond> wird durch eine neue Variable ausgedrückt, deren Name sich aus dem Namen des Anfangs- und des Endknotens zusammensetzt z.B. AB, AC, etc.

$$RHS = \begin{cases} Start; nextvariable(x); & iff\ x\ ist\ ein\ Startknoten \\ End; & iff\ x\ ist\ ein\ Endknoten \\ invoke\ x; nextvariable(x); & iff\ x\ ist\ eine\ Aktivität \\ für\ alle\ ausgeheden\ Kante(x,y)\ und\ a = pred(x)\ \{ & \\ \quad if\ cond(a,y) then\ y; & \\ \} & iff\ x\ ist\ ein\ XOR-Split\ Konnektor \end{cases}$$

Für unser Beispiel ergeben sich folgenden Gleichungen:

(1) x_1 = Start; x_2;
(2) x_2 = invoke A; x_3;
(3) x_3 = if AB then x_4;
if AC then x_5;
(4) x_4 = invoke B; x_5
(5) x_5 = invoke C; x_6
(6) x_6 = if CD then invoke D; x_7
endif;
if CEnd then x_8;
if CA then x_2;
if CB then x_4;
(7) x_7 = if DB then x_4;
if DA then x_2;
if DEnd then x_8;
(8) x_8 = End ;

Abb. 19 : Initiale Gleichungen d. Bestellprozesses [39]

3.3.1.3 Schritt 3: Transformationen

Im dritten Schritt werden die Gleichungen mit Hilfe der nachfolgenden Transformationsregeln vereinfacht. D.h. es wird nach anwendbaren Regeln gesucht und die passendste davon angewandt. Dies wird so lange wiederholt, bis nur mehr eine Gleichung übrig bleibt, und diese keine Fortführungsvariable enthält. Die resultierende Gleichung enthält die Repräsentation der strukturierten Komponente.
Hier die Definition der Transformationsregeln:

Substitution: Bei der Substitution wird eine Fortführungsvariable, die in einer beliebigen Gleichung in der RHS vorkommt, durch deren Definitionsgleichung ersetzt [39].

$$\begin{array}{l} x_0 = invoke\ A; \boxed{x_1}; \\ \boxed{x_1} = invoke\ B; \end{array} \longrightarrow \begin{array}{l} x_0 = invoke\ A; \\ \quad invoke\ B; \end{array}$$

Abb. 20: Substition [39]

Kommt eine Fortführungsvariable in keiner RHS einer anderen Regel vor, wird deren Definitionsgleichung aus dem Gleichungssystem eliminiert.

Factorization: Diese Regel wird angewendet, wenn zwei oder mehrere disjunkte Verzweigungen in dieselbe Fortführungsvariable enden. Dabei wird diese herausgehoben und ans Ende der Gleichungen gestellt, umschlossen von einer kombinierten Ausführungsbedingung. Die Bedingung setzt sich aus den Bedingungen der betroffenen Zweige zusammen [39].

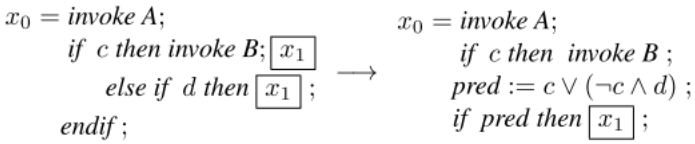


Abb. 21: Factorization [39]

Derecursivation: Sie wird gebraucht, wenn eine Rekursion in einer Gleichung vorkommt, d.h. die gleiche Fortführungsvariable kommt in der LHS und der RHS vor. Dadurch werden Rekursionen eliminiert, in dem der Bereich vom Anfang der RHS bis zur rekursiven Variablen in eine Schleife verpackt wird. Treten in diesem Bereich andere Fortführungsvariablen auf, müssen diese erst mit Hilfe der If-Distribution aus der Schleife verschoben werden [39].

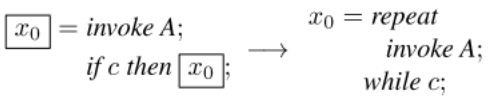


Abb. 22: Derecursivation [39]

If-Distribution: Hier werden verschachtelte und bedingte Verzweigungen in bedingte Verzweigungen umgeschrieben, die in beliebiger Reihenfolge in der Gleichung vorkommen können [39].

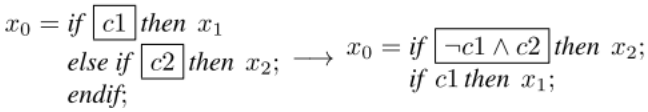


Abb. 23: If-Distribution [39]

Nach Anwenden der Regeln auf das initiale Gleichungssystem in Abb. 19 ergibt sich die folgende Gleichung. Für die Zwischenschritte wird auf [39] verwiesen.

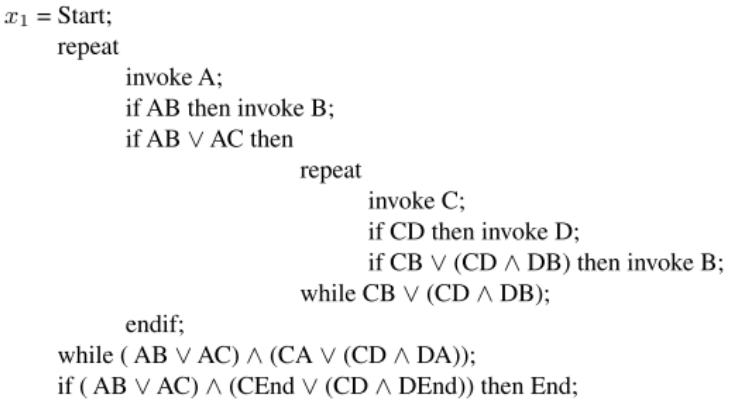


Abb. 24: Prozess mit nicht optimierten Tautologien [39]

3.3.2 Optimierungen

Die Qualität der entstandenen Transformation ist im Wesentlichen von der Anwendungsreihenfolge der Transformationsregeln abhängig. Es gibt grundsätzlich mehrere mögliche Reihenfolgen, die terminieren, jedoch ergeben sich unterschiedliche syntaktische Ergebnisse, die eben auch Qualitätsunterschiede aufweisen [39].

Um ein optimale Anwendungsreihenfolge zu erzielen, wurde folgende Heuristik entwickelt:

- Vor dem Suchen nach einer geeigneten Regel wird für alle Variablen das Vorkommen in der RHS ermittelt. Variablen, die nur einmal vorkommen, werden mit Hilfe der Substitution eliminiert.
- Factorization soll vor Derecursivation ausgeführt werden.
- Derecursivation wird nur vor Substitutionen oder als letzter Schritt ausgeführt.
- If-Distribution wird nur ausgeführt, um
    - eine Variable, die zum Endezustand führt, zum Ende der Gleichung zu verschieben.
    - eine Variable aus dem Geltungsbereich einer Derecursivation zu verschieben.

Als weitere Optimierungsmaßnahme können Tautologien in den Schleifen- und Verzweigungsbedingungen entfernt werden [39]. Z.B. ist $(AB \vee AC)$ eine Tautologie, da damit jeder mögliche Nachfolgezustand von A abgedeckt ist. Dasselbe gilt für $(CEnd \vee (CD \wedge DEnd))$.

3.4 Komponentenerkennung und Klassifizierung

Um eine Transformation durchzuführen, muss der Prozessgraf in Komponenten geteilt werden. Eine Komponente ist in diesem Kontext eine Region des Prozessgrafen, mit einer eingehenden und einer ausgehenden Kante (single-entry single-exit, SESE) [38]. SESE-Komponenten sind eine Grundvoraussetzung für strukturierte Komponenten, und nur diese können in Konstrukte blockorientierter Beschreibungssprachen umgesetzt werden.
Ist der Prozessgraf in Komponenten geteilt, müssen diese klassifiziert (And-Block, Schleife, etc.) werden, damit bei der Übersetzung die richtige Entität des Zielmodells (z.B. Flow, While, Switch) ausgewählt werden kann.
Dafür sind effiziente Methoden notwendig, damit der gesamte Übersetzungsprozess effizient abläuft.

In [38] wird ein zweistufiges Modell vorgestellt, das eine BPMN-Modellinstanz verarbeitet. In der ersten Phase erfolgt die Unterteilung in SESE-Komponenten. In der zweiten Phase wird eine Kontrollflussanalyse der einzelnen Komponenten durchgeführt, um sie zu klassifizieren.

3.4.1 Process Structure Tree

Eine SESE-Komponente hat genau einen Einstiegs- und einen Ausstiegspunkt. Die Knoten die dazwischen liegen, somit in der Komponente enthalten sind, müssen auf mindestens einem Pfad zum Ausstiegspunkt liegen.
Es werden hier nur kanonische SESE-Komponenten betrachtet, die sich nicht überschneiden. Man kann SESE-Komponenten jedoch verschachteln, und diese Relation in Form eines Baums darstellen, der als Process Struktur Tree (PST) bezeichnet wird [38]. Abb. 25 zeigt (a)einen Beispielprozess in BPMN-Notation und (b) den dazugehörigen Process Structure Tree.
In [40] wird ein Algorithmus erläutert, um einen PST in linearer Zeit zu konstruieren.

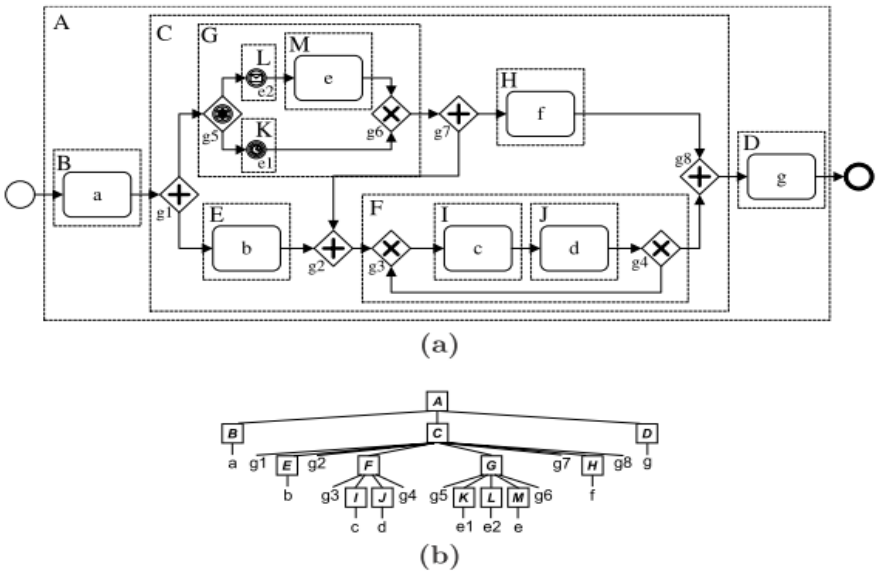


Abb. 25: a) BPMN Beispielprozess b) dazugehöriger PST [38]

3.4.2 Kontrollflussanalyse

Im Zuge der Kontrollflussanalyse werden für die Knoten einer SESE-Komponente verschiedene Mengen anhand von Gleichungen (dataflow equation) gebildet. Diese Mengen werden anschließend verwendet, die Komponente zu klassifizieren und somit ein Pattern in der SESE-Komponente zu identifizieren [38].

Über folgende Gleichungen werden die für die Analyse benötigten Mengen gebildet:

- Trace: Trace(x) gibt jene Knoten an, die x auf einem beliebigen Pfad vorausgehen können.

$$Trace(x) = \{x\} \cup_{p \in pred(x)} Trace(p)$$

- Subtrace: Subtrace(x) gibt jene Aktivitäten(Tasks) an, die x auf einem beliebigen Pfad vorausgehen können.

$$Subtrace(x) = \begin{cases} \{x\} \cup_{p \in pred(x)} Subtrace(p); wenn\ x\ eine\ Aktivität\ ist \\ \emptyset; sonst \end{cases}$$

- SplitTrace: SplitTrace(x) gibt jene Split-Konnektoren an, die x auf einem beliebigen Pfad vorausgehen können.

$$SplitTrace(x) = \begin{cases} \{x\} \cup_{p \in pred(x)} SplitTrace(p); wenn\ x\ ein\ Split\ GW\ ist \\ \emptyset; wenn\ x\ ein\ Join\ GW\ ist \\ \cup_{p \in pred(x)} SplitTrace(p); sonst \end{cases}$$

- DomSplit: DomSplit(x) gibt jene Split-Konnektoren an, die x dominieren, wenn x ein Join-Konnektor ist.

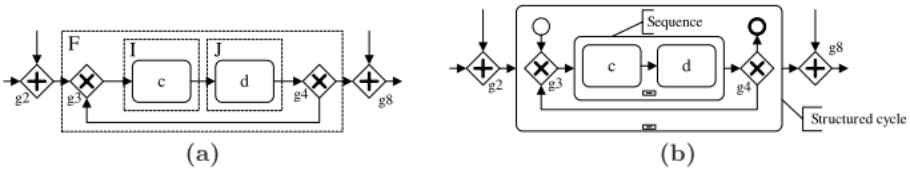
$$DomSplit(x) = \begin{cases} \cap_{p \in pred(x)} SplitTrace(p); wenn\ x\ ein\ Split\ GW\ ist \\ \emptyset; sonst \end{cases}$$

- VisSplits: VisSplit(x) gibt jene Split-Konnektoren an, die von den Vorgängerknoten von x sichtbar sind.

$$VisSplit(x) = \begin{cases} \cup_{p \in pred(x)} SplitTrace(p); wenn\ x\ ein\ Split\ GW\ ist \\ \emptyset; sonst \end{cases}$$

Sind die Mengen für die Knoten einer SESE-Komponente gebildet, kann sie, basierend auf diesen Informationen, klassifiziert werden.

- Sequenzen werden dadurch charakterisiert, dass die Subtrace-Menge mehr als ein Element enthält (siehe Abb. 26c). Damit können interne Sequenzen innerhalb einer Schleife oder Blocks identifiziert werden, aber auch Sequenzen, die von keinem Block umschlossen sind, sind so erkennbar. Interne Sequenzen müssen zuerst reduziert werden, um anschließend Schleifen oder Blöcke zu reduzieren.



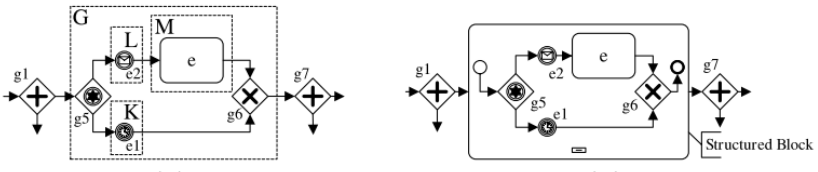
Node	$Trace$	$Subtrace$
$g3$	$\{g3, c, d, g4\}$	$\{\}$
c	$\{g3, c, d, g4\}$	$\{c\}$
d	$\{g3, c, d, g4\}$	$\{c, d\}$
$g4$	$\{g3, c, d, g4\}$	$\{\}$

(c)

Abb. 26: strukturierte Schleife mit interner Sequenz a) der originale Teilgraph, b) Komponenten als Subprozesse realisiert, c) Analyseinformation [38]

- Strukturierte Schleifen sind daran erkennbar, dass sie einen XOR-Join Konnektor als Einstiegspunkt, einen XOR-Split Konnektor als Ausstiegspunkt haben und die Trace-Menge für den Ein- und Ausstiegspunkt gleich ist (siehe Abb. 26). Anmerkend sei noch erwähnt, dass die Trace-Menge in jedem Knoten einer Schleife gleich ist.

- Strukturierte Blöcke besitzen einen Split Konnektor als Einstiegspunkt in die Komponente, einen Join Konnektor als Ausstiegspunkt, und die DomSplit- und VisSplit-Menge enthält genau einen identen Knoten (siehe Abb. 27). $DomSplit = VisSplit \wedge |DomSplit| = |VisSplit| = 1$
  Je nach Typ des Blocks sind nur bestimmte Konnektor Kombinationen zulässig (And-And, Xor-Xor, Or-Or, EventXor-Xor).



Node	$Subtrace$	$SplitTrace$	$DomSplit$	$VisSplits$
$g5$	$\{\}$	$\{g5\}$	$\{\}$	$\{\}$
$e1$	$\{\}$	$\{g5\}$	$\{\}$	$\{\}$
$e2$	$\{\}$	$\{g5\}$	$\{\}$	$\{\}$
e	$\{e\}$	$\{g5\}$	$\{\}$	$\{\}$
$g6$	$\{\}$	$\{\}$	$\{g5\}$	$\{g5\}$

(c)

Abb. 27: Strukturierter Block Sequenz a) der originale Teilgraph, b) Komponenten als Subprozesse realisiert, c) Analyseinformation [38]

- Unstrukturierte Blöcke erfüllen die Anforderungen an strukturierte nicht und sind vor allem daran erkennbar, dass der Einstiegspunkt nicht in der DomSplit-Menge des Ausstiegspunkt liegt.

- quasi-strukturierte Komponenten können als Teilmenge der unstrukturierten Blöcke angesehen werden. Das sind jene Blöcke, die durch Umstrukturierung in einen strukturierten Block umgewandelt werden können.
  Für die Identifizierung sind folgende zusätzliche Mengendefinitionen erforderlich:

$$JoinRTrace(x) = \begin{cases} \{x\} \cup_{s \in succ(x)} JoinRTrace(s); wenn\ x\ ein\ Join\ GW\ ist \\ \emptyset; wenn\ x\ ein\ Split\ GW\ ist \\ \cup_{s \in succ(x)} JoinRTrace(s); sonst \end{cases}$$

$$PostJoins(x) = \begin{cases} \cap_{s \in succ(x)} JoinRTrace(s); wenn\ x\ ein\ Join\ GW\ ist \\ \emptyset; sonst \end{cases}$$

$$VisJoins(x) = \begin{cases} \cup_{s \in succ(x)} JoinRTrace(s); wenn\ x\ ein\ Join\ GW\ ist \\ \emptyset; sonst \end{cases}$$

  Eine quasi strukturierte Komponente ist vorhanden, wenn Konnektor x weder der Ein- noch Ausstiegspunkt einer Komponente ist und PostdomSplit(x) = VisJoins(x) gilt [38]. Auf diese Weise kann jedoch nur erkannt werden, dass kein Gegenstück für einen Join- oder Split-Konnektor vorhanden ist, somit kein gültiges Konnektorpaar gebildet, wird und die Komponente deshalb unstrukturiert ist.

Die Klassifizierung des gesamten BPMN-Prozesses beginnt mit dem Erstellen des Process Structure Trees. Der PST wird dann in postorder durchlaufen, und die jeweilige SESE-Komponente wird analysiert. Der Teilgraf, den die analysierte Komponente repräsentiert, wird in einen Subprozess kopiert, der mit Strukturinformation aus der Analyse versehen wird. Der Subprozess ersetzt den Teilgrafen im Hauptprozess und die Analyse wird im reduzierten PST fortgesetzt. Die Analyse endet, wenn ein einzelne Aktivität oder ein Subprozess übrig bleibt.

4 BPMN-zu-Asbru Transformation

In diesem Kapitel werden die Ergebnisse dieser Arbeit vorgestellt, wobei die Kapitel 4.1 bis 4.3 Analysen und theoretische Grundlagen und Kapitlel 4.4 die Implementierung Beschreibt. Konkret sind das eine musterbasierte Analyse von BPMN und Asbru, in der die unterstützten Kontrollflussmuster der beiden Prozessmodelle gegenübergestellt werden (Abschnitt 4.1). Des Weiteren wird ein Mapping zwischen BPMN- und Asbru-Entitäten definiert (Abschnitt 4.2) und eine Transformationsstrategie ausgewählt, die implementiert werden soll (Abschnitt 4.3). Abschließend wird die Funktionsweise des implementierten Software-Prototyps im Kontext der Kontrollflussmuster beschrieben (Abschnitt 4.4).

4.1 Musterbasierte Analyse von BPMN und Asbru

Transformationen zwischen Prozessbeschreibungssprachen werden nicht nur durch deren Paradigma, wie sie den Kontrollfluss repräsentieren, beeinflusst. So vielfältig die Beschreibungssprachen sind, so vielfältig fallen auch die Möglichkeiten aus, mit ihnen verschiedene Kontrollflussvarianten auszudrücken. Immer wiederkehrende Varianten wurden abstrahiert und zu Kontrollflussmustern (controllflow pattern) verschiedenster Kategorien zusammengefasst.
Prozessbeschreibungssprachen lassen sich mit Hilfe dieser Kontrollflussmuster miteinander vergleichen.

Somit übt die Tatsache, ob ein Kontrollflussmuster von einem Workflow-Model unterstützt wird oder nicht, Einfluss auf den Übersetzungsprozess aus.
Wird ein Muster vom Quellmodell, aber nicht vom Zielmodell der Transformation unterstützt, kann es auch nicht übersetzt werden. Dadurch können nicht alle Prozessinstanzen des Quellmodells übersetzt werden, und eine Schlüsselanforderung des Transformationsprozesses, nämlich die Vollständigkeit, wird eingeschränkt.
Kontrollflussmuster wie sie in [31] vorgestellt werden, bieten einen systematischen Rahmen für eine Analyse und den Vergleich von WfM und werden dafür immer wieder herangezogen.
Das hat mehrere Gründe. Zum einen benützen Anwender von WfM-Werkzeugen Kontrollflussmuster, um das richtige Werkzeug für sich zu finden. Andererseits evaluieren Anbieter dieser Werkzeuge ihr Produkt ebenfalls damit. Somit sind Kontrollflussmuster auf beiden Seiten weit verbreitet und akzeptiert. Des Weiteren haben Kontrollflussmuster den richtigen Abstraktionslevel für solche Vergleiche.
Kontrollflussmuster bilden mit Daten- und Ressourcenmustern ein System, um verschieden Aspekte von Informationssystemen zu analysieren [15]. In dieser Arbeit wird jedoch nur Augenmerk auf die Kontrollflussmuster gelegt.

Kontrollflussmuster werden in sechs Kategorien unterteilt: die Standard KFM, die erweiterten Verzweigungs- u. Synchronisations-KFM, die strukturellen KFM, die Multiple-Instanzen KFM, die zustandsbasierten KFM und die Abbruch KFM.
Im Folgenden werden für jede Kategorie die entsprechen Muster erläutert und deren Umsetzung in BPMN und Asbru gezeigt. Die Informationen dafür stammen für BPMN aus [15] und für Asbru aus [1] und [41]. Die gerade genannten Arbeiten verwenden ebenfalls Kontrollflussmuster zur Analyse, und deren Ergebnisse werden in den folgenden Unterabschnitten präsentiert.

4.1.1 Standard KFM (basic control-flow pattern)

Kontrollflussmuster dieser Kategorie definieren grundlegendes Verhalten im Kontrollfluss und sind in fast jeder WfM vorhanden.

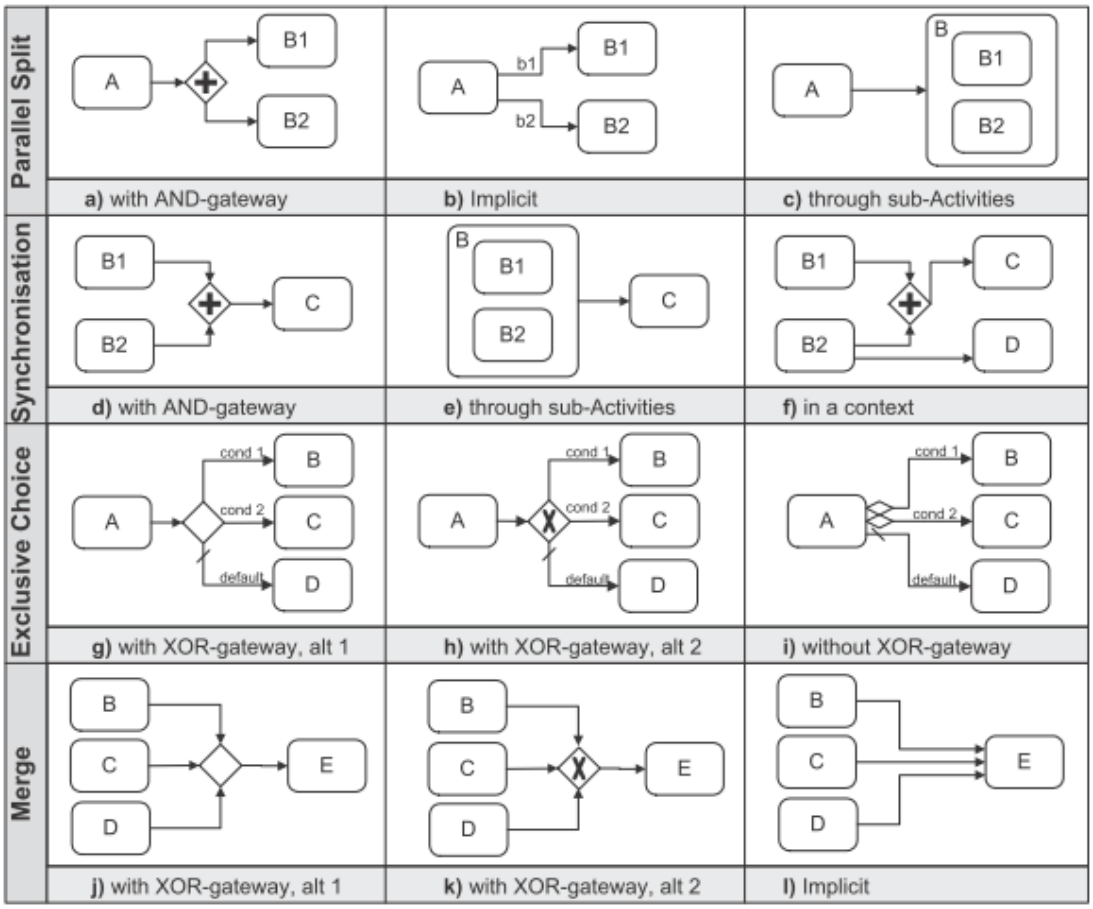


Abb. 28: grundlegende KFM in BPMN [15]

4.1.1.1 Sequence

Muster: Die Sequenz wird dadurch charakterisiert, dass zwei oder mehrere Aktivitäten hintereinander ausgeführt werden.
BPMN: Dies wird in BPMN durch eine gerichtete Kontrollflusskante (Sequenceflow) zwischen den Aktivitäten dargestellt.
Asbru: Für eine sequenzielle Ausführung von Plänen (Aktivitäten) wird in Asbru ein Plan mit sequentiellem Plan-Typ eingesetzt. Die Subpläne des sequentiellen Plans werden in der spezifizierten Reihenfolge ausgeführt, und ein Plan beginnt, wenn der vorrangegangene beendet wurde [1].

4.1.1.2 Parallel Split

Muster: Beim Parallel Split wird ein Kontrollflusstoken in mehrere Token aufgeteilt und führt zu einer nebenläufigen Ausführung von Aktivitäten.
BPMN: In BPMN gibt es den AND-Konnektor (AND-split gateway), mit dem der Kontrollfluss in nebenläufige Zweige aufgeteilt wird (siehe Abb. 28a). Dasselbe kann auch ohne AND-Konnektor mit zwei ausgehenden Kanten b1 und b2 dargestellt werden, die zu unterschiedlichen Folgeaktivitäten führen (siehe Abb. 28b). Eine weitere Variante ist in Abb. 28c zu sehen, in der die parallelen Aktivitäten in einem Subprozess ausgeführt werden.
Asbru: Dafür wird ein Plan mit parallelem Plan-Typ in Asbru verwendet. Alle Subpläne werden gleichzeitig gestartet, parallel ausgeführt, und der parallele Plan wird beendet, wenn alle Subpläne beendet wurden [1].

4.1.1.3 Synchronisation

Muster: Hier werden mehrere Kontrollflusstoken aus mehreren Zweigen zu einem einzigen Token vereint.
BPMN: Dafür wird in BPMN der AND-Join Konnektor verwendet, der ein Token an die ausgehende Kante weiterleitet, wenn er aus jeder eingehenden Kante ein Token erhalten hat (siehe Abb. 28d + f). Eine andere Lösung verwendet einen Subprozess (siehe Abb. 28d). Sind alle parallel ausgeführten Aktivitäten im Subprozess beendet, wird der Token an Aktivität C weitergeleitet.
Asbru: Durch die blockorientierte Beschaffenheit Asbrus ist dieses Muster Bestandteil des Parallel Split Musters, da alle Subpläne bzw. Zweige beendet sein müssen, damit der parallele Plan beendet wird.

4.1.1.4 Exclusive Choice

Muster: Hier wird das Kontrollflusstoken genau an einen Zweig, aus mehreren möglichen, weitergegeben.
BPMN: BPMN setzt dies mit dem XOR-Split Konnektor um, wobei eine Kante als Default-Zweig dienen kann (siehe Abb. 28g + h). Unabhängig von den spezifizierten Verzweigungsbedingungen wird gewährleistet, dass genau an einen Zweig das Token weitergeleitet wird.
Abb. 28i zeigt eine Variante, bei der das gewünschte Verhalten durch mehrere bedingte, ausgehende Kanten realisiert wird. Die Kantenbedingungen müssen hier jedoch so definiert sein, dass immer nur eine einzelne Bedingung erfüllt ist, um das gewünschte Verhalten zu erzielen.
Asbru: Dieses Muster wird in Asbru durch das if-then-else Element realisiert, das eine Bedingung erfordert und einen Then-Zweig und einen Else-Zweig definiert. Dadurch wird sichergestellt, dass genau eine Verzweigung aktiviert werden kann [1].
Es gibt jedoch eine zweite Variante, die das Muster mit Filterbedingungen umsetzt. Dafür wird ein Plan mit Plan-Typ „unordered“ verwendet, dessen Subpläne, die die verschiedenen Zweige darstellen, mit Filterbedingungen versehen werden, die sich nicht überlappen (ähnlich Abschnitt 4.1.2.1).

4.1.1.5 Simple Merge

Muster: Mehrere Zweige im Prozess kommen in einem Punkt zusammen und, unabhängig aus welchem Teil ein Token kommt wird es ohne Verzögerung an die nachkommende Aktivität weitergeleitet.
BPMN: BPMN realisiert dies mit einem XOR-Join Konnektor (siehe Abb. 28j + k) oder über Kontrollflusskanten, die in einer Aktivität zusammengeführt werden (siehe Abb. 28l). Die letzte Lösungsvariante eignet sich jedoch nur, wenn das Token auch nur aus einem einzigen Zweig ankommen kann.
Asbru: Durch die Strukturiertheit von blockorientierten Sprachen ist dieses Muster in der If-then-else-Variante des Exlusive Choice enthalten [1].
Wird für Exclusive Choice jedoch die Variante mit dem ungeordneten Plan verwendet, muss die Fortsetzungsbedingung eines Plans durch das *wait-for* Element ausgedrückt, auf „*one*“ gesetzt werden. Dadurch wird spezifiziert, dass das Beenden eines einzelnen Subplans ausreicht, um den übergeordneten Plan zu beenden (siehe auch Abschnitt 4.1.2.4).

4.1.2 Erweiterte Verzweigungs- u. Synchronisations-KFM (advanced branching and synchronisation patterns)

Die hier angeführten Kontrollflussmuster sind häufig in Geschäftsprozessen anzutreffen und etwas komplexer als die grundlegenden KFM.

4.1.2.1 Multiple Choice

Muster: Darunter versteht man, wenn bei einer Verzweigung des Kontrollflusses ein, mehrere oder alle Zweig(e) das Token erhalten.
BPMN: In BPM kann dies mit dem OR-Split (Abb. 29a) oder dem Komplex-Split Konnektor (Abb. 29c) realisiert werden. Abb. 29b zeigt die Lösung mit bedingten Kanten, wobei der Unterschied zu Exclusive Choice darin liegt, dass sich die Kantenbedingungen auch überschneiden dürfen.

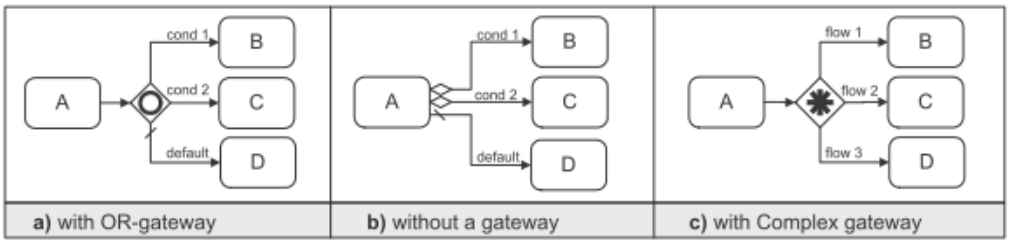


Abb. 29: Multiple Choice Varianten in BPMN [15]

Asbru: Dies wird in Asbru in der Form realisiert, dass ein Plan mit ungeordnetem Plan-Typ verwendet wird, dessen Subpläne, die die verschiedenen Zweige darstellen, mit Filterbedingungen versehen werden, die sich überlappen [1].

4.1.2.2 Synchronising Merge

Muster: Damit lassen sich mehrere Token in einem Punkt synchronisieren und zu einem einzelnen Token vereinigen. Es synchronisiert die Anzahl der Token, die durch Multiple Choice eingebracht wurden.
BPMN: BPMN stellt dafür den OR-Join Konnektor zur Verfügung (Abb. 30).

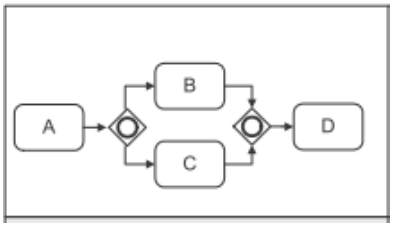


Abb. 30: BPMN Synchronising Merge [15]

Asbru: Dieses Muster wird von Asbru nur indirekt unterstützt, allerdings in zwei Varianten [1].
Wenn die Zweige, die synchronisiert werden sollen, durch ein *If-Then-Else* Element entstanden sind, dann wird nach dem *If-Then-Else* Element vor dem nächsten auszuführenden Schritt synchronisiert.

Werden die Verzweigungen jedoch mit einem ungeordneten Plan realisiert (siehe Abschnitt 4.1.2.1), dann wird synchronisiert, wenn dessen Fortsetzungsbedingung (continuation condition), ausgedrückt durch das *wait-for* Element, erfüllt ist [1].

4.1.2.3 Multiple Merge

Muster: Beim Multiple Merge Muster werden Token unmittelbar an die nachfolgende Aktivität weitergegeben. Treten mehrere Token aus unterschiedlichen Zweigen auf, wird die Nachfolgeaktivität für jedes Token aufgerufen.
BPMN: BPMN setzt dies gleich um wie Simple Merge (siehe Abb. 28 j,k,l)

Asbru: Asbru bietet für dieses Muster keinen adäquaten Mechanismus [1].

4.1.2.4 Discriminator

Muster: Das Discriminator Muster synchronisiert mehrere Token, gibt jedoch nur das erste Token an die weiterführende Aktivität weiter und verwirft alle anderen.
BPMN: BPMN unterstützt dieses Muster nicht direkt. Diese Einschränkung kann jedoch mit Textbemerkungen oder einem Komplexen-Join Konnektor umgangen werden (siehe Abb. 31 a, b).

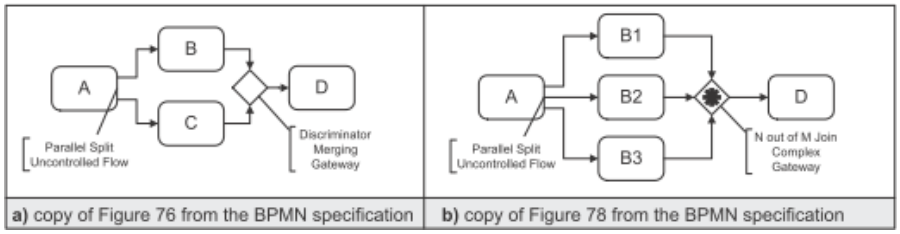


Abb. 31: BPMN-Lösungen für Discriminator Muster [15]

Asbru: In Asbru realisiert man dieses Muster, indem man die Fortsetzungsbedingung eines Plans durch das *wait-for* Element ausgedrückt, auf „*one*“ setzt. Dadurch wird spezifiziert, dass das Beenden eines einzelnen Subplans ausreicht, um den Plan zu beenden [1].

4.1.3 Zustandsbasierte KFM

Muster dieser Klasse zeichnen sich dadurch aus, dass der weitere Verlauf an einem Punkt im Prozess von dessen Zustand abhängt.

4.1.3.1 Deferred Choice

Muster: Dabei wird an einem Punkt im Prozess aus mehreren möglichen Verzweigungen gewählt, wie der Prozess weiter verläuft. Die Entscheidung wird jedoch nicht im Prozess selbst entschieden, sonder durch dessen Umgebung –z.B. durch die Eingabe eines Users.
BPMN: BPMN bildet dieses Kontrollflussmuster mit Event-Basiertem-Split Konnektor und Message-Events (Abb. 32a) bzw. Receive Tasks (Abb. 32b) ab. Für beide Varianten gilt, wenn der Prozess die entsprechende Nachricht (b oder c) erhält, wird der dazugehörige Teilprozess ausgeführt.

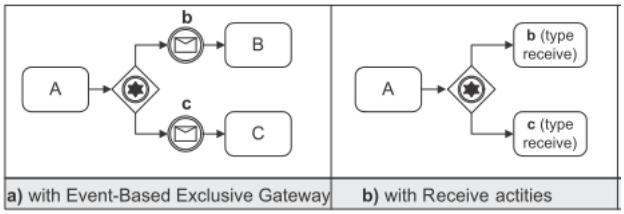


Abb. 32: BPMN-Lösungen für Deferred Choice [15]

Asbru: Es kann in Asbru mit einem Plan umgesetzt werden, der folgende Bedingungen erfüllt:

- Der Plan-Typ muss auf „any-order“ gesetzt sein.
- Die Fortsetzungsbedingung, durch das *wait-for* Element ausgedrückt, muss auf „*one*“ gesetzt sein.
- In den Subplänen muss die Option für die Userbestätigung als Filterbedingung aktiviert sein.

Dadurch wird dem User für jede Option eine Bestätigung aufgeworfen. Der bestätigte Subplan wird als einziger ausgeführt (any-order), und nach Beendigung dieses Subplans der übergeordnete Plan beendet (wait-for one), und der Prozess weiter ausgeführt [1].

4.1.3.2 Interleaved Parallel Routing

Muster: Dieses Muster wird verwendet, um mehrere Aktivitäten in beliebiger Reihenfolge auszuführen.
BPMN: BPMN sieht dafür die Ad-Hoc-Aktivität vor. Dadurch sind beliebige Reihenfolgen A.C, C.A möglich (Abb. 33a) oder, wenn man Sequenzen verwendet, auch A.B.C.D oder C.D.A.B (Abb. 33b).
Sollen auch ineinander verschränkte Abfolgen, wie z.B. A.C.B.D, A.C.D.B oder C.A.B.D, möglich sein, können die Lösungen in Abb. 33c und d herangezogen werden.

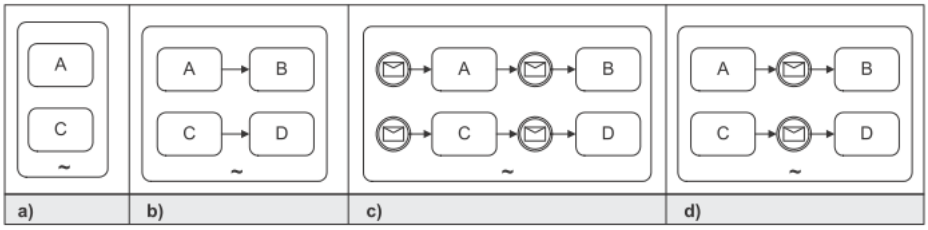


Abb. 33: BPMN-Varianten zu Interleaved Parallel Routing [15]

Asbru: Dies wird mit einem Plan mit dem Plan-Typ „any-order“ realisiert, da damit nur ein Subplan zur selben Zeit ausgeführt werden kann [1]. Partielle Ordnung von Subplänen müssen über Filterbedingungen ausgedrückt werden.

4.1.3.3 Milestone

Muster: Mit diesem Muster wird ausgedrückt, dass ein Zustand erst ausgeführt wird, wenn ein bestimmter Zustand im Prozess erreicht wird.
BPMN: Dieses Muster wird nicht direkt von BPMN unterstützt, es kann jedoch simuliert werden.
Abb. 34a zeigt eine Lösung, in der Aktivität D erst ausgeführt wird, wenn B beendet wurde, jedoch bevor E ausgeführt wird.
Die vorangegangene Lösung berücksichtigt nicht, dass ein Meilenstein nur zeitlich begrenzt gültig ist und ablaufen kann, wie es in Abb. 34b berücksichtigt wird.

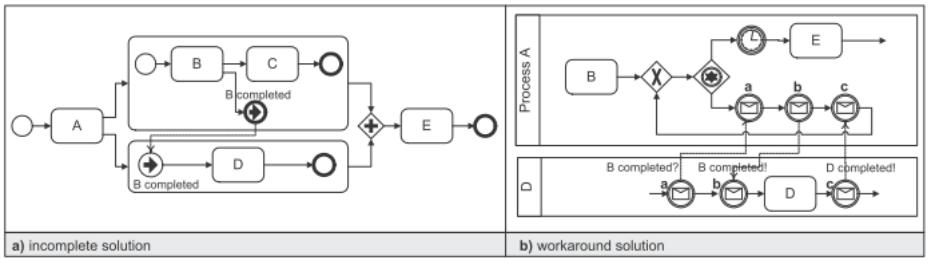


Abb. 34: BPMN-Varianten zu Milestone [15]

Asbru: Asbru bietet für dieses Muster keinen adäquaten Mechanismus [1].

4.1.4 Multiple-Instanzen (MI) KFM

Die folgenden Muster befassen sich mit Situationen, in denen mehrere Instanzen einer Aktivität zur selben Zeit im selben Prozess ausgeführt werden.

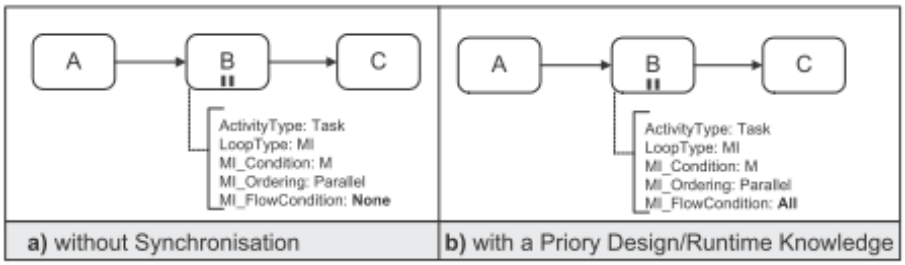


Abb. 35: BPMN-Lösungen MI Muster [15]

4.1.4.1 MI ohne Synchronisation

Muster: Es werden möglicherweise mehrere Instanzen einer Aktivität ausgeführt, und für jede beendete Instanz wird ein Token erzeugt und an die nachfolgende Aktivität unsynchronisiert weitergegeben.
BPMN: In BPMN wird eine MI Aktivität dafür verwendet, dass das MI_FlowCondition Attribut auf None gesetzt hat, um sich nicht synchronisiert zu verhalten (siehe Abb. 35a).
Asbru: Asbru bietet für dieses Muster keinen adäquaten Mechanismus.

4.1.4.2 MI mit a priori Design-Time Wissen

Muster: Es werden möglicherweise mehrere Instanzen einer Aktivität ausgeführt und nach Beendigung aller Instanzen wird ein Token an die nachfolgende Aktivität weitergegeben. Die Anzahl der aktiven Instanzen ist zur Design-Time des Prozesses bekannt.
BPMN: In BPMN wird eine MI Aktivität dafür verwendet, dass das MI_FlowCondition Attribut auf All gesetzt hat, um die Synchronisation aller Instanzen zu indizieren (siehe Abb. 35b).
Asbru: Dieses Muster kann nicht direkt mit Asbru abgebildet werden, es kann jedoch simuliert werden. Dafür wird ein paralleler Plan mit so vielen Subplänen definiert, wie Instanzen der Aktivität benötigt werden. Für jeden Subplan wird eine Instanz erzeugt, und der übergeordnete Plan wird beendet, wenn alle Subpläne beendet wurden.

4.1.4.3 MI mit a priori Run-Time Wissen

Muster: Es werden möglicherweise mehrere Instanzen einer Aktivität ausgeführt, und nach Beendigung aller Instanzen wird ein Token an die nachfolgende Aktivität weitergegeben. Die Anzahl der aktiven Instanzen ist zur Laufzeit bekannt, bevor sie erstellt werden.
BPMN: Die Lösung aus Abschnitt 4.1.4.2 ist auch auf dieses KFM anwendbar, jedoch ist die Anzahl der aktiven Instanzen erst zur Laufzeit bekannt.
Asbru: Asbru bietet für dieses Muster keinen adäquaten Mechanismus.

4.1.4.4 MI ohne a priori Run-Time Wissen

Muster: Es werden möglicherweise mehrere Instanzen einer Aktivität ausgeführt, und nach Beendigung aller Instanzen wird ein Token an die nachfolgende Aktivität weitergegeben. Die Anzahl der aktiven Instanzen ist zur Laufzeit nicht bekannt, und es können auch, während bereits Instanzen ausgeführt werden, neue hinzukommen.
BPMN: Dieses Kontrollflussmuster wird von BPMN nicht direkt unterstützt, es kann jedoch eine alternative Lösung in BPMN realisiert werden (siehe Abb. 36).

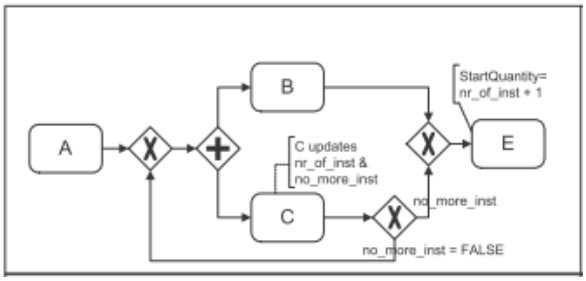


Abb. 36: BPMN MI o. a priori Run-Time Wissen [15]

Asbru: Asbru bietet für dieses Muster keinen adäquaten Mechanismus.

4.1.5 Strukturelle KFM

Muster in dieser Klasse beschreiben, ob die Ausdruckskraft bzgl. der Strukturierung des Prozesses eingeschränkt ist oder nicht.

4.1.5.1 Arbitrary Cycles

Muster: Damit können Schleifen mit mehreren Ein- und Ausstiegspunkten beschrieben werden.
BPMN: BPMN unterstützt dieses Muster direkt. Mit Hilfe von Kontrollflusskanten und Konnektoren können solche zyklische Konstrukte erstellt werden.
Asbru: Unstrukturierte Schleifen werden von Asbru nicht unterstützt, da Asbru für Schleifen den zyklischen Plan vorsieht. Dieser hat nur einen Ein- und einen Ausgangspunkt.

4.1.5.2 Implicit Termination

Muster: Es sagt aus, dass es eine Notation gibt, die ausdrückt das ein Subprozess terminiert, wenn keine weiteren aktiven Token im Prozess vorhanden sind.
BPMN: BPMN unterstützt dies mit dem End-Event, der den letzten Token konsumiert und einen ganzen Prozess abschließt.
Asbru: Ein Prozess/Subprozess wird in Asbru beendet, wenn der sich in Ausführung befindliche Plan ausgeführt, also regulär beendet wurde, oder ein Abbruch stattfand.

4.1.6 Abbruch KFM

Diese Muster beschäftigen sich mit dem Abbruch von Aktivitäten/Prozessen.

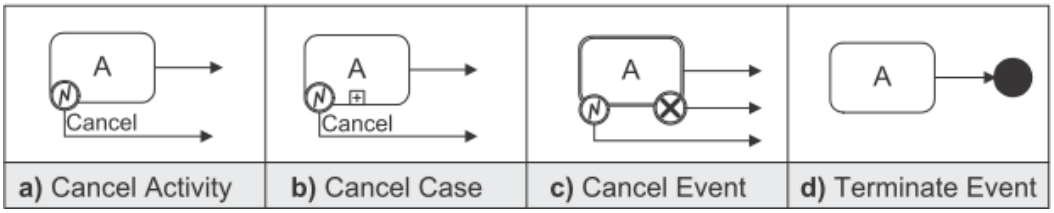


Abb. 37: Abbruch Muster in BPMN [15]

4.1.6.1 Cancel Task

Muster: Dieses Muster beschreibt, unter welchen Umständen eine Aktivität abgebrochen wird und einen etwaigen alternativen Kontrollfluss ausführt.
BPMN: In BPMN wird hierfür eine Aktivität mit einem Error-Event verbunden, von dem eine separate Kante wegführt (siehe Abb. 37a). Der damit assoziierte alternative Kontrollfluss wird somit im Fehlerfall ausgeführt.
Asbru: Asbru unterstützt dieses Muster durch Abbruchbedingungen (cancel condition) in Plänen/Aktivitäten [1].

4.1.6.2 Cancel Case

Muster: Dieses Muster ähnelt dem Cancel Task Muster. Dabei wird jedoch nicht eine einzelne Aktivität, sondern ein ganzer Prozess mit allen beinhalteten Aktivitäten abgebrochen.

BPMN: Abb. 37b zeigt wie ein Subprozess mit dem Error-Event verbunden ist und die Kante, die den alternativen Kontrollfluss im Fehlerfall indiziert.
Alternative Lösungen werden in Abb. 37c und d aufgezeigt.
Asbru: Dieses Muster wird durch eine Abbruchbedingung im Wurzelplan realisiert. Der Wurzelplan ist der Plan in der Verschachtelungshierarchie, der auf der obersten Ebene angesiedelt ist und nicht Teil eines anderen Plans ist [1].

4.1.7 Gegenüberstellung der Resultate

Probleme treten bei der Transformation dort auf, wo ein Konstrukt im Quellmodell unterstützt wird, welches im Zielmodell nicht umgesetzt werden kann. Basierend auf den Resultaten der musterbasierenden Analyse aus Abschnitt 4.1.1 bis 4.1.6 wird ein musterbasierender Vergleich zwischen BPMN und Asbru in diesem Abschnitt angestellt.
Einerseits dient dies dazu, um einen Überblick über die unterstützten Kontrollflussmuster zu bekommen, und andererseits um Diskrepanzen, die durch nicht unterstützte Muster entstehen, von vornherein zu identifizieren.

zeigt eine Zusammenfassung der realisierbaren Kontrollfluss Muster und somit die Resultate aus Abschnitt 4.1.1 bis 4.1.6. Wird das Kontrollflussmuster durch eine Entität oder Konstrukt des Modells direkt unterstützt, dann wird dies durch „+“ symbolisiert. Wird ein KFM. nur teilweise oder gar nicht unterstützt, wird dies in der Tabelle mit „+/-“ bzw. „-“ angezeigt.

BPMN wie auch Asbru unterstützen alle Standard Kontrollflussmuster direkt und vollständig. Daher sollten hier keine Konflikte bei der Transformation auftreten.

Die erweiterten Verzweigungs- und Synchronisations-Muster werden nicht mehr von beiden Sprachen gleichermaßen unterstützt.
Das Multi-Merge KFM ist dann erfüllt, wenn in einem Synchronisationspunkt mehrere Zweige in einen zusammengeführt werden, und jedes eingehende Token an den ausgehenden Zweig unverzüglich weitergeleitet wird.
BPMN unterstützt dies durch einen XOR-Join Konnektor bzw. durch mehrere eingehende, uneingeschränkte Kanten einer Aktivität.
Asbru hat dafür keinen Mechanismus um dies umzusetzen, da dafür keine passende Continuation Specification vorhanden ist.

Ein generelles Problem zwischen graf- und blockorientierten Prozessbeschreibungssprachen ist, dass unstrukturierte Schleifen, mit mehreren Ein- und/oder Ausstiegspunkten, in grafbasierten Sprachen möglich sind in blockorientierten jedoch nicht. Deshalb unterstützt BPMN das Arbitrary-Cycle Muster, Asbru jedoch nicht. Hierfür müssen, im Bezug auf den Transformationsprozess, geeignete Strategien entwickelt bzw. bekannte angepasst werden, um unbefriedigende Einschränkungen zu vermeiden.
Die restlichen der strukturellen Muster werden wieder von beiden Sprachen unterstützt.

Die Multiple-Instanzen Kontrollflussmuster werden von Asbru nicht unterstützt, mit Ausnahme des „Mi with a priori Design Time Knowledge“ Musters. Dies kann simuliert werden, indem für jede benötigte Instanz ein paralleler Zweig mit der relevanten Aktivität modelliert wird.

Tabelle 1: Zusammenfassung der unterstützten Kontrollflussmuster von BPMN u. Asbru aus [1], [15], [41]

Muster	BPMN	Asbru
Standard Kontrollflussmuster (Basic Control-flow)		
Sequence	+	+
Parallel Split	+	+
Synchchronisation	+	+
Exclusive Choice	+	+
Simple Merge	+	+
Erweiterte Verzweigungen & Synchronisation (Advanced Branching & Synchronisation)		
Multiple Choice	+	+
Synchronising Merge	+/-	+/-
Multiple Merge	+	-
Structured Discriminator	+/-	+
Strukturelle Kontrollflussmuster (Structural Patterns)		
Arbitrary Cycles	+	-
Implicit Termination	+	+
Multiple-Instanzen Kontrollflussmuster (Multiple Instances Patterns)		
MI without Synchronisation	+	-
Mi with a priori Design Time Knowledge	+	+/-
Mi with a priori Runtime Knowledge	+	-
Mi without a priori Runtime Knowledge	-	-
Zustandsbasierte Kontrollflussmuster (State-Based Patterns)		
Deferred Choice	+	+
Interleaved Parallel Routing	+/-	+
Milestone	-	-
Abbruch Kontrollflussmuster (Cancelation Patterns)		
Cancel Task	+	+
Cancel Case	+	+

4.2 Mapping zwischen BPMN und Asbru

Transformationen beruhen darauf, Elemente des Quellprozesses in semantisch äquivalente Elemente des Zielprozesses umzuwandeln, um so einen Ausgangsprozess zu erhalten, der sich wie der Eingangsprozess verhält. Egal, ob die Transformation musterbasiert erfolgt oder auf Metamodellelementen beruht, das Identifizieren und Wissen über semantisch äquivalente Elemente ist für eine korrekte Übersetzung unerlässlich.

In diesem Abschnitt werden deshalb BPMN-Elementen Asbru-Elemente zugeordnet, die sich gleichwertig verhalten. Damit soll eine Mapping-Funktion definiert werden, die in der Implementierung des Transformationssystems ein- und umgesetzt wird, und die eine korrekte Transformation erst ermöglicht.

Um die Ergebnisse zu erzielen, wurden die relevanten BPMN-Elemente mit Hilfe des BPMN-Spezifikationsdokuments [6] identifiziert und die semantische Bedeutung abgeklärt. Anschließend wurde durch Recherche in der Asbru-Spezifikation [12] versucht, entsprechende semantische Äquivalente zu finden.

Bedingt durch die unterschiedlichen Ausdruckstärken der beiden Prozessmodelle lassen sich nicht zu jedem BPMN-Element ein entsprechendes Asbru-Konstrukt finden. Dies ist ebenfalls eine Erkenntnis, die genutzt wird, um geeignete Einschränkungen des Eingangsmodells zu formulieren.
Somit soll dieser Abschnitt eine Mapping-Funktion, aber auch Einschränkungen definieren.

Die folgenden Unterabschnitte beschäftigen sich mit dem Mapping der BPMN-Elemente und gliedern sich nach den BPMN-Elementkategorien, nämlich: Tasks, Sub-Processes, Events und Gateways.

4.2.1 Tasks

Tasks sind das zentrale Element eines BPMN Prozesses und stellen Aktivitäten dar, die nicht weiter zerlegt werden können. Aktivitäten sind Teilschritte des Gesamtprozesses, die in bestimmten Reihenfolgen ausgeführt werden, um den Geschäftsprozess erfolgreich zu beenden.
Tasks unterscheiden sich anhand ihres Aufgaben-Typs und anhand von Markierungen.

Abb. 38: Tasks versehen mit den drei BPMN Task-Markierungen [6]

Der Task wird als Rechteck mit abgerundeten Ecken dargestellt und die jeweilige Markierung wird innerhalb des Tasks mittig an die untere Begrenzungsline gestellt (Abb. 38).
Es gibt drei Markierungen für Tasks und ein Task kann mit bis zu zwei dieser Markierungen versehen sein, die Auskunft über das Ausführungsverhalten des Tasks geben [6]:

(1) Die Schleifen-Markierung (Abb. 38, Loop) sagt aus, dass der Task mehrmals ausgeführt wird.
(2) Die Multi-Instanz-Markierung (Abb. 38, Multi-Instance), dass mehrere Instanzen dieses Tasks parallel ausgeführt werden.
(3) Die Kompensation-Markierung (Abb. 38, Compensation), dass mit diesem Task Teile eines regulären Ablaufs rückgängig gemacht werden.

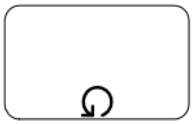
Der Aufgaben-Typ wird in der oberen linken Ecke des Tasks dargestellt und gibt Auskunft über den Charakter des Tasks. So sagt z.B. der Service Task aus, dass er ein Service, wie ein Web Service oder eine automatisierte Applikation, aufruft. Markierungen und Aufgaben-Typen können auch in Kombinationen auftreten [6].

Tabelle 2 listet die verschiedenen BPMN-Task-Elemente auf und zeigt die semantischen Äquivalente in Asbru dazu.

Tabelle 2: Mapping von BPMN-Tasks zu Asbru. Symbole aus [6]. Spalte 3 ist geteilt, wobei oben die BPMN-XML Repräsentation und unten die dazugehörige Asbru-Umsetzung zu finden ist.

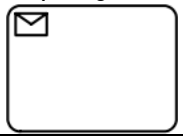
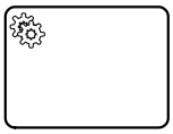
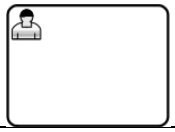
	Grafisches BPMN-Element	BPMN-Xml bzw. Asbru-Umsetzung
a	abstrakter Task 	`<task id="A" name="A"> ...` `</task>`

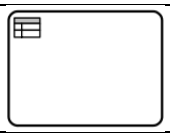
		`<plan name="A" title="A">...`
		`<plan-body>`
		`<user-performed />`
		`</plan-body>`
		`</plan>`
		
		Referenz auf Plan A in einem plan-body *Element:*
		`<plan-activation> <plan-schema name="A" /> </plan-activation>`

b	Schleife 	`<task id="A" name="A">`
		`<standardLoopCharacteristics testBefore="true" loopMaximum="-1">`
		`<loopCondition>a==1</loopCondition>`
		`</standardLoopCharacteristics>`
		`</task>`

		`<plan name="Schleife">...`
		`<plan-body>`
		`<iterative-plan>`
		`<do-repeatedly>`
		`<plan-activation> <plan-schema name="A" />`
		`</plan-activation>`
		`</do-repeatedly>`
		`<termination-condition>...<!--Abbruchbedingung -->`
		`</termination-condition>`
		`</iterative-plan>`
		`</plan-body>`
		`</plan>`
		...
		`<plan name="A" title="A">`
		`<plan-body><user-performed /></plan-body>`
		`</plan>`

c	Parallele 	`<task id="A" name="A">`
		`<MultiInstanceLoopCharacteristics isSequential="[true|false]"`
		`behavior="All">`
		`<loopCardinality>5</loopCardinality>`
		`<completionCondition>b==c<completionCondition>`
		`</MultiInstanceLoopCharacteristics>`

	bzw. Sequentielle  Mehrfachausführung	</task> siehe (a) abstrakt Task
d	Kompensation 	<task id="A" name="A" isForCompensation="true"> ... </task> siehe (a) abstrakt Task
e	Senden Task 	<sendTask id="A" name="A"> ... </sendTask> <plan name="A"> <conditions> <filter-precondition confirmation-required="yes" > ... </filter-precondition> </conditions> ... </plan>
f	Empfangen Task 	<receiveTask id="A" name="A"> ... </receiveTask> Siehe (e) Senden Task
g	Service Task 	<serviceTask id="A" name="A" implementation="##unspecified" operationRef="y"> ... </serviceTask> <operation name="y" implementationRef="..."> <inMessageRef>mi</inMessageRef> <outMessageRef>mo</outMessageRef> </operation> <domain>... <primitive-plan-def name="y" class-name="class1" method-name="method1" return-typ="mo"> <argument name="mi.arg1" typ="String"/> <argument name="mi.arg2" .../> </primitive-plan-def> </domain>
i	Benutzer Task 	<userTask id="A" name="A"> ... </userTask> siehe (a) abstrakt Task
j	Manueller Task 	<manualTask id="A" name="A"> ... </manualTask> siehe (a) abstrakt Task
k	Geschäftsregel Task	<businessRuleTask id="A" name="A"> ...

		*</businessRuleTask>*
		siehe (a) abstrakt Task
I	Skript Task 	*<scriptTask id="A" name="A">* *...* *</scriptTask>*
		siehe (a) abstrakt Task

Ein *abstrakter Task* (Tabelle 2 a) hat keinen besonderen Verwendungszweck, wird weder durch eine Markierung oder einen Aufgaben-Typ gekennzeichnet und stellt die allgemeinste Form des Tasks dar. Asbru stellt dies mit einem Plan dar, der in seinem *plan-body* Element ein *user-performed* Element enthält (Tabelle 2 a, oben). Dies stellt eine Aktivität dar, die manuell ausgeführt werden kann. Zumeist stellt es Tasks dar, die nicht weiter aufgegliedert werden bzw. werden können.
Referenziert wird auf diesen Plan mit Hilfe des *plan-activation* Elements (Tabelle 2 a, unten) und kann so von anderen Asbru Plänen aus aktiviert werden [6], [12].

Ein *Schleifen-Task* (Tabelle 2 b) wird in zwei Teilen abgebildet. Zuerst wird der Task, der die eigentliche Arbeitseinheit darstellt in einem Plan (Plan A) abgebildet (Tabelle 2 b, unten). In Kombination mit dem Plan (Plan Schleife), der die Schleife mit einem *iterative-plan* Element simuliert und Plan A bei jeder Iteration aktiviert, kann der Schleifen-Task in Asbru imitiert werden [6], [12].

Der *Senden-Task* (Tabelle 2 e) bzw. der *Empfangen-Task* (Tabelle 2 f) zeigen, dass in der Aktivität eine Nachricht an einen anderen Prozess gesendet bzw. von ihm empfangen wird. Der Task wird erfolgreich beendet, und der Kontrollfluss wird fortgesetzt, wenn die Nachricht verschickt bzw. empfangen wurde.
Das Empfangen bzw. Senden von externen Nachrichten kann mit Asbru nicht dargestellt oder automatisiert simuliert werden. Darum werden die beiden Tasks in Asbru-Pläne abgebildet, deren Aktivierung vom Benutzer bestätigt werden muss.
Dies wird in Asbru mit dem *confirmation-required* Attribut im *filter-precondition* Element ausgedrückt (Tabelle 2 e). Die Intention dahinter ist, dass der Benutzer bestätigt, die Nachricht versendet bzw. empfangen zu haben [6], [12].

Der Service Task (Tabelle 2 g, BPMN-Teil) beschreibt eine Aktivität, die eine Methode eines Services (Webservice oder anderes automatisiertes System) aufruft. Dabei verweist das *serviceTask* Element optional auf ein *operation* Element, welches den Namen (*name* Attribut) der Mehode und über ein- und ausgehenden Nachrichtenreferenzen (*inMessageRef* bzw. *outMessageRef* Element) die Eingangs- bzw. Rückgabeparameter definieren.
Die Asbru-Spezifikation sieht das *primitive-plan-def* Element (Tabelle 2 g, Asbru-Teil)vor, um komplexe Abläufe in einer externen Programmiersprache (z.B. Java oder C) zu implementieren. Damit kann der qualifizierte Name der Methode (*class-name* bzw. *name* Element), die Eingangs- (*argument* Elemente) und Rückgabeparametertypen (*return-typ* Element) auf die Implementierung definiert werden. Das ausführende System ermöglicht dadurch den Aufruf der Methode. Leider ist dieses Element im Moment in den Asbru Interpretern nicht umgesetzt und wird erst in späteren Versionen berücksichtigt werden [6], [12].

Mehrfache Instanziierung eines Tasks, wie bei der *Mehrfachausführung* (Tabelle 2 c), sind in Asbru nicht möglich und müssen durch den Benutzer ausgeführt werden, sowohl in der parallelen als auch der sequentiellen Abarbeitungsvariante.
Die Information, die in der *Kompensationsmarkierung* (Tabelle 2 d) steckt, nämlich Teilaufgaben im Prozess rückgängig zu machen, können in Asbru nicht dargestellt und müssen durch den Benutzer durchgeführt werden.

Die verbleibenden *Aufgaben-Typen* (Tabelle 2 e-l) können auf einen abstrakten Task zurückgeführt werden, da Asbru nur die Ausführung einer Aktivität durch einen Benutzer vorsieht. Dieser Benutzer weiß, wie er die Aufgabe löst und benötigt dafür keine zusätzliche Information, wie sie in den Aufgaben-Typen enthalten ist.
Daher werden Tasks, mit der Mehrfachausführungs- (Tabelle 2 c) bzw. Kompensationsmarkierung (Tabelle 2 d) und die noch verbleibenden *Aufgaben-Typen* (Tabelle 2 g-l) wie eine abstrakter Task umgesetzt [6], [12].

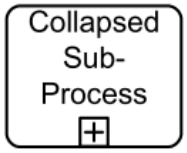
Nachdem für BPMN-Tasks gleichwertige Asbru-Konstrukte gefunden wurden, werden im folgenden Abschnitt BPMN-Subprozesse untersucht.

4.2.2 Sub-Processes

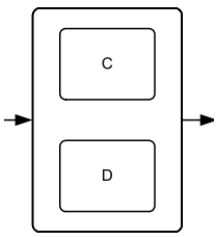
Subprozesse (Sub-Processes) teilen einen Prozess in kleinere Teilprozesse bzw. lassen sich damit Prozesse aus Teilprozessen zusammenstellen. Dadurch können Teile eines Prozess wiederverwendet werden, aber es lassen sich damit auch Details verbergen, um so die Übersichtlichkeit eines Prozesses zu erhöhen. Es gilt die Faustregel, nicht mehr als 50 Elemente in einem Prozessdiagramm darzustellen, da sonst die Lesbarkeit darunter leidet [6], [13].
Tabelle 3 listet BPMN-Subprozess-Elemente und deren Umsetzung in Asbru auf.

Tabelle 3: Mapping von BPMN Sub-Processes zu Asbru. Symbole aus [6]. Spalte 3 ist geteilt, wobei oben die BPMN-XML Repräsentation und unten die dazugehörige Asbru-Umsetztung zu finden ist.

	Grafisches BPMN-Element	BPMN-XML bzw. Asbru-Umsetzung

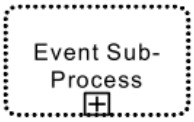
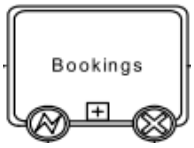
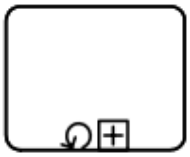
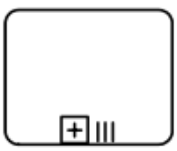
a	Subprozess[5]
	 
		`<subProcess id="Subprozess1" name="Subprozess1"`
		`triggeredByEvent="false"> ...`
		`</subProcess>`

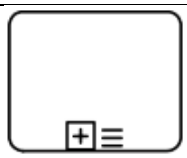
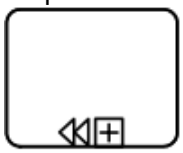
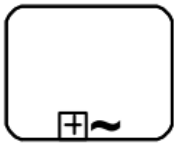
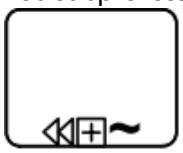
		`<plan name="Subprozess1" title="Subprozess1">`
		`<plan-body>`
		`<subplans ...> ...`
		`<!-- enthält Subprozess-->`
		`</subplans>`
		`</plan-body>`
		`</plan>`

b	Parallel Box
	 
		`<subProcess id="ParallelBox" name="ParallelBox"`
		`triggeredByEvent="false">`
		`<task id="C" name="C"/>`
		`<task id="D" name="D"/>`
		`</subProcess>`

		`<plan name="ParallelBox" title=" ParallelBox">`
		`<plan-body>`
		`<subplans type="parallel">`
		`<wait-for><all/></wait-for>`
		`<plan-activation> <plan-schema name="C" />`
		`</plan-activation>`

[5] Die Repräsentation eines Subprozesses ist unabhängig von der grafischen Darstellung, d.h. die BPMN XML-Darstellung eines *subProcess* Elements ist unabhängig davon, ob das Element grafisch zusammengefaltet (collapsed) dargestellt wird oder nicht.

		<plan-activation> <plan-schema name="D" /> <plan-activation> </subplans> </plan-body> </plan>
c	Ereignis Subprozess 	<subProcess id="sub1" name="sub1" triggeredByEvent="true"> ... </subProcess>
		Wird von Asbru nicht unterstützt, jedoch auf einen einfachen Subprozess reduziert. Durch einen Kommentar wird auf den Unterschied hingewiesen.
d	Transaktion 	<transaction id="sub1" name="sub1" method="##Compensate"> ... </transaction>
		Wird von Asbru nicht unterstützt, jedoch auf einen einfachen Subprozess reduziert. Durch einen Kommentar wird auf den Unterschied hingewiesen.
e	Schleifen Subprozess 	<subProcess id="Schleife" name="Schleife" triggeredByEvent="false"> < standardLoopCharacteristics testBefore="true> < loopCondition>a==1</model:loopCondition> </ standardLoopCharacteristics> ... </subProcess>
		<plan name="Schleife"> <plan-body> <iterative-plan> <do-repeatedly> <plan-activation> <plan-schema name=" Subprozess" /> </plan-activation> </do-repeatedly> <termination-condition>...</termination-condition> </iterative-plan> </plan-body> </plan> ... <plan name="Subprozess1" title="Subprozess1"> <plan-body>... <!-- enthält Subprozess --> </plan-body> </plan>
f	Mehrfachausführung Subprozess parallel  Sequentiel	<subProcess id="sub1" name="sub1" triggeredByEvent="false"> <MultiInstanceLoopCharacteristics isSequential="[true|false]" behavior="All"> <loopCardinality>5</loopCardinality> <completionCondition>b==c<completionCondition> </MultiInstanceLoopCharacteristics> ... </subProcess>
		Wird von Asbru nicht unterstützt, jedoch auf einen einfachen Subprozess reduziert. Durch einen Kommentar wird auf den Unterschied

		*hingewiesen.*
g	Kompensation Subprozess 	*<subProcess id="sub1" name="sub1"* *triggeredByEvent="false" isForCompensation="true">* *...* *</subProcess>*
		Siehe (a) Subprozess
h	Ad-Hoc Subprozess 	*<adHocSubProcess id="AdHoc" name="AdHoc"* *ordering="[Parallel|Sequential]" cancelRemainingInstances="true">* *<completionCondition>b==3</completionCondition>* *<task id="A" name="A"/>* *...* *</adHocSubProcess>*
		<plan name="AdHoc" title="AdHoc"> *<plan-body>* *<subplans type="[unordered|any-order]"* *wait-for-optional-subplans="yes">* *<wait-for><one/></wait-for>* *<plan-activation> <plan-schema name="A" />* *</plan-activation>* *...* *</subplans>* *</plan-body>* *</plan>* *referenzierter Plan:* *<plan name="A">* *<conditions>* *<filter-precondition confirmation-required="yes" > ...* *</filter-precondition>* *</conditions>* *...* *</plan>...*
i	Kompensation/Ad-Hoc Subprozess 	*<adHocSubProcess id="adhoc1" name="adhoc1"* *isForCompensation="true" ordering="[Parallel|Sequential]"* *cancelRemainingInstances="true">* *<completionCondition>b==3</completionCondition>* *...* *</adHocSubProcess>*
		Siehe (h) Ad-Hoc Subprozess

Der *Subprozess* (Tabelle 3 a) referenziert auf einen Teilprozess und aktiviert ihn, wenn der Subprozess durch ein Kontrollflusstoken aktiviert wird. Man kann ihn sich wie einen Funktionsaufruf in einer allgemeinen Programmiersprache vorstellen.
Asbru setzt einen Subprozess ähnlich wie einen abstrakten Task (Tabelle 2 a) um. D.h. es wird ein Plan (Plan Subprozess1, Tabelle 3 a oben) für das *subprocess* Element erzeugt, jedoch enthält dessen *plan-body* Element weitere Asbru-Pläne, die über ein *subplans* Element referenziert werden und den

Subprozess repräsentieren. Wie auf jeden anderen Asbru-Plan auch kann mit einem *plan-activation* Element auf diesen Plan referenziert werden (Tabelle 2 a unten) [6], [12].

Enthält ein Subprozess ausschließlich Tasks (Tabelle 3 b), die jedoch durch keine Kontrollflusskanten miteinander verbunden sind, dann spricht man von einer *parallelen Box*. D.h. alle Tasks im Subprozess werden parallel ausgeführt und der Subprozess wird erfolgreich beendet, wenn alle Subtasks erfolgreich beendet wurden. Somit entspricht dieses Konstrukt einem parallelem Block (AND-Block, Parallel Split Pattern in Kombination mit dem Synchronization Pattern).
Asbru kann dies mit einem Plan (Plan ParallelBox, Tabelle 3 b) realisieren, der ein *subplan* Element, dessen *type* Attribut den Wert „parallel“ hat, realisieren, in dem auf die entsprechenden Sub-Pläne referenziert wird. Außerdem muss die Fortsetzungsbedingung (wait-for=all) entsprechend gesetzt sein. Dadurch wird das gewünschte Verhalten erzielt [6], [12].

Der *Schleifen Subprozess* unterscheidet sich vom Schleifen Task dadurch, dass kein einzelner Task bei jeder Iteration aufgerufen wird, sondern ein ganzer Teilprozess.
In Asbru wird dies mit zwei Teilen umgesetzt. Zuerst wird ein Plan (Plan Subprozess1, Tabelle 3 e unten) erstellt, der in seinem *plan-body* Element den aufzurufenden Teilprozess enthält. Mit dem zweiten Plan (Plan Schleife, Tabelle 3 e oben) wird die Schleife mit dem *iterative-plan* Element realisiert, die bei jeder Iteration den Plan Subprozess aktiviert [6], [12].

Die Idee des *Ad-Hoc Subprozesses* (Tabelle 3 h) ist, dass mehrere Tasks zur Auswahl stehen und der User entscheidet, welche Tasks und wie oft sie ausgeführt werden. Zu unterscheiden ist dabei die sequentielle Abarbeitung (*ordering=“Sequential“* Attribut), bei der immer nur eine Aktivität zur selben Zeit abgearbeitet wird, und die parallele Abarbeitung (*ordering=“Parallel“* Attribut), bei der auch mehrere Aktivitäten gleichzeitig abgearbeitet werden können. Jedes Mal wenn eine einzelne Subaktivität beendet wurde, wird die optional spezifizierte Beendigungsbedingung (*completionCondition* Element) überprüft, und der Ad-Hoc Subprozess beendet, wenn sie erfüllt ist.
Diese Semantik kann in Asbru nicht vollständig ausgedrückt werden.
In Asbru kann, lt. Spezifikation, ausgedrückt werden, dass ein, mehrere oder alle Subtask(s) ein Mal oder öfter in beliebiger Reihenfolge ausgeführt werden kann bzw. können. Subpläne die in einem *subplans* Element referenziert werden, könnten auch mehrmals aktiviert werden, bevor der referenzierende übergeordnete Plan beendet wird. Hier ist die Asbru-Referenz nicht eindeutig. Es steht weder explizit, dass ein einzelner Subtask mehrmals ausgeführt werden kann, noch dass es nicht möglich ist. Der Asbru-Interpreter kann einen Subplan nicht mehrmals ausführen. Dies könnte jedoch bei einer späteren Interpreter Version geändert werden.
In dieser Arbeit wird folgende Variante verwendet und umgesetzt:
Mit Asbru kann der Ad-Hoc Subprozess durch einen Plan (Plan *AdHoc*, Tabelle 3 h, Asbru-Teil oben) mit einem *subplans* Element realisieren werden, dessen *type* Attribut den Wert „any-order“ (sequentielle Abarbeitung) bzw. „unorderd“ (parallele Abarbeitung) hat. Auf die erfolgreiche Beendigung optionaler Subpläne wird abgewartet (*wait-for-optional-subplans*="yes" Attribut). Weiters ist das *wait-for* Element mit dem *one* Element erforderlich. Dadurch wird festgelegt, dass der Subprozess nach Beendigen eines Subtasks beendet werden kann, auch etwaige andere optionale Pläne beendet werden können und die Aktivierungsreihenfolge der Subtasks beliebig ist. Die referenzierten Asbru-Pläne (z.B. Plan A, Tabelle 3 h, Asbru-Teil unten) müssen zur Aktivierung durch den Benutzer bestätigt (confirmation-required="yes") werden und simulieren so die Auswahl des relevanten Subtasks [6], [12].

Der Ereignis Subprozess, die Transaktion sowie der Mehrfachausführung Subprozess werden von Asbru nicht unterstützt und können deshalb nicht übersetzt werden. Diese Elemente werden jedoch als einfaches *subProcess* Element übersetzt, und es wird in einem Kommentar darauf hingewiesen, dass es sich dabei eigentlich um das korrespondierende ursprüngliche Element handelt.

Nach den Subprozessen wird im kommenden Abschnitt das Mapping von Ereignissen (Events) besprochen.

4.2.3 Events

Ereignisse (Events) zeigen an, dass während der Prozess ausgeführt wurde, „etwas passiert“ ist. Es könnten also Ereignisse eingetreten sein, die zur Fortführung des Prozesses notwendig sind oder es könnten Ereignisse auftreten, auf die durch einen alternativen Prozessablauf reagiert werden soll. Z.B. wird mit Ereignissen in BPMN ausgedrückt, dass eine bestimmte Nachricht vorhanden sein muss, um den Prozess fortzusetzen, oder wenn ein Task abgebrochen wird, soll nicht der normale sondern ein alternativer Ablauf ausgeführt werden [6].

In BPMN können drei Kategorien von Ereignissen unterschieden werden [6]:

- Startereignisse (Start Events): Ereignisse die bei Eintreten einen Prozessablauf starten.
- Zwischenereignisse (Intermediate Events): Ereignisse, die einen Prozess unterbrechen bis ein anderes Ereignis eintrifft oder geworfen wird.
- Endereignisse (End Events): Ereignisse, die einen Prozessablauf beenden, nachdem ein Ereignis geworfen wurde.

Des Weiteren kann man unterscheiden, ob ein Ereignis *konsumiert*, also der Kontrollfluss an dieser Stelle aktiviert wird, wenn an andere Stelle das entsprechende Ereignis geworfen wurde, oder ob es *geworfen* wird und somit den Kontrollfluss an entsprechend andere Stelle aktiviert, indem es ein entsprechendes Ereignis wirft. Startereignisse können Ereignisse nur konsumieren, während Endereignisse Ereignisse nur werfen können. Bei Zwischenereignissen ist beides möglich.
Ereignisse haben einen Ereignis-Typ und können optional mit einem Namen versehen werden. Wird ein Ereignis geworfen, so werden anhand des Ereignis-Typs bzw. in Verbindung mit dem Namen die entsprechenden konsumierenden Ereignisse aktiviert, und der Kontrollfluss an diesen Stellen weitergeführt.
BPMN sieht zur Darstellung der Ereignisse sechs XML-Elemente vor.
Die werfenden Ereignisse sind das implizit geworfene Ereignis (*implicitThrowEvent* Element), das Endereignis (*endEvent* Element) und das werfende Zwischenereignis (*intermediateThrowEvent* Element). Die konsumierenden Ereignisse sind das konsumierende Zwischenereignis (*intermediateCatchEvent* Element), das Startereignis (*startEvent* Element) und das anhaftende Ereignis (*boundaryEvent* Element).
Zum genaueren Spezifizieren der o.a. Ereignisse gibt es insgesamt 13 verschiedene Ereignis-Typen in BPMN, die über spezielle Ereignisdefinitionselemente (Event Definitions) unterschieden werden. Näheres dazu siehe [6].
In diesem Abschnitt werden zur Vereinfachung der Implementierung nur Start-, End- und Zwischenereignisse behandelt, und hier auch nur die konsumierenden und werfenden Varianten. Anhaftende Ereignisse und Ereignisse, die einen Ereignis Subprozess starten, werden hier nicht berücksichtigt.

Wie vorhin erwähnt gibt es insgesamt 13 verschiedene Ereignis-Typen in BPMN. Einige dieser Ereignisse können mit Asbru simuliert werden, einige nicht. Es können auch nicht alle Ereignisse auf dieselbe Art umgesetzt werden. Die folgenden Abschnitte fassen jene Ereignisse zusammen, die mit demselben Asbru-Konstrukt abgebildet werden und erläutern diese.

4.2.3.1 Basis Ereignisse

Asbru besitzt keinen Mechanismus, der das Werfen und Konsumieren von Ereignissen unterstützt. Einige Ereignistypen können jedoch mit boolschen Variablen simuliert werden. D.h. im Asbru Prozess

wird für jedes geworfene Ereignis eine boolsche Variable definiert. Wird das Ereignis geworfen, wird die entsprechende Variable auf den Wert „true“ gesetzt und aktiviert jenen Plan, der das konsumierende Gegenstück repräsentiert. Dadurch wird der Kontrollfluss im richtigen Prozessteil weiter ausgeführt.

```
<plan name="EventKonsumieren">
  <conditions>
   <filter-precondition>
     <simple-condition>
       <comparison type="equal">
         <left-hand-side>
           <variable-ref name="eventVar1" />
         </left-hand-side>
         <right-hand-side>
           <qualitative-constant value="true"/>
         </right-hand-side>
       </comparison>
     </simple-condition>
   </filter-precondition>
  </conditions>
  <plan-body>
        <!-- rücksetzen des Events -->
    <variable-assignment variable="eventVar1">
      <qualitative-constant value="false" />
    </variable-assignment>
  </plan-body>
</plan>
```

Abb. 39: Asbru-Code für ein mit Variablen simuliertes konsumierendes Ereignis

```
<plan name="EventWerfen">
  ...
  <plan-body>
        <!-- Event werfen -->
    <variable-assignment variable="eventVar1">
      <qualitative-constant value="true" />
    </variable-assignment>
  </plan-body>
</plan>
```

Abb. 40: Asbru-Code für ein mit Variablen simuliertes werfendes Ereignis

Abb. 39 zeigt einen Plan (EventKonsumieren), der erst unter der Bedingung aktiviert wird, wenn die Variable eventVar1 den Wert „true“ hat. Der Plan an sich setzt bei Aktivierung die Variable eventVar1 wieder auf „false“ (variable-assignment) zurück und konsumiert so das Ereignis. Ein solcher Plan ist entsprechend, meist in einem sequentiellen Ablauf, in den Kontrollfluss eingebunden.

Abb. 40 zeigt einen Plan (EventWerfen), der ein Ereignis wirft, indem er die Variable eventVar1 auf den Wert „true“ setzt.
Tabelle 4 zeigt jene Ereignis-Typen, die mit einer Variablen simuliert werden können.

Tabelle 4: Ereignis-Typen, die mit einer boolschen Variable simuliert werden können. Symbole aus [6]

	Ereignis-Typ	Startereignis	Zwischenereignis konsumierend	Zwischenereignis werfend	Endereignis
	Eskalation				
	Kompensation				
	Signal				
	Mehrfach				

Diese Realisierung ist theoretisch möglich, konnte jedoch praktisch nicht verifiziert werden, da der Interpreter die Wertezuweisung von Variablen nicht unterstützt. Für Details sei auf Abschnitt 4.3.4 verwiesen.

4.2.3.2 Fehler und Abbruch Ereignisse

Anhaftende Ereignisse werden an eine Aktivität bzw. einen Subprozess angehaftet und signalisieren, dass im Fehlerfall bzw. wenn der Subprozess abgebrochen wird, ein alternativer Prozessablauf eingeschlagen werden soll.
Das konsumierende Fehler-Zwischenereignis kann an Tasks und Subprozesse angehaftet werden, Abbruch-Zwischenereignis nur an Transaktionssubprozesse.
Wie schon in Abschnitt 4.2.2 angeführt wurde, werden zur Vereinfachung des Transformationsprozesses Transaktionssubprozesse nicht berücksichtigt. Somit ist es auch nicht sinnvoll, Abbruch-Ereignisse zu berücksichtigen.

Das *Fehler-Ereignis* kann in Asbru, ähnlich wie bei den Basis Ereignissen, mit boolschen Variablen simuliert werden. Das Werfen der Ereignisse wird gleich wie in Abb. 40 gezeigt durchgeführt.

Das konsumierende Fehler-Ereignis (Abb. 41), das z.B. an einem Subprozess angehaftet wird, kann mit einer Abbruchbedingung (abort-Condition, Abb. 41 unten) realisiert werden. D.h. der Plan der den Subprozess repräsentiert (Plan FehlerAbbruch), wird mit einer Abbruchbedingung versehen, die den Subprozess abbricht, wenn Variable eventVar1 true wird. Die Variable wird innerhalb des Subprozesses im Fehlerfall auf den Wert „true“ gesetzt und bricht ihn ab. Der alternative Prozessablauf wird im *plan-activation* Element (Abb. 41 oben) mit der Referenz auf den alternativen Plan (Plan AlternativerAblauf) im *on-abort* Element spezifiziert. Dadurch wird Plan AlternativerAblauf ausgeführt, wenn der Plan FehlerAbbruch abgebrochen wurde.

Tabelle 5: Anhaftende Ereignis-Typen, die mit einer boolschen Variable simuliert werden können. Symbole aus [6]

	Ereignis-Typ	Startereignis	Zwischenereignis konsumierend	Zwischenereignis werfend	Endereignis
a	Fehler	-		-	
b	Abbruch	-		-	

```
Referenz mit alternativem Kontrollfluss im Fehlerfall:
<plan-activation>
  <plan-schema name="FehlerAbbruch" />
  <on-abort>
    <plan-activation>
      <plan-schema name="AlternativerAblauf" />
    </plan-activation>
  </on-abort>
</plan-activation>

Prozess mit Abbruchbedingung:
<plan name=" FehlerAbbruch ">
  <conditions>
    <abort-condition>
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <variable-ref name="eventVar1" />
          </left-hand-side>
          <right-hand-side>
            <qualitative-constant value="true" />
          </right-hand-side>
        </comparison>
      </simple-condition>
    </abort-condition>
  </conditions>
  <plan-body>...
        <!—hier wird  eventVar1 eventuell true-->
  </plan-body>
</plan>
```

Abb. 41: Asbru-Code eines Subprozesses mit alternativem Ablauf, wenn Abbruchbedingung erfüllt ist.

Auch hier ist die Realisierung dieser Lösung auf Grund der fehlenden Interpreter-Unterstützung nur theoretisch umsetzbar.

4.2.3.3 Nachricht und Bedingung Ereignisse

Ereignisse vom Typ *Nachricht* sind Ereignisse, die durch den Erhalt einer Nachricht von einem externen Prozessteilnehmer ausgelöst (konsumierendes Ereignis) werden bzw. das Verschicken einer Nachricht an einen externen Teilnehmer auslösen (werfend).

Ereignisse vom Typ *Bedingung* gibt es nur in der konsumierenden Variante. Sie werden durch ein externes System ausgelöst, das bestimmte Bedingungen (z.B. Geschäftsregeln) überwacht und dessen Auftreten oder Nichterfüllung eines Ereignisses meldet.
Da bei den beiden Ereignis-Typen das Ereignis aus einem anderen System als dem eigenen Prozess entspringt bzw. eine Nachricht an einen externen Teilnehmer verschickt wird, kann dies nicht durch eine boolsche Variable simuliert werden. Bei den Ereignis-Typen in den vorangegangenen Abschnitten ist dies möglich, da Werfen und Konsumieren des Ereignisses im eigenen Prozess stattfindet.

Mit Asbru kann dies mit Hilfe von Plänen simuliert werden, deren Filterbedingung und somit deren Aktivierung durch manuelle Bestätigung durch den Benutzer erfolgt. Somit bestimmt der Benutzer ob

eine Nachricht eintrifft oder eine Bedingung erfüllt ist, und der Ablauf startet bzw. fortgesetzt wird oder eine Nachricht versendet wurde.

```
<plan name="A">
  <conditions>
    <filter-precondition confirmation-required="yes" >
...<!-- beliebige Tautologie -->
    </filter-precondition>
  </conditions>
  ...
</plan>
```

Abb. 42: Asbru Plan mit manueller Aktivierung

Dies wird in Asbru mit dem *confirmation-required* Attribut im *filter-precondition* Element ausgedrückt (Abb. 42). Die Intention dahinter ist, dass der Benutzer bestätigt, die Nachricht versendet bzw. empfangen zu haben, bevor der nachfolgende Task ausgeführt wird.
Tabelle 6 zeigt die grafische Darstellung jener Ereignis-Typen, die über eine manuelle Planaktivierung in Asbru dargestellt werden.

Tabelle 6: Ereignis-Typen, die mit einer manuellen Planaktivierung repräsentiert werden. Symbole aus [6]

	Ereignis-Typ	Startereignis	Zwischenereignis konsumierend	Zwischenereignis werfend	Endereignis
	Nachricht				
	Bedingung			-	-

```
Referenz auf Plan Link1:
<plan-activation>
  <plan-schema name="Link1" />
</plan-activation>

Plan Link1:
<plan name="Link1" >
  <plan-body>
    <subplans ...>
      <!—enthält weiterführenden Prozessteil -->
    ... </subplans>
  </plan-body>
</plan>
```

Abb. 43: Asbru-Code eines Subprozessaufrufes

4.2.3.4 Link

Ein Link Ereignis wird verwendet, um einen normalen Prozessablauf an einer anderen Stelle fortzusetzten. D.h. tritt im Prozessablauf eine werfendes Link-Ereignis auf, dann wird zu dem dazugehörigen konsumierenden Link-Ereignis gesprungen, und der Prozessablauf dort fortgesetzt.
Daraus folgt auch, dass es mindestens ein werfendes und genau ein konsumierendes Ereignis für einen korrespondierenden Link geben muss.

Dies ist durchaus vergleichbar mit einem Subprozessaufruf am Ende eines Prozessstrangs, der auf den folgenden Teilprozess verweist.

Asbru setzt den Link wie einen Subprozess um. D.h. es wird ein Plan (Plan Link1, Abb. 43 unten) für den aufgerufenen Prozessteil erzeugt, dessen *plan-body* Element weitere Asbru-Pläne enthält, die in einem *subplans* Element referenziert werden und den weiterführenden Prozessteil repräsentieren. Referenziert wird auf den Plan mit dem *plan-activation* Element (Abb. 43 oben).

Tabelle 7 zeigt die grafischen Symbole, die für einen Link-Event verwendet werden.

Tabelle 7: BPMN Link Ereignis-Typ. Symbole aus [6]

	Ereignis-Typ	Startereignis	Zwischenereignis konsumierend	Zwischenereignis Werfend	Endereignis
	Link	-		-	

4.2.3.5 Weitere Ereignisse

Die noch verbleibenden Ereignis-Typen werden in diesem Unterabschnitt behandelt.

Die Blanko Ereignis-Typen (Tabelle 8 a) haben keinen speziellen Auslöser. Sie indizieren den Start bzw. Endzustand eines Prozessablaufs. Prozesse, die einen Blanko-Ereignis Startzustand haben, werden durch einen expliziten Aufruf (Call Activity) aus einem anderen Prozess oder eines Subprozessaufrufes gestartet.
Start und Endzustände werden in Asbru implizit durch den Kontrollfluss ausgedrückt, der durch die Verschachtelung der Pläne entsteht.
Der Zwischenereignistyp wird durch einen Aspru-Plan dargestellt, dessen Aktivierung durch den Benützer manuell erfolgt (Abb. 42) [6], [12].

Der Ereignis-Typ Terminierung (Tabelle 8 b) wird verwendet, um einen Subprozess oder Prozess abzubrechen. In Asbru kann dies über die Abbruchbedingung (abort-condition, siehe Abb. 44) des Plans, der den (Sub-)Prozess repräsentiert, umgesetzt werden.

Timer-Ereignisse stellen zeitliche Ereignisse dar, die erreicht wurden, und dadurch einen Prozess starten oder reaktivieren. Das kann z.B. ein Zeitpunkt, der erreicht wurde oder ein periodisches Zeitmuster, das erfüllt wurde, sein. Da der Fokus dieser Arbeit auf der Transformation des Kontrollflusses liegt, wird die Transformation zeitlicher Aspekte hier nicht berücksichtigt.

```
Subprozess mit Terminierungs-Ereignissen:
<plan name=" TerminierungProzess ">
  <conditions>
    <abort-condition>
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <variable-ref name="TermVar1" />
          </left-hand-side>
          <right-hand-side>
            <qualitative-constant value="true" />
          </right-hand-side>
        </comparison>
      </simple-condition>
    </abort-condition>
  </conditions>
  <plan-body>...
        <!—hier wird  TermVar1 eventuell true-->
  </plan-body>
</plan>
```

Abb. 44: Asbru-Code eines Subprozesses mit Terminierungs-Ereignissen

Tabelle 8 zeigt die grafische Darstellung der in diesem Abschnitt erörterten Ereignis-Typen.

Tabelle 8: a) Ereignis-Typ ohne spezifischen Auslöser, b) Terminierungs-Ereignis und c) Timer-Ereignis. Symbole aus [6]

	Ereignis-Typ	Startereignis	Zwischenereignis konsumierend	Zwischenereignis werfend	Endereignis
a	Blanko		-		
b	Terminierung	-	-	-	
c	Timer			-	-

4.2.4 Gateways

Gateways sind BPMN-Konstrukte, die den Kontrollfluss verzweigen bzw. vereinigen und so parallele, exklusiv- und inklusiv-bedingte Ablaufzweige ermöglichen. Wird der Kontrollfluss verzweigt, hat das Gateway eine Eingangskante und mindestens zwei Ausgangskanten. Wird der Kontrollfluss vereinigt, besitzt das Gateway mindestens zwei Eingangskanten und eine Ausgangskante [6].
Asbru folgt zur Repräsentation des Kontrollflusses dem blockorientierten Paradigma. Daraus folgt, dass nur Blöcke ausgedrückt werden können, die folgende Verzweigungs-Vereinigungs-Paare bilden [12]:
AND-Verzweigung /AND-Vereinigung (Tabelle 9 a),
XOR-Verzweigung/XOR-Vereinigung (Tabelle 9 b),
OR-Verzweigung/OR-Vereinigung (Tabelle 9 c),

Komplexe-Verzweigung/XOR-Vereinigung (Tabelle 9 d),
Ereignis-basierte-Verzweigung/XOR-Vereinigung (Tabelle 9 e).

In diesem Abschnitt werden deshalb auch nur diese Blockkonstellationen erörtert.

Tabelle 9: Mapping von BPMN-Gateways zu Asbru. Symbole aus [6]. Spalte 3 ist geteilt, wobei oben die BPMN-XML Repräsentation und unten die dazugehörige Asbru-Umsetzung zu finden ist.

	BPMN-Element	BPMN-Xml bzw. Asbru-Umsetzung
a	Paralleles Gateway 	`<parallelGateway id="g1" name="g1"/>`

		`<plan name="AndBlock" title="AndBlock">`
		`  <plan-body>`
		`  <subplans type="parallel">`
		`  <wait-for><all/></wait-for>`
		`  <plan-activation> <plan-schema name="..." />`
		`  </plan-activation>`
		`  <plan-activation> <plan-schema name="..." />`
		`  </plan-activation>`
		`  ...`
		`  </subplans>`
		`  </plan-body>`
		`</plan>`

b	Exklusives Gateway  o. 	`<exclusiveGateway id="g1" name="g1"/>`

		`<plan name="XORBlock" title=" XORBlock ">`
		`  <plan-body>`
		`  <subplans type="any-order">`
		`  <wait-for><one/></wait-for>`
		`  <plan-activation> <plan-schema name="A" />`
		`  </plan-activation>`
		`  <plan-activation> <plan-schema name="..." />`
		`  </plan-activation>`
		`  ...`
		`  </subplans>`
		`  </plan-body>`
		`</plan>`
		
		`referenzierter Plan:`
		`<plan name="A">`
		`  <conditions>`
		`  <filter-precondition > ...`
		`  <!-- exklusive Verzweigungsbedingung für Zweig A-->`
		`  </filter-precondition>`
		`  </conditions>`
		`  ...`
		`</plan>`
		`...`

c	Inklusives Gateway 	`<inclusiveGateway id="g1" name="g1"/>`

		`<plan name="ORBlock" title=" ORBlock ">`
		`  ...`
		`  <plan-body>`

		<subplans type="unordered" wait-for-optional-subplans="yes"> <wait-for> <one /> </wait-for> <if-then-else> <simple-condition> ... <!-- Verzweigungsbedingung 1 --> </simple-condition> <then-branch> <plan-activation> <plan-schema name="Plan_A" /> </plan-activation> </then-branch> </if-then-else> <if-then-else> <simple-condition> ... <!-- Verzweigungsbedingung 2 --> </simple-condition> <then-branch> <plan-activation> <plan-schema name="Plan_B" /> </plan-activation> </then-branch> </if-then-else> ... </subplans> </plan-body> </plan>
d	Komplexe Gateway 	<complexGateway id="g1" name="g1"> <activationCondition>...<!-- Verzweigungsbedingung --> <activationCondition> </complexGateway>
		Siehe (c) inklusives Gateway
e	Ereignis-basiertes Gateway 	<eventBasedGateway id="EventBlock" name="EventBlock" instantiate="false" eventGatewayType="[Exclusive|Parallel]"/>
		<plan name="EventBlock" title=" EventBlock "> <plan-body> <subplans type="any-order"> <wait-for><one/></wait-for> <plan-activation> <plan-schema name="A" /> </plan-activation> <plan-activation> <plan-schema name="B" /> </plan-activation> ... </subplans> </plan-body> </plan> referenzierter Plan: <plan name="A">

```
  <conditions>
    <filter-precondition confirmation-required="yes" > ...
    </filter-precondition>
  </conditions>
  ...
</plan>

<plan name="B">
  <conditions>
    <filter-precondition>
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <variable-ref name="eventVar1" />
          </left-hand-side>
          <right-hand-side>
            <qualitative-constant value="true" />
          </right-hand-side>
        </comparison>
      </simple-condition>
    </filter-precondition>
  </conditions>
  ...
</plan>
```

Beim *parallelen Gateway* (verzweigend) (AND-Block, Tabelle 9 a) werden alle Kontrollflusszweige parallel aktiviert und am Sychronisationspunkt (paralleles Gatway vereinigend) wird auf die Beendigung aller Zweige gewartet, um den Kontrollfluss weiterzuführen.
Asbru kann dies mit einem Plan (Plan AndBlock, Tabelle 9 a), der ein *subplans* Element, dessen *type* Attribut den Wert „parallel“ hat, realisieren, in dem auf die entsprechenden Sub-Pläne referenziert wird. Die Fortsetzungsbedingung (*wait-for* Element) muss derart konfiguriert sein, dass auf die Beendigung aller Teilpläne (*all* Element) gewartet wird. Dadurch wird das gewünschte Verhalten erzielt.

Beim *exklusiven Gateway* (verzeigend) (XOR-Block, Tabelle 9 b) werden die Verzweigungsbedingungen der ausgehenden Kanten evaluiert, und genau ein Zweig wird aktiviert. Am Synchronisationspunkt (exklusiven Gateway vereinigend) wird auf die Beendung eines Zweiges gewartet, um den Kontrollfluss weiterzuführen.
Asbru bietet zwei Varianten (siehe auch Abschnitt 4.1.1.4) dafür an. Hier wird nur die Variante mit exklusiven Filterbedingungen beschrieben (Tabelle 9 b).
D.h. es wird ein Plan (Plan XORBlock, Tabelle 9 b, oben), der ein *subplans* Element, dessen *type* Attribut den Wert „any-order“ hat, erstellt, und die Fortsetzungsbedingung (*wait-for* Element) derart konfiguriert, dass auf die Beendigung eines einzigen Plans (*one* Element) gewartet wird. Somit können alle referenzierten Teilpläne ausgeführt werden, und wird einer davon beendet, wird auch der Plan XORBlock beendet.
Die Filterbedingung der Subpläne (Plan A, Tabelle 9 b, unten) müssen dabei so beschaffen sein, dass jeweils nur ein einzelner Zweig ausgeführt werden kann (nicht überlappend).

Das *inklusive Gateway* (verzweigend) (OR-Block, Tabelle 9 c) wird verwendet, um den Kontrollfluss auf einen Zweig, mehrere oder alle Zweige, entsprechend den Verzweigungsbedingungen, zu verteilen.
Am Synchronisationspunkt (inklusives Gateway vereinigend) wird auf die Beendigung aller aktivierten Zweige gewartet, um den Kontrollfluss weiterzuführen.

Dies kann in Asbru mit zwei Lösungsansätzen realisiert werden, nämlich mit *if-then-else* Elementen oder mit einem „unordered“ Plan-Typ mit überlappenden Filterbedingungen (siehe Abschnitt 4.1.2.1). Hier wird nur die Variante mit dem *if-then-else* Element erläutert, da sie besser lesbar ist. Sonst sind beide Varianten gleichwertig.
Tabelle 9 c zeigt die relevanten Auszüge des Asbru-Codes für den OR-Block. Zur Realisierung wird ein *subplans* Element mit dem *type* Attribut=“unordered“, dem *wait-for-optional-subplans* Attribute=“yes“ und der Weiterführungsbedingung *(wait-for*=“one“) erstellt. D.h. es können alle Teilpläne (A, B, etc.) parallel gestartet werden, ein beendeter Teilplan reicht aus, um Plan „ORBlock“ zu beenden (*wait-for*=*one*), aber es werden alle optionalen Pläne (*wait-for-optional-subplans*=yes) abgewartet, um fortzufahren. Welche Subpläne ausgeführt werden, hängt von den Verzweigungsbedingungen der *if-then-else* Elemente ab.

Das *komplexe Gateway* wird eingesetzt, wenn Verzweigungsbedingungen durch die bisher genannten Gateways nicht ausgedrückt werden können. Mit Asbru kann dies wie mit dem OR-Block gelöst werden, es müssen nur die Verzweigungsbedingungen entsprechend angepasst werden.

Im Gegensatz zu den bisher genannten Gateways bzw. Blöcken beruht die Aktivierung eines Zweiges beim *Ereignis-basierten Gateway (Event-Block,* Tabelle 9*e)* nicht auf der Evaluierung einer Verzweigungsbedingung. Hier entscheidet das Eintreten eines Ereignisses (Events) darüber, an welchen Zweig das Kontrollflusstoken weitergereicht wird. Am Synchronisationspunkt (exklusives Gateway vereinigend) wird auf die Beendung des einen aktivierten Zweiges gewartet, um den Kontrollfluss weiterzuführen.
Dieses Verhalten in Asbru umzusetzen ist aufwändig, da Events von Asbru nicht unterstützt werden.
Wie schon im vorangegangenen Abschnitt 4.2.3 erläutert, werden Events entweder mit dem manuellen Bestätigen eines Plans (Plan A, Tabelle 9 e), der den Event repräsentiert, oder mit eigens dafür vorgesehenen boolschen Variablen (Variable eventVar1), die Teil der Filterbedingung sind, simuliert (Plan B, Tabelle 9 e). Diese Pläne müssen von einem Plan (Plan EventBlock, Tabelle 9 e) aus referenziert werden, der ein *subplans* Element besitzt, dessen *type* Attribut den Wert „any-order“ hat, und die Fortsetzungsbedingung (*wait-for* Element) derart konfiguriert ist, dass auf die Beendigung eines einzigen Plans (*one* Element) gewartet wird.
Somit wird gewährleistet, dass ein Plan bzw. Zweig, der von einem Ereignis aktiviert wurde, ausgeführt wird, bevor der Kontrollfluss an die Nachfolgeaktivität des Blocks weitergegeben wird.

4.2.5 Einschränkungen

In den vorrausgegangenen Unterabschnitten wurde versucht BPMN-Elementen, die zur Darstellung des Kontrollflusses benötigt werden, Asbru Elemente zuzuordnen, und somit ein Mapping zu definieren. Dies gelang nicht immer, da sich bei manchen Elementen kein Asbru-Äquivalent fand.
In dieser Arbeit sollen die in Abschnitt 0 beschriebenen Muster, die sowohl von BPMN als auch von Asbru unterstützt werden, transformiert werden. D.h. es wird nur das Mapping jener BPMN-Elemente im implementierten Prototyp verwendet, die zur Beschreibung der entsprechenden Kontrollflussmuster benötigt werden.
Dies beschränkt das Mapping auf den abstrakten Task (Tabelle 2a), das exklusive (Tabelle 9b), inklusive (Tabelle 9c), parallele (Tabelle 9a) und Ereignis-basierte (Tabelle 9e) Gateway. Des Weiteren werden Start-, Endzustände (Tabelle 8a), Terminierungsereignis (Tabelle 8b) und das anhaftende Fehler-Zwischenereignis (Tabelle 5a) zur Beschreibung benötigt.
Für die benötigten Elemente ist jeweils das Mapping vorhanden.

4.3 Transformationsstrategie Auswahl

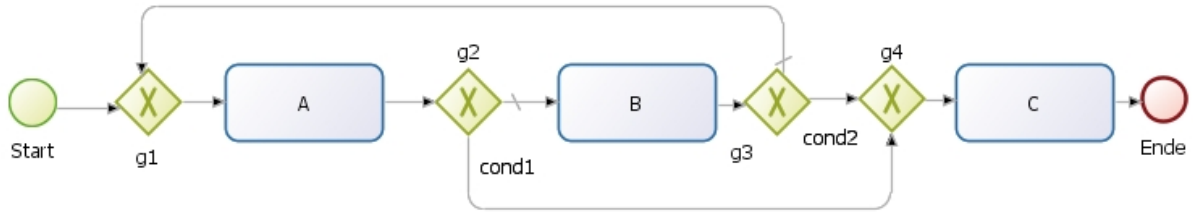
In Abschnitt 3.2.5 wurden Transformationsstrategien vorgestellt, mit denen graforientierte Prozessmodelle in BPEL, stellvertretend für dem blockorientierten Paradigma folgende Prozessmodelle, umgewandelt werden können. In diesem Abschnitt soll nun überprüft werden, ob die o.a. Strategien auch auf den Transformationsprozess zwischen BPMN und Asbru angewandt und wie diese Strategien in Asbru realisiert werden können.
Dadurch soll aus diesen Strategien der geeignete Transformationsansatz ermittelt werden, der in der Implementierung des Prototyps umgesetzt wird.

Im Evaluierungsprozess werden die Structure-Identification, Element-Preservation und Event-Condition-Action-Rule Strategie berücksichtigt. Da es sich bei der Element-Minimization-Strategie und der Structure-Maximization Strategie um Mischformen der vorhin genannten Strategien handelt, werden diese hier nicht überprüft, und das Kombinieren dem Leser überlassen.

Abhängig von der zu bewertenden Transformationsstrategie, werden unterschiedliche Herangehensweisen genutzt:
Bei der Structure-Identification Strategie werden Konstrukte bzw. Teilprozesse in BPMN beschrieben, die in Asbru umgesetzt werden können und so strukturierte Aktivitäten definieren.
Bei der Element-Preservation und der Event-Condition-Action-Rule Strategie wird der Fokus auf die Umsetzung von unstrukturierten Teilprozessen gelegt. Daher wird der jeweilige Ansatz mit Beispielen unstrukturierter Schleifen und Blöcken überprüft, die mit der jeweiligen Strategie realisiert werden. Dabei wird überprüft, ob die Beispiele mit der jeweiligen Strategie umgesetzt werden können, und der entstandene Asbru-Code das gewünschte Verhalten zeigt, wenn er mit dem Asbru Interpreter ausgeführt wird.

4.3.1 Beispiele unstrukturierter Teilprozesse

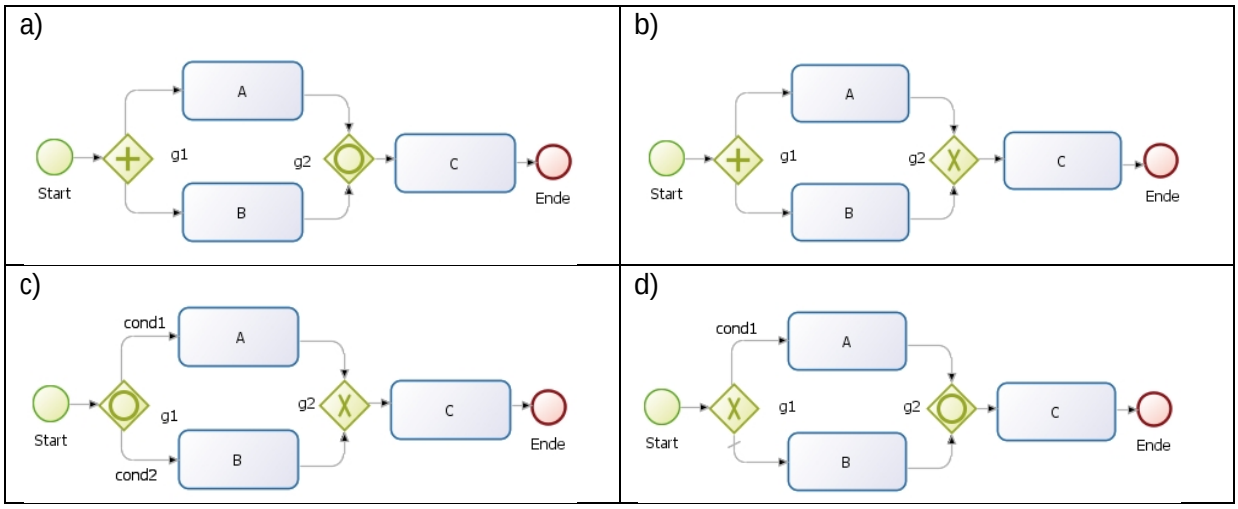
Wie im vorangegangenen Absatz erwähnt, werden Beispiele herangezogen, um zu überprüfen, ob unstrukturierte Schleifen und Blöcke mit dem jeweiligen Ansatz umsetzbar sind. Dabei wird versucht, die folgenden BPMN-Prozesse mit der jeweiligen Strategie umzusetzen.



Beispiel 1: unstrukturierte Schleife mit einem Einstiegspunkt (g1) und zwei Ausstiegspunkten (g2, g3) in den bzw. aus dem Schleifenkörper in BPMN-Notation.

Beispiel 1 zeigt eine unstrukturierte Schleife mit einem Einstiegspunkt (*g1*) in den Schleifenkörper und zwei möglichen Ausstiegspunkten (*g2* und *g3*). Bei der ersten Iteration wird das Kontrollflusstoken vom Startzustand *Start* über das exklusive Gateway *g1* an den Task *A* weitergegeben. Task *A* wird aktiviert und übergibt das Token nach Beendigung an das exklusive Gateway *g2*. Ist die Bedingung *cond1* erfüllt, wird der Schleifenkörper verlassen und das Kontrollflusstoken an Gateway *g4* weitergegeben, das den Task *C* aktiviert und nach dessen Beendigung den Prozess beendet.
Ist die *cond1* nicht erfüllt, wird Task *B* aktiviert und nach dessen Beendigung der Kontrollfluss in Gateway *g3* weitergeführt. Ist hier die Bedingung *cond2* erfüllt, wird der Prozess beendet, nachdem das Token *g4* und Task *C* passiert hat.
Ist *cond2* in *g3* jedoch nicht erfüllt, wird das Token an *g1* weitergeleitet, und eine weitere Iteration des Schleifenkörpers beginnt.

Dies kann mit Asbrus strukturiertem Konstrukt, dem Iterative Plan, nicht ausgedrückt werden, da dies nur einen Ausstiegspunkt aus der Schleife realisiert.



Beispiel 2: unstrukturierte Blöcke mit a) AND-Verzweigung/OR-Vereinigung b) AND-Verzweigung/XOR-Vereinigung c) OR-Verzweigung/XOR-Vereinigung und d) XOR-Verzweigung/OR-Vereinigung.

Beispiel 2 zeigt eine Variation von unstrukturierten Blöcken, d.h. der Typ des verzweigenden Gateways unterscheidet sich von dem des verschmelzenden Gateways. Insgesamt gibt es sechs Kombinationen, unstrukturierte Blöcke zu bilden. Jene Kombinationen, die ein XOR- bzw. ein OR-Gateway zur Verzweigung und ein AND-Gateway als Synchronisationspunkt haben, werden nicht berücksichtigt, da sie einen Deadlock repräsentieren.
Die restlichen vier Kombinationen sind in Beispiel 2a-d abgebildet und beschreiben folgende Semantik:

a) AND-OR Block: Ausgehend vom Startzustand *Start* wird am Verzweigungspunkt *g1* das Kontrollflusstoken parallel an alle ausgehenden Zweige weitergegeben. Dadurch werden die Tasks *A* und *B* aktiviert. Am Synchronisationspunkt *g2* wird das Kontrollflusstoken an die nachfolgende Aktivität *C* weitergegeben, wenn die Token aller eingehenden Zweige eingetroffen sind, somit beide Tasks *A* und *B* beendet wurden.

b) AND-XOR Block: Der Verzweigungspunkt *g1* verhält sich wie unter a). Am Synchronisationspunkt *g2* wird für jedes eingehende Token unsynchronisiert ein ausgehendes Token an die Nachfolgeaktivität *C* weitergegeben. In diesem Beispiel wird Aktivität *C* zwei Mal aktiviert werden, einmal wenn Task *A* und einmal wenn Task *B* beendet wurde, und die Token der jeweiligen Zweige unverzögert und unsynchronisiert an Task *C* weitergegeben wurden.

c) OR-XOR Block: Ausgehend vom Startzustand *Start* wird am Verzweigungspunkt *g1*, abhängig von den Verzweigungsbedingungen, ein Kontrollflusstoken an einen Zweig, mehrere oder alle Zweige weitergeleitetet. Somit könnten, abhängig von den Verzweigungsbedingungen *cond1* und *cond2*, Task *A*, *B* oder beide Tasks aktiviert werden. Am Synchronisationspunkt *g2* wird für jedes eingehende Token unsynchronisiert ein ausgehendes Token an die Nachfolgeaktivität *C* weitergegeben. Daher wird in diesem Beispiel Task *C* aktiviert, sobald *A*, *B* oder beide Tasks beendet wurden.

d) XOR-OR Block: Am Verzweigungspunkt *g1* wird, abhängig von der Verzweigungsbedingung *cond1*, das Kontrollflusstoken an genau einen Zweig weitergeleitet. Ist *cond1* erfüllt, wird Task *A* aktiviert. Ist *cond1* jedoch nicht erfüllt, wird Task *B* aktiviert. Am

Synchronisationspunkt *g2* wird das Kontrollflusstoken an die nachfolgende Aktivität *C* weitergegeben, wenn die Token aller aktivierten Zweige eingetroffen sind. In diesem Beispiel ist dies exakt ein Zweig, der aktiviert wird.

Diese Beispiele dienen als Grundlage für den Evaluierungsprozess. Der Fokus wird bei den Beispielen darauf gelegt, zu überprüfen, ob unstrukturierte Schleifen und Blöcke mit der Strategie umgesetzt werden können.

In den folgenden Unterabschnitten werden die zu evaluierenden Strategien untersucht und deren Umsetzung beschrieben.

4.3.2 Structure-Identification Strategie

Die Structure-Identification Strategie beruht darauf, das graforientierte Eingangsmodell mit Hilfe von Reduktionsregeln zu einem einzigen Knoten zu reduzieren (siehe Abschnitt 3.2.5.3 bzw. [4], [29]).

Es werden strukturierte Aktivitäten definiert, die eine Konstruktion im Eingangsmodell darstellen, die in ein Konstrukt im Zielmodell, also Asbru, umgewandelt werden kann. Dabei wird in jedem Reduktionsschritt ein entsprechender Code für den jeweiligen Teil des Ausgangsmodells erzeugt.
Somit wird bei der Reduktion des graforientierten BPMN Eingangsmodells gleichzeitig die Transformation durchgeführt.

In diesem Abschnitt sollen nun entsprechende Reduktionsregeln festgelegt und formal definiert werden, um so eine Übersetzung eines strukturierten BPMN Eingangsmodells zu ermöglichen d.h. es ist mit diesen Reduktionsregeln vollständig vereinfachbar. Um systematisches Vorgehen zu gewährleisten, werden als Leitrahmen für die Definition der Reduktionsregeln die von Asbru unterstützten Kontrollflussmuster herangezogen.

Zur formalen Definition der Regeln muss zuerst die aus Definition 1 stammende Spezifikation graforientierter Prozessmodelle angepasst werden, um die relevanten BPMN-Konstrukte zur Beschreibung der Kontrollflussmuster darzustellen. Um dies zu erreichen, wird die Spezifikation in Definition 1 mit der Spezifikation eines Core Business Process Diagrams aus [4] verschmolzen und erweitert.

Definition 13: Ein BPMN Process Graph (BPG) ist ein Tupel $BPG = \{O, E^S, E^I, E^E, F, C, l, A, g, t, subref, h\}$, das aus fünf disjunkten Mengen E^S, E^I, E^E, F, C, einer Funktion $l: C \rightarrow \{AND_{Split}, AND_{Join}, OR_{Split}, OR_{Join}, XOR_{Split}, XOR_{Join}, Event_{Split}\}$, einer binären Relation $A \subseteq (E^S \cup E^I \cup F \cup C) \times (E^I \cup E^E \cup F \cup C)$ und einer Funktion $g: A \rightarrow expr$ besteht [29]. Weiters muss gelten [4]:

- O ist die Menge an Knotenobjekten in BPG. $O = E^S \cup E^I \cup E^E \cup F \cup C$
- E^S ist die Menge an Startereignissen. $|E^S| \geq 1$und $\forall s \in E^S: |succ^6(s)| = 1 \wedge |pred^7(s)| = 0$
- E^I ist die Menge an Zwischenereignissen (Intermediate Events). Mit Ausnahme von E^I_F gilt: $\forall i \in E^I: |succ(i)| = 1 \wedge |pred(i)| = 1$. E^I kann in folgende disjunkte Mengen unterteilt werden:

[6] succ(n) gibt die Menge der Nachfolgeknoten eines Knotens n an: $succ(n) = \{x \in O | (n, x) \in A\}$
[7] pred(n) gibt die Menge der Vorgängerknoten eines Knotens n an: $pred(n) = \{x \in O | (x, n) \in A\}$

  - E^I_N Ist die Menge an Nachrichten Zwischenereignissen
  - E^I_T ist die Menge an Timer Zwischenereignissen
  - E^I_F ist die Menge an Fehler Zwischenereignissen, die nur an Tasks und Subprozessen anhaftend auftreten können. Hier gilt $\forall i \in E^I: |succ(i)| = 1 \wedge |pred(i)| = 0$

- E^E ist die Menge der Endereignisse. $|E^E| \geq 1$ und $\forall e \in E^E: |succ(e)| = 0 \wedge |pred(e)| = 1$ E^E kann in zwei disjunkte Teilmengen unterteilt werden: $E^E / (E^E_T \cup E^E_F) \neq \emptyset$
  - E^E_T ist die Menge an Terminierung Endereignissen $E^E_T \subseteq E^E$
  - E^E_F ist die Menge an Fehler Endereignissen $E^E_F \subseteq E^E$
- F ist die Menge an Tasks (Aktivitäten). $\forall f \in F: |succ(f)| = 1 \wedge |pred(f)| = 1$
- C ist die Menge an Konnektoren. $\forall c \in C: |succ(c)| = 1 \wedge |pred(c)| > 1 \vee |succ(c)| > 1 \wedge |pred(c)| = 1$
- Funktion l spezifiziert den Typ eines Konnektors $c \in C$. $l: C \rightarrow \{AND_{Split}, AND_{Join}, OR_{Split}, OR_{Join}, XOR_{Split}, XOR_{Join}, Event_{Split}\}$
- A definiert die Menge der gerichteten Kanten, die die Knoten im Graf verbinden. Es gibt keine Kanten mit identen Start- und Zielknoten (reflexiv) und keine paarweise identen Kanten (mehrfach Kanten). $A \subseteq (E^S \cup E^I \cup F \cup C) \times (E^I \cup E^E \cup F \cup C)$
- Die Funktion g gibt für eine Kante $a \in A$ die Kantenbedingung, also den boolschen Ausdruck $expr$, zurück. Gehen die Kanten von einem Konnektor des Typs XOR-Split aus, dann müssen alle Kantenbedingungen so gestaltet sein, dass exakt eine Kantenbedingung erfüllt ist (mutually exclusive). Kantenbedingungen können nur auf Kanten definiert werden, die von einem XOR- oder OR-Split ausgehen $((c,n) \in A | \, l(c) \neq AND \wedge n \in E \cup F \cup C)$. Die Bedingungen aller anderen Kanten, also Kanten von AND-Splits und Sequenzen, sind immer erfüllt.
- Funktion t bildet einen Task auf dessen Typ ab. $t: F \rightarrow \{Subprocess, AdhocSubprocess\}$
- Funktion subref gibt einen BPG zurück, der den Subprozess repräsentiert. $subref: F \rightarrow BPG$
- Funktion h bildet einen Task auf dessen anhaftendes Fehler-Zwischenereignis ab. $h: F \rightarrow E^I_F$

Nachdem die formale Definition für die Beschreibung eines graforientierten Prozesses adaptiert und an die Anforderungen, BPMN darzustellen, angepasst wurde, können nun die Reduktionsregeln spezifiziert werden.

Definition 14 strukturierte Komponente: Wenn BPD ein BPMN Process Graph wie in Definition 13 ist, dann ist $C = \{O_c, F_c, A_c, g_c\}$ eine Komponente von BPD, i_c ist das Eingangsobjekt, o_c das Ausgangsobjekt von C und die folgenden Komponenten sind strukturiert, wenn gilt [4]:

- Sequenz: C ist eine Sequenz-Komponente, wenn eine sequenzielle Verkettung von Tasks und Ereignissen vorliegt, wobei die beteiligten Knotenobjekte genau eine eingehende und eine ausgehend Kante besitzen. Formal: $O_c \subseteq F \cup (E^I \backslash E^I_F)$, $(\forall o \in O_c: |succ(o)| = 1 \wedge$ *predo=1, Oc=o1,...,on, AC={o1,o2,o2,o3,...,(on−1,on)}*
  Hier folgt nun das dazugehörige Asbru-Mapping.

```
<plan name="Sequenz" title=" Sequenz ">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation> <plan-schema name="A" /></plan-activation>
      <plan-activation> <plan-schema name="B" /></plan-activation>
      ...
    </subplans>
  </plan-body>
</plan>
```

Abb. 45: Asbru-Code einer sequentiellen Ausführung von Plan A und B

Abb. 45 zeigt einen Plan (Plan Sequenz) durch dessen *subplans* Element und dessen *type* Attribut, das den Wert „sequentially“ hat, indiziert wird, dass die referenzierten Pläne A und B sequentiell ausgeführt werden. Weiters müssen alle Subpläne (A, B,...) ausgeführt werden, damit Plan „Sequenz“ beendet werden kann.

- Paralleler Block: C ist ein paralleler Block, wenn das Eingangsobjekt ein Konnektor vom Typ AND_{Split} ist, das Ausgangsobjekt ein Konnektor vom Typ AND_{Join} ist und zwischen Ein- und Ausgangsobjekt pro Zweig genau ein Knotenobjekt, aus Tasks oder Zwischenereignissen, liegt. Formal: $i_c, o_c \in C,\ g(i_c) = AND_{Split},\ g(oc) = AND_{Join},\ O_c \subseteq F \cup E^I \cup \{i_c, o_c\}, \forall x \in O_c \backslash \{i_c, o_c\}: (i_c, x) \in A_C \wedge (x, o_c) \in A_C$ Asbru-Mapping siehe Tabelle 9 a.

- Exklusiver Block: C ist ein exklusiver Block, wenn das Eingangsobjekt ein Konnektor vom Typ XOR_{Split} ist, das Ausgangsobjekt ein Konnektor vom Typ XOR_{Join} ist und zwischen Ein- und Ausgangsobjekt pro Zweig genau ein Knotenobjekt, aus Tasks oder Zwischenereignissen, liegt. Formal: $i_c, o_c \in C,\ g(i_c) = XOR_{Split},\ g(oc) = XOR_{Join},\ O_c \subseteq F \cup E^I \cup \{i_c, o_c\},\ \forall x \in O_c \backslash \{i_c, o_c\}: (i_c, x) \in A_C \wedge (x, o_c) \in A_C$ Asbru-Mapping siehe Tabelle 9 b.

- Inklusiver Block: C ist ein inklusiver Block, wenn das Eingangsobjekt ein Konnektor vom Typ OR_{Split} ist, das Ausgangsobjekt ein Konnektor vom Typ OR_{Join} ist und zwischen Ein- und Ausgangsobjekt pro Zweig genau ein Knotenobjekt, aus Tasks oder Zwischenereignissen, liegt. Formal: $i_c, o_c \in C,\ g(i_c) = OR_{Split},\ g(oc) = OR_{Join},\ O_c \subseteq F \cup E^I \cup \{i_c, o_c\},\ \forall x \in O_c \backslash \{i_c, o_c\}: (i_c, x) \in A_C \wedge (x, o_c) \in A_C$ Asbru-Mapping siehe Tabelle 9 c.

- Ereignisbasierter Block: C ist ein ereignisbasierter Block, wenn das Eingangsobjekt ein Konnektor vom Typ $Event_{Split}$ ist, das Ausgangsobjekt ein Konnektor vom Typ XOR_{Join} ist und zwischen Ein- und Ausgangsobjekt pro Zweig ein Zwischenereignis gefolgt von einem optionalen Task liegt. Formal: $i_c, o_c \in C,\ g(i_c) = Event_{Split},\ g(oc) = XOR_{Join}, n = |pred(i_c)|,\ F_C = \{f_1, \dots, f_m\} \subseteq F, m = |F_c| \leq n,\ E_C^I = \{e_1, \dots, e_n\} \subseteq (E^I \setminus E_F^I),\ O_c = E_C^I \cup F_C \cup \{i_c, o_c\},\ A_C = \cup_{i=1}^{n}(\{(i_c, e_i), (f_i, o_c), (e_i, f_i)\} \vee \{(i_c, e_i), (e_i, o_c)\})$ Asbru-Mapping siehe Tabelle 9 e.

- Strukturierte While-Schleife: C ist eine strukturierte While-Schleife, wenn das Eingangsobjekt ein Konnektor vom Typ XOR_{Join} ist, das Ausgangsobjekt ein Konnektor vom Typ XOR_{Split} ist und auf dem rücklaufenden Zweig ein Knotenobjekt, aus Tasks oder Zwischenereignissen, liegt. Formal: $i_c, o_c \in C,\ g(i_c) = XOR_{Join},\ g(oc) = XOR_{Split},\ O_c = \{i_c, o_c, x\},\ x \in F \cup E^I \setminus E_F^I,\ A_C = \{(i_c, o_c), (o_c, x), (x, i_c)\}$ Asbru-Mapping siehe Tabelle 2b bzw. Tabelle 3e.

- Strukturierte Repeat-Schleife: C ist eine strukturierte Repeat-Schleife, wenn das Eingangsobjekt ein Konnektor vom Typ XOR_{Join} ist, das Ausgangsobjekt ein Konnektor vom

Typ XOR_{Split} ist und auf dem vorlaufenden Zweig ein Knotenobjekt, aus Tasks oder Zwischenereignissen, liegt. Formal: $i_c, o_c \in C,\ g(i_c) = XOR_{Join},\ g(oc) = XOR_{Split},\ O_c = \{i_c, o_c, x\},\ x \in F \cup E^I \setminus E_F^I,\ A_C = \{(i_c, x), (x, o_c), (o_c, i_c)\}$ Asbru-Mapping siehe Tabelle 2b bzw. Tabelle 3e.

- Ad-Hoc Subprozess: C ist ein Ad-Hoc Subprozess, wenn ein Subprozesselement mit einer Ad-Hoc Markierung (Tilde Symbol) gekennzeichnet wurde und der Subprozess nur Tasks bzw. Subprozesse enthält, die mit keinen Kanten verbunden sind. Formal: $i_C = o_C \in F,\ t(i_C) = AdhocSubprocess,\ subref(i_C) = \{\emptyset, \emptyset, \emptyset, \emptyset, F_{sub}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\},\ F_{sub} \subseteq F$ Asbru-Mapping siehe Tabelle 3h.

- Cancel Block: C ist ein Cancel Block, wenn einem Task oder Subprozesselement ein Fehler Zwischenereignis anhaftet und davon ausgehend eine Kante einen alternativen Kontrollfluss anzeigt. Dieser Teilprozess, der den alternativen Kontrollfluss realisiert, darf nicht mit dem normalen Ablauf überlappen. Dies wird als vereinfachende Einschränkung angenommen. Formal: $i_C = o_C \in F, e_F = h(i_C) \in E_F^I,\ f_{alternative} \in F_C,\ (e_F, f_{alternaitve}) \in A_C$ Asbru-Mapping siehe Abb. 41.

Die Reduktion erfolgt, indem die Komponente C, die ein Teilgraf des BPMN Process Graph BPG ist, durch einen neuen Task t_C in BPG ersetzt wird. Dabei werden alle Knotenobjekte und Kanten der Komponente C aus BPG entfernt und durch t_C und entsprechende Kanten ergänzt, die in t_C hinein und aus t_C herausführen. Formal ist dies wie folgt definiert (ähnlich wie in [4])

Definition 15 Reduktionsschritt: Wenn $BPG = \{O, E^S, E^I, E^E, F, C, l, A, g, t, subref, h\}$ ein BPG ist und $C = \{O_c, F_c, A_c, g_c\}$ ist ein Komponente von BPG, dann wird C durch einen Task t_C in BPG ersetzt und dadurch entsteht $B\acute{P}G = \{\acute{O}, \acute{E}^S, \acute{E}^I, \acute{E}^E, \acute{F}, \acute{C}, \acute{l}, \acute{A}, \acute{g}, \acute{t}, \acute{subref}, \acute{h}\}$. Weiters muss gelten:

- $\acute{O} = O \setminus O_C \cup t_C$ Alle Knotenobjekte von C werden durch t_C ersetzt.
- $\acute{F} = F \setminus F_C \cup t_C$ Alle Tasks von C werden durch t_C ersetzt.
- $\acute{C} = C \setminus C_C$ Alle Konnektoren von C werden entfernt.
- $\acute{A} = \left(A \cap (F \setminus F_C \times F \setminus F_C)\right) \cup \{(entry(C), t_C), (t_C, exit(C))\}$ Alle Kanten von C werden entfernt und die Kanten in t_C und aus t_C werden hinzugefügt.
- $\acute{g} = g[\acute{A}]$ Dies bedeutet, dass die Definitionsmenge der Funktion g auf die Kantenmenge A‘ eingeschränkt wird und so g‘ bildet.
- $\acute{t} = t[\acute{F}]$
- $\acute{subref} = subref[\acute{F}]$
- $\acute{h} = h[\acute{F}]$
- $\acute{E}^S = E^S \setminus E_C^S$
- $\acute{E}^I = E^I \setminus E_C^I$
- $\acute{E}^E = E^E \setminus E_C^E$

Nachdem strukturierte Aktivitäten und deren Asbru-Mapping definiert wurden, und auch die Reduktion für einen Schritt beschrieben wurde, kann somit die Structure-Identification Strategie für die BPMN-zu-Asbru Transformation wie in Abschnitt 3.2.5.3 angewandt werden.

Tabelle 10 fasst die Ergebnisse der Evaluierung zusammen und zeigt die Umsetzbarkeit der Beispiele unstrukturierter Blöcke. Beispiel 2 a bzw. b lässt sich realisieren, da es sich wie der o.a. strukturierte parallele Block bzw. exklusive Block verhält. Die restlichen Beispiele lassen sich mit Asbru-Konstrukten nicht realisieren.

Tabelle 10: Resultate der mit dieser Strategie realisierbaren Beispiele unstrukturierter Blöcke. + vollständig, +/- teilweise, - nicht realisierbar

	Realisierbarkeit	Bemerkung
Beispiel 1	-	
Beispiel 2a	+	Verhält sich wie ein paralleler Block
Beispiel 2b	-	
Beispiel 2c	-	
Beispiel 2d	+	Verhält sich wie ein exklusiver Block

4.3.3 Element-Preservation Strategie

Bei der Element-Preservation Strategie wird jede Aktivität des Eingangsmodells in eine BPEL-Basisaktivität umgewandelt. Der Kontrollfluss wird mit Hilfe von BPEL-Links dargestellt, mit denen die Abfolge und Abhängigkeiten der Tasks abgebildet werden (siehe Abschnitt 3.2.5.1 bzw. [29]).

Pläne (user-performed) können in Asbru als Gegenstück zu BPMN Aktivitäten gesehen werden. Wird eine Asbru Leitlinie ausgeführt, können Pläne mehrere Zustände annehmen, bis sie aktiviert und erfolgreich ausgeführt wurden. Asbru kann die Aktivierung eines Plans davon abhängig machen, dass ein anderer Plan einen bestimmten Zustand erreicht hat. Also kann definiert werden, dass Plan B aktiviert wird, wenn Plan A erfolgreich beendet wurde. Dieses Verhalten entspricht dem Verhalten von BPEL-Links, um den Kontrollfluss eines BPEL-Prozesses darzustellen.

Asbru bietet dafür das *plan-state-constraint* Element an, das als eine Art logischer Operator, z.B. in der Filterbedingung eines Plans, verwendet werden kann.

```
<plan name="Plan_Block" title="Plan_Block">
  <plan-body>
    <subplans type="any-order" wait-for-optional-subplans="yes">
      <wait-for><one/></wait-for>
      ...
      <plan-activation>
        <plan-schema name="Plan_B"/>
      </plan-activation>
      ...
    </subplans>
  </plan-body>
</plan>

Referenzierter Plan B:
<plan name="Plan_B" title="Plan B">
  <conditions>
    <filter-precondition>
      <plan-state-constraint state="completed">
        <plan-pointer >
          <static-plan-pointer plan-name="Plan_A" />
        </plan-pointer>
        ...
      </plan-state-constraint>
    </filter-precondition>
  </conditions>
  <plan-body><user-performed /></plan-body>
</plan>
```

Abb. 46: Asbru-Code, mit dem durch plain-state-constraint die Planaktivierung von Plan-B abhängig von der Beendigung von Plan-A gemacht wird.

Abb. 46 zeigt einen Plan (Plan_Block), dessen *subplans* Element es erlaubt, die Subpläne in beliebiger Reihenfolge (type="any-order"; auch „unordered“ möglich, falls parallele Ausführungen notwendig sind) auszuführen und zur erfolgreichen Beendigung einen einzelnen erfolgreich beendeten Plan benötig (wait-for=one), jedoch das Beenden optionaler Pläne abwartet (wait-for-optional-subplans="yes").

In den referenzierten Plänen (z.B. Plan_B) wird in der Filterbedingung das *plan-state-constraint* Element verwendet, um die Abhängigkeit der Aktivierung von anderen Plänen zu realisieren. So wird Plan_B erst aktiviert, wenn Plan_A (plan-name="Plan_A") den Zustand „completed“ (state="completed") erreicht hat.

Somit lassen sich Kontrollflüsse darstellen, die mit strukturierten Asbru Plänen nicht dargestellt werden können. Um die Tauglichkeit zu zeigen und um die Strategie zu veranschaulichen, wird am Beispiel 2a die Vorgehensweise veranschaulicht und an folgenden Asbru Code-Ausschnitten die Realisierung gezeigt.

```
<plan name="Plan_A" title="Plan A">
  <plan-body>
    <user-performed />
  </plan-body>
</plan>

<plan name="Plan_B" title="Plan B">
  <plan-body>
    <user-performed />
  </plan-body>
</plan>

<plan name="Plan_C" title="Plan C">
  <conditions>
    <filter-precondition>
      <constraint-combination type="or">
        <plan-state-constraint state="completed">
          <plan-pointer >
            <static-plan-pointer plan-name="Plan_A" />
          </plan-pointer>  ...
        </plan-state-constraint>
        <plan-state-constraint state="completed">
          <plan-pointer >
            <static-plan-pointer plan-name="Plan_B" />
          </plan-pointer>  ...
        </plan-state-constraint>
      </constraint-combination>
    </filter-precondition>
  </conditions>
  <plan-body>
    <user-performed />
  </plan-body>
</plan>
```

Abb. 47: Referenzierte Pläne aus Abb. 48. Aktivierung von Plan C ist abhängig von Plan A und B .

Abb. 47 zeigt die Pläne A bis C, die von dem Asbru-Codeteil in Abb. 48 referenziert und aktiviert werden. Die Pläne A und B haben keine Filterbedingung und können somit ohne weitere Vorbedingung ausgeführt werden. Die Filterbedingung von Plan C ist jedoch erst dann erfüllt, wenn Plan A oder Plan B erfolgreich beendet wurde. In Kombination mit dem Asbru Plan in Abb. 48 wird der Kontrollfluss aus Beispiel 2a simuliert. Plan_Block überprüft, ob die Pläne A bis C ausgeführt werden können, was auf Grund der Filterbedingung anfangs jedoch nur auf Plan A und B zutrifft. Nachdem einer der beiden Pläne oder beide gleichzeitig den Zustand „completed“ erreicht bzw.

erreichen, kann auch Plan C ausgeführt werden, da nun dessen Filterbedingung auch erfüllt ist. Nun kann auch Plan_Block beendet werden, da der eine erforderliche Subplan und die optionalen Subpläne beendet wurden.

```
<plan name="Plan_Block" title="Plan_Block">
  <plan-body>
    <subplans type="unordered" wait-for-optional-subplans="yes">
      <wait-for><one/></wait-for>
      <plan-activation>
        <plan-schema name="Plan_A"/>
      </plan-activation>
      <plan-activation>
        <plan-schema name="Plan_B"/>
      </plan-activation>
      <plan-activation>
        <plan-schema name="Plan_C"/>
      </plan-activation>
    </subplans>
  </plan-body>
</plan>
```

Abb. 48: Asbru-Code um Pläne A bis C in beliebiger auch überlappender Anordnung auszuführen, wobei auch optionale Pläne beendet werden. Teil der Realisierung von Beispiel 2a

Zu beachten ist jedoch, dass die Semantik, die einem BPMN (X)OR-Join Gateway zugeordnet ist, mit der OR bzw. XOR Verknüpfung einer Asbru Filterbedingung (siehe Abb. 47 *constraint-combination* Element) nicht exakt nachgebildet werden kann.
Ein BPMN OR-Join Gateway sagt aus, dass ein Token erst an die ausgehende Kante weitergegeben wird, wenn alle Token der aktivierten, eingehenden Zweige am Gateway eingetroffen sind.
Im Gegensatz dazu wird bei einem Asbru *constraint-combination* Element vom Typ „or“ die Filterteilbedingung erfüllt, wenn eine oder mehrere Teilbedingungen erfüllt sind. D.h., dass hier keine Synchronisation wie bei einem BPMN Gateway stattfindet, und hier das Discriminator Pattern realisiert wird.
Auch die Semantik des BPMN XOR-Join Gateways deckt sich nicht mit Asbru *constraint-combination* Element vom Typ „xor“. Das BPMN XOR-Join Gateway würde Token aus verschiedenen eingehenden Zweigen mehrfach an die ausgehende Kante weitergeben. Während ein *constraint-combination* Element vom Typ „xor“ die Filterbedingung erfüllt, wenn genau eine Teilbedingung erfüllt wird. D.h. sind mehrere Teilbedingungen erfüllt, wird der Nachfolgeplan (in unserem Beispiel Plan C) gar nicht aktiviert und nicht nur nicht mehrfach, wie in der BPMN Semantik.

Der vollständige Asbru-Code zu Beispiel 2a ist in Anhang B, Abschnitt 1 zu finden. Die Ergebnisse aller Beispiele sind in Tabelle 11 zusammengefasst.

Tabelle 11: Resultate der mit dieser Strategie realisierbaren Beispiele unstrukturierter Blöcke. + vollständig, +/- teilweise, - nicht realisierbar.

	Realisierbarkeit	Bemerkung
Beispiel 1	-	
Beispiel 2a	+/-	Synchronisationspunkt m. abweichender Semantik
Beispiel 2b	+/-	Synchronisationspunkt m. abweichender Semantik
Beispiel 2c	+/-	Synchronisationspunkt m. abweichender Semantik
Beispiel 2d	+/-	Synchronisationspunkt m. abweichender Semantik

Diese Strategie eignet sich nicht zur Simulation von Schleifen, da jeder in einem Subplan-Element referenzierter Plan nur max. einmal aufgerufen werden kann. Deshalb lassen sich Schleifen mit dieser Strategie nicht simulieren.

Somit eignet sich diese Strategie nur für azyklische unstrukturierte BPMN Eingangsmodelle(-grafen), wobei bei unstrukturierten Blöcken die u.a. abweichende Semantik bei Join-Gateways berücksichtigt werden muss.

4.3.4 Event-Condition-Action-Rules Strategie

Bei der Event-Condition-Action-Rules Strategie wird, je nach Variante, jede Aktivität in einem Prozess bzw. jede Aktivität in einer unstrukturierten Komponente mit einem Event-Handler assoziiert. Dadurch kann jede Aktivität bzw. der entsprechende Event-Handler durch Aktivieren des entsprechenden Events ausgeführt werden. Dadurch, dass jeder Event-Handler beim Ausführen den Event des bzw. der nachkommenden Event-Handler aktiviert, kann der entsprechende Ablauf definiert werden (siehe Abschnitt 3.2.5.5 bzw. [4], [29]).

Die Asbru Spezifikation weist jedoch keine vergleichbare Konstrukte auf, um Events zu werfen bzw. Event-Handler zu definieren [12].
Simuliert kann dieses Verhalten jedoch mit globalen Variablen werden. Die Idee ist, für jede Aktivität eine globale boolsche Variable zur Verfügung zu stellen, mit der die Beendigung der korrespondierenden Aktivität ausgedrückt werden kann, indem die Variable auf „true“ gesetzt wird. Diese Variablen können nun eingesetzt werden, um Filterbedingungen der Pläne so zu definieren und so die entsprechende Abfolge zu bestimmen. Dabei ist zu beachten, dass jeder Plan die Variablen, die in seiner Filterbedingung verwendet werden, wieder auf „false“ zurück setzt, um unbeabsichtigtes, mehrfaches Aufrufen zu vermeiden.

Im Detail wird dies an folgenden allgemeinen Asbru Code-Abschnitten demonstriert:

```
<plan-library>
  <library-defs>
    <qualitative-scale-def name="Boolean">
      <qualitative-entry entry="false" />
      <qualitative-entry entry="true" />
    </qualitative-scale-def>

    <variable-def  name="a_comp"  >
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="false"/>
        </initial-value>
      </scalar-def>
    </variable-def>
    ...
  </library-defs>
  ...
```

Abb. 49: Asbru-Code zur Definition des Aufzählungstyp Boolean und einer globalen Variable von diesem Typ

Zuerst muss für jede Aktivität eine boolsche Steuervariable definiert werden, mit der das Beenden der korrespondierenden Aktivität dargestellt werden kann. Abb. 49 zeigt die Definition eines Asbru

Aufzählungstyps mit dem Namen „Boolean“ (*qualitative-scale-def* Element) mit den möglichen Ausprägungen „false“ und „true“ (qualitative-entry-Element).
Weiters wird in Abb. 49 die globale Variable „a_comp“ (*variable-def* Element), vom Aufzählungstyp „Boolean“ (*scalar-def* Element *type*=“Boolean“), definiert und mit dem initialen Wert „false“ belegt (*initial-value* Element). Die Definition des Typs und der Variablen sind global, da sie im *library-defs* Element deklariert wurden. Somit ist die Variable in jedem Plan der Plan Library sichtbar, und der Typ kann in jedem Plan zum Deklarieren lokaler Variablen verwendet werden.

Nach der Variablendefinition können die Steuervariablen in den entsprechenden Filterbedingungen verwendet werden und so entsprechende Abläufe definiert werden.

```
<plan name="Plan_B" title="Plan B">
  <conditions>
    <filter-precondition>
      <simple-condition>
        <comparison type="equal">
          <left-hand-side>
            <variable-ref name="a_comp"/>
          </left-hand-side>
          <right-hand-side>
            <qualitative-constant value="true"/>
          </right-hand-side>
        </comparison>
      </simple-condition>
    </filter-precondition>
  </conditions>

  <plan-body>
    <subplans type="sequentially">
      ...
      <user-performed />
      <variable-assignment variable="a_comp"> <! --Steuervariable rücksetzen -->
        <qualitative-constant value="false"/>
      </variable-assignment>
      <variable-assignment variable="b_comp"> <! --Steuervariable für Plan B setzen -->
        <qualitative-constant value="true"/>
      </variable-assignment>
    </subplans>
  </plan-body>
</plan>
```

Abb. 50: Asbru-Code für Plan B mit Filterbedingung mit globaler Steuervariable (a_comp) und Plan-Body mit Wertezuweisungen für Variable a_comb und b_comp.

Abb. 50 zeigt die Asbru-Definition von Plan B, die einer BPMN-Aktivität entspricht. Die Filterbedingung besagt, dass Plan B erst aktiviert werden kann, wenn die Steuervariable a_comp (left-hand-side Element) den Wert „true“ (*right-hand-side* Element) annimmt (comparison type="equal"). Das bedeutet im Kontext der Strategie, dass Plan B erst aktiviert wird, wenn Plan A beendet wurde.
Im unteren Teil von Abb. 50, dem Plan Body von Plan B, wird eine Sequenz von Anweisungen ausgeführt. Zuerst wird eine vom Benutzer ausgeführte Aktivität (*user-performed* Element) durchgeführt, die in Asbru nicht weiter modelliert wird. Anschließend wird die Steuervariable a_comp zurückgesetzt (*variable-assignment* Element), indem ihr der Wert „false“ zugeordnet wird

(*qualitative-constant* Element). Danach, als letzter Schritt, wird die Steuervariable b_comp auf den Wert „true“ gesetzt, um die Beendigung von Plan (Aktivität) B darzustellen.

Plan A (der hier nicht angeführt ist), Plan B und diverse andere Pläne, die eine BPMN-Aktivität darstellen, müssen nun wiederholt aufgerufen werden, um die gewünschte Abfolge von Aktivitäten zu simulieren.

```
<plan name="Root" title="Root">
  <plan-body>
    <iterative-plan>
        <do-repeatedly>
          <plan-activation>
            <plan-schema name=" Block"/>
          </plan-activation>
        </do-repeatedly>
        <termination-condition>... </termination-condition>
    </iterative-plan>
  </plan-body>
</plan>

<plan name="Block" title="Block">
  <plan-body>
    <subplans type="unordered" wait-for-optional-subplans="yes">
      <wait-for><one/></wait-for>
      <plan-activation>  <plan-schema name="Plan_A"/>
      </plan-activation>
      <plan-activation>  <plan-schema name="Plan_B"/>
      </plan-activation>
      <plan-activation>  <plan-schema name="Plan_C"/>
      </plan-activation>
      ...
    </subplans>
  </plan-body>
</plan>
```

Abb. 51: Asbru-Code für die iterativen Aktivierung von Plan Block, der je nach Steuervariablenbelegung die jeweiligen Subpläne aktiviert.

In Abb. 51 aktiviert Plan „Root“ iterativ Plan „Block“ bis eine hier nicht näher bestimmte Abbruchbedingung erfüllt ist. Die Abbruchbedingung könnte z.B. sein, dass ein Endzustand erreicht wurde, also die Aktivität vor dem Endzustand beendet wurde bzw. die korrespondierende Steuervariable auf „true“ gesetzt wurde.
Plan „Block“ wird wiederholt von Plan „Root“ aktiviert, und der aktiviert alle sein Subpläne (A, B, ...). Je nach Belegung der Steuervariablen, wird der Subplan (bei paralleler Ausführung auch mehrere gleichzeitig) aktiviert, dessen Filterbedingung die Steuervariable enthält, die gerade mit „true“ belegt ist. Vorausgesetzt alle notwendigen Nebenbedingungen sind ebenfalls erfüllt.
In Verbindung mit Subplänen, die eine Verkettung der Ablaufreihenfolge mit deren Filterbedingungen (wie in Abb. 50) realisieren, kann so eine Abfolge von Asbru Plänen realisiert werden, die der Event-Condition-Action-Rule-Strategie entspricht.

Diese Strategie wird nun wieder an Beispiel 2a veranschaulicht. Das Beispiel wird Top-Down anhand von Code-Ausschnitten erläutert, wobei der vollständigen Asbru-Code in Anhang B, Abschnitt 1.1 zu finden ist.

Zuerst werden die globalen boolschen Steuervariablen start_comp, a_comp, b_comp, c_comp wie in Abb. 49 definiert, wobei start_comp den Initialwert „true“ und die restlichen Variablen den Initialwert „false“ haben. Variable start_comp symbolisiert den Startzustand, der die Nachfolgezustände, in unserm Beispiel A und B, aktiviert und ist daher zu Beginn „true“.
Danach muss die entsprechende Abbruchbedingung im *iterative-plan* Element des Plans „root“ gesetzt werden (siehe Abb. 52). Die iterative Ausführung von Plan „Block“ wird abgebrochen, wenn Aktivität C beendet wurde, also Variable c_comp den Wert „true“ hat.
Plan „Block“ ist wie in Abb. 51 definiert und aktiviert Plan A, B, oder C, abhängig von der Filterbedingung.

```
<plan name="root" title="root">
  <plan-body>
    <iterative-plan>
      <do-repeatedly>
        <plan-activation>
          <plan-schema name="Block"/>
        </plan-activation>
      </do-repeatedly>
      <termination-condition>
        <comparison type="equal">
          <left-hand-side>
            <variable-ref name="c_comp"/>
          </left-hand-side>
          <right-hand-side>
            <qualitative-constant value="true"/>
          </right-hand-side>
        </comparison>
      </termination-condition>
    </iterative-plan>
  </plan-body>
</plan>
```

Abb. 52: Iterativer Plan, der Plan „block“ wiederholt aktiviert und abbricht, wenn c_comp true ist

Plan A und B werden jeweils aktiviert, wenn Variable start_comp „true“ ist, was dem Intialwert der Variable entspricht. Im Plan Body der Pläne A und B werden als letzter Schritt die Variablen a_comp bzw. b_comp auf den Wert „true“ gesetzt. Dies wiederum aktiviert Plan C (siehe Abb. 53). Die Filterbedingung von Plan C ist erfüllt, wenn die Variable a_comp oder b_comp den Wert „true“ hat und stellt den Synchronisationspunkt bzw. das OR-Join Gateway aus Beispiel 2a dar.
Im Plan Body von Plan C werden nach Beenden des *user-performed* Elements die Variablen a_comp und b_comp zurück auf „false“ gesetzt. Durch das Setzen der Variable c_comp auf „true“ wird das Ende von Plan (Aktivität) C repräsentiert und so die Abbruchbedingung des iterativen Plans in Plan „Root“ (Abb. 52) erfüllt. Somit wurde der Kontrollfluss aus Beispiel 2a mit geänderter Semantik im Synchronisationspunkt erreicht.

```
<plan name="Plan_C" title="Plan C">
  <conditions>
    <filter-precondition>
      <constraint-combination type="or">
        <simple-condition>
          <comparison type="equal">
            <left-hand-side><variable-ref name="a_comp"/></left-hand-side>
            <right-hand-side><qualitative-constant value="true"/></right-hand-side>
          </comparison>
        </simple-condition>
        <simple-condition>
          <comparison type="equal">
            <left-hand-side><variable-ref name="b_comp"/> </left-hand-side>
            <right-hand-side><qualitative-constant value="true"/></right-hand-side>
          </comparison>
        </simple-condition>
      </constraint-combination>
    </filter-precondition>
  </conditions>
  <plan-body>
    <subplans type="sequentially">
      <wait-for>
        <all/>
      </wait-for>
      <user-performed />
      <variable-assignment variable="a_comp"><qualitative-constant value="false"/>
      </variable-assignment>
      <variable-assignment variable="b_comp"><qualitative-constant value="false"/>
      </variable-assignment>
     <variable-assignment variable="c_comp"><qualitative-constant value="true"/>
      </variable-assignment>
    </subplans>
  </plan-body>
</plan>
```

Abb. 53: Asbru-Code: Plan C wird aktiviert, wenn a_comp oder b_comp „true“ sind, und setzt beide Variablen im Plan Body zurück, sowie c_comp auf true.

Die oben beschriebene Strategie beruht auf der Asbru-Spezifikation. Bei der praktischen Evaluierung mit dem Asbru-Interpreter wurde jedoch festgestellt, dass der Interpreter in der aktuellen Version zwei für die Strategie essentielle Konstrukte nicht unterstützt.
Zum Einen ist das *iterative-plan* Element nicht implementiert, und somit lassen sich Asbru Plan Libraries mit diesem Element nicht ausführen. Alternativ steht das cyclical-plan Element zur Verfügung, um wiederholende Prozessteile auszuführen. Das *cyclical-plan* Element wurde entwickelt, um zeitlich bedingte Wiederholungen zu modellieren, z.B. um einen Teilprozess alle 30 Minuten zu starten. Es kann jedoch auch so konfiguriert werden, dass eine iterative Wiederholung erfolgt, die unabhängig von zeitlichen Bedingungen ist. Da das *cyclical-plan* Element im Asbru-Interpreter implementiert ist, stellt es die einzige funktionierende Alternativ zum *iterative-plan* Element dar. Um im Sinn der Spezifikation zu handeln, sollte jedoch grundsätzlich das *iterative-plan* Element verwendet werden.
Zum Anderen sind Wertezuweisungen (*variable-assignment* Element), die den initial festgelegten Variablenwert ändern, nicht möglich und werden vom Interpreter ignoriert, können jedoch

ausgeführt werden. Dadurch lässt sich keines der Beispiele unstrukturierter Blöcke im Moment mit dieser Strategie praktisch realisieren.
Mit dem momentanen Entwicklungsstand des Interpreters ist diese Strategie nicht anwendbar. Sie birgt jedoch das Potential, eine universelle Lösungsstrategie zu einem späteren Zeitpunkt zu sein, wenn der Interpreter das *variable-assignment* bzw. *iterativ-plan* Element (im Sinn eines Prototypen, der konform zur Spezifikation implementiert ist) unterstützt.

Tabelle 12: Resultate der mit dieser Strategie realisierbaren Beispiele unstrukturierter Blöcke. + vollständig, +/- teilweise, - nicht realisierbar.

	Realisierbarkeit	Bemerkung
Beispiel 1	-	
Beispiel 2a	-	
Beispiel 2b	-	
Beispiel 2c	-	
Beispiel 2d	-	

4.3.5 Umzusetzende Strategie

Die zu evaluierenden Strategien wurden nun im Detail bewertet, und es muss eine Strategie gewählt werden, die für die Implementierung im Prototyp geeignet erscheint.

Es wurde die Structure-Identification Strategie für die Realisierung des Prototypen gewählt, da damit jene strukturierten Asbru-Elemente verwendet werden, die zum Ausdrücken des Kontrollflusses vorgesehen sind.
Dies erhöht die Lesbarkeit der generierten Lösung und verringert die Fehlerhäufigkeit bei einer etwaigen manuellen Adaption der generierten Asbru Plan Library.
Diese Strategie kann zwar nur strukturierte Eingangsgrafen verarbeiten, der Prototyp kann jedoch zu einem späteren Zeitpunkt für die Transformation unstrukturierter Komponenten erweitert werden. Dies ermöglicht es, eine der beiden anderen Strategien ergänzend umzusetzen, wenn der Entwicklungsstand des Asbru-Interpreters dies zulässt.
Gegen die anderen Strategien spricht, dass mit der Element-Preservation Strategie keine Schleifen umgesetzt werden können. Die Event-Condition-Action-Rules Strategie kann, aufgrund des Entwicklungsstands des Interpreters, zurzeit nicht praktisch umgesetzt werden.

Tabelle 13: Alle evaluierten Strategien im Vergleich mit den damit realisierbaren Beispielen unstrukturierter Blöcke. + vollständig, +/- teilweise, - nicht realisierbar.

Strategie	Umsetzbarkeit				
	Beispiel 1	Beispiel 2a	Beispiel 2b	Beispiel 2c	Beispiel 2d
Structure-Identification Strategie	-	-	-	-	-
Element-Preservation Strategie	-	+/-	+/-	+/-	+/-
Event-Condition-Action-Rules Strategie	-	-	-	-	-

4.4 Implementierung des Transformationssystem

Dieser Abschnitt beschreibt die Implementierung eines Transformationssystems, das die o.a. Transformationsstrategie und das o.a. Mapping in eine reale Anwendung umsetzt. Damit soll gezeigt werden, dass die gewählte Transformationsstrategie in einem Software-Prototyp umsetzbar ist, und das Mapping zu einer korrekten Übersetzung führt. Daher beantwortet dieser Abschnitt die 4. Forschungsfrage dieser Arbeit.

Das umgesetzte Transformationssystem kann eine Teilmenge möglicher BPMN-Diagramme übersetzen, wobei die Teilmenge übersetzbarer BPMN-Prozessen nur strukturierte Komponenten, wie sie in Definition 14 (Seite 75, Abschnitt 4.3.2) erläutert wurden, und zusätzlich strukturierte Schleifen enthalten darf. Damit sind jene 20 ursprünglichen Kontrollflussmuster, wie in [31] und [14] definiert, abgedeckt, die von Asbru unterstützt werden. Eine genauere Auflistung der von Asbru unterstützten Kontrollflussmuster finden sie in
.

Die generelle Idee des Transformationssystems ist, das zu übersetzende BPMN-Diagramm, welches im serialisierten, standardisierten BPMN XML-Format vorliegt, durch Anwenden von XSLT-Stylesheets in das Asbru XML-Format überzuführen. Dies erfolgt in zwei Stufen, indem das BPMN-Diagramm in eine Zwischendarstellung übergeführt wird (Stufe 1), und aus dieser Zwischendarstellung abschließend das Asbru XML-Format gewonnen wird (Stufe 2). Die Zwischendarstellung bildet schon die Struktur der Asbru-Elemente und Konstrukte ab und folgt daher auch schon dem blockorientierten Kontrollflussparadigma.

Der gesamte Übersetzungsprozess kann in zwei Phasen eingeteilt werden. Wobei die Transformationsphase die Elemente der Zwischendarstellung erzeugt und optimiert, und die Asbru-Erzeugungsphase die letztendliche Übersetzung in Asbru-Code durchführt. Somit realisiert die Transformationsphase Stufe 1, und die Asbru-Erzeugungsphase Stufe 2. Jeder Phase werden ein oder mehrere XSLT-Stylesheets zugeordnet, welche die entsprechenden Teilschritte realisieren. Die einzelnen Phasen setzen unterschiedliche Aufgaben um, die wie folgt gegliedert sind:

- In der Transformationsphase erfolgt zuerst eine Initalisierung/Präparierung des Models. Dabei wird (1) nicht benötigte Information entfernt, (2) bereits einfache Übersetzungen durchgeführt und (3) ein XML-Format etabliert, das während des gesamten Transformationsprozesses Verwendung findet. Anschließend werden Muster transformiert. Im Detail bedeutet das:

  (1) BPMN-Diagramme enthalten Informationen zur grafischen Repräsentation der im Prozess verwendeten Elemente. Dies ist z.B. die Anordnung der Elemente in einem Koordinatensystem oder der grafische Verlauf der Kontrollflusskanten im Diagramm. Diese Informationen sind für den Transformationsprozess unwesentlich und werden in der Initialisierungsphase entfernt.

  (2) Ein Zwischenformat wird erstellt, d.h. es werden hier einzelne BPMN-Elemente in Elemente der Zwischendarstellung übersetzt (z.B. abstrakter Task, Benutzer Task, Skript Task, etc.).

  (3) Die Anpassung des BPMN-Eingangsformats an ein XML-Format erfolgt, welches während des gesamten Transformationsprozesses verwendet werden kann und das die Aufnahme der Elemente des Zwischenformats ermöglicht. So gibt es drei wesentliche Informationsstränge, die im XML-Format enthalten sind und hier nachfolgend aufgelistet werden:
     - Eine Komponente stellt den um Informationen zur grafischen Darstellung bereinigten ursprünglichen BPMN-Prozess dar, der während der Übersetzungsschritte unverändert

bleibt, um ständig auf Informationen des Originalprozesses zugreifen zu können. Der in Abb. 54 unter C gezeigte Bereich enthält die ursprüngliche Definition eines BPMN-Prozesses.

- Weiters ist eine Arbeitskopie des informationsbereinigten BPMN-Prozesses enthalten, die zu Beginn mit dem Originalprozess übereinstimmt. Da bei der Mustertransformation jedoch die Reduktionsschritte auf die Arbeitskopie angewendet werden, werden hier sukzessiv Elemente daraus entfernt, bis keine reduzierbare Muster mehr vorhanden sind. Der in Abb. 54 gezeigte Bereich B enthält die bereinigte Kopie des ursprünglichen BPMN-Prozesses aus Bereich C.

- Die bei der Anwendung der Reduktionschritte in der Mustertransformationsphase entstehenden Elemente der Zwischendarstellung bilden die dritte Komponente (siehe Abb. 54 Bereich A).

```
<?xml version="1.0" encoding="UTF-8"?>
<bat:transformation xmlns:model="http://www.omg.org/spec/BPMN/20100524/MODEL"
          xmlns:bat="http://www.masterarbeit/thomastschach/transform"... >
      <bat:plans>
A       <bat:userPerformedPlan id="A" name="A"/>
      </bat:plans>

      <bat:workingcopy>...
        <model:process id="P1_1" name="P1_1">
          <model:startEvent id="Start1" name="Start1"/>
B         <model:task id="A" name="A"/>...
          <model:sequenceFlow id="e1" sourceRef="Start1" targetRef="A"/>...
        </model:process>
      </bat:workingcopy>

      <model:definitions>...
        <model:process id="P1_1" name="P1_1">
          <model:documentation/>
          <model:startEvent id="Start1" name="Start1">
C           <model:documentation/>
          </model:startEvent>
          <model:task id="A" name="A">
            <model:documentation/></model:task>
          <model:sequenceFlow id="e1" sourceRef="Start1" targetRef="A"/> ...
        </model:process>
      </model:definitions>
</bat:transformation>
```

Abb. 54: Darstellung des durch die Initialisierungsphase geschaffenen XML-Formats

(4) Danach werden in der Transformationsphase jene Übersetzungen durchgeführt, die komplexer sind, da sie von der Konfiguration mehrerer BPMN-Elemente zueinander abhängig sind. D.h. hier werden die verschiedenen Muster identifiziert, die übersetzt und reduziert werden können, und entsprechende Elemente der Zwischendarstellung erzeugt. Das zu übersetzende BPMN-Diagramm besteht meist aus mehreren verschachtelten Mustern. Da pro Transformationsschritt nur ein Kontrollflussmuster reduziert wird, ist eine mehrmalige Anwendung des Stylesheets notwendig, das die Mustertransformationsphase umsetzt, um den BPMN-Prozess vollständig zu reduzieren.

Im Zuge der Transformation der Sequenz, können Optimierungsschritte durchgeführt werden.

- Die Asbru-Erzeugungsphase führt die Übersetzung der Elemente der Zwischendarstellung in Asbru XML-Elemente durch. Da die Zwischendarstellung bereits Struktur der Asbru-Elemente aufweist, ist hierfür nur ein einzelner Transformationsschritt nötig.

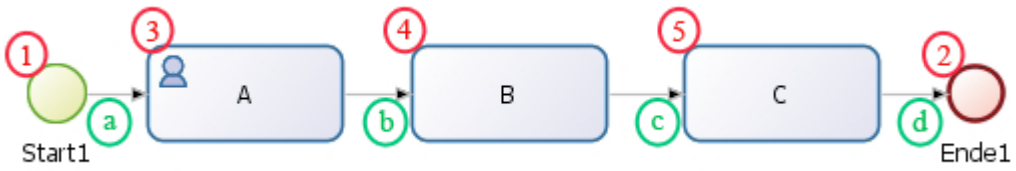
In den weiteren Unterabschnitten wird für jedes von Asbru unterstützte Kontrollflussmuster bzw. jede Musterkombination (Verzweigungs- und Synchronisationspunkt bei Blöcken) das konzeptuelle Vorgehen der Transformationssystems erläutert. Für jedes Kontrollflussmuster bzw. jede Musterkombination werden daher die einzelnen Schritte für jede Phase beschrieben, um so einen Einblick in die Arbeitsweise des Transformationssystems zu bekommen. Dies erfolgt meist anhand eines veranschaulichenden Beispiels.

4.4.1 Standard Kontrollflussmuster

Die Standard Kontrollflussmuster bestehen aus dem Sequence, Parallel Split, Synchronization, Exclusive Choice und dem Simple Merge Pattern.

4.4.1.1 Sequence (WCP1)

Das Kontrollflussmuster der Sequenz beschreibt zwei oder mehrere Aktivitäten, die in sequenzieller Abfolge nacheinander ausgeführt werden (siehe auch Abschnitt 4.1.1.1). In BPMN sind dafür zwei oder mehrere Tasks mit jeweils einer Kontrollflusskante (*bpmn:sequenceFlow*[8] Element) sequentiell miteinander verbunden. Jeder involvierte Task hat genau eine eingehende und eine ausgehende Kante, wobei die ausgehende Kante des Vorgängers gleichzeitig die eingehende Kante des nachfolgenden Tasks ist (siehe auch Definition 14 S.75).




```
1   <bpmn:startEvent id="Start" name="Start"/>
2   <bpmn:endEvent id="Ende" name="Ende"/>
3   <bpmn:userTask id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
5   <bpmn:task id="C" name="C"/>

a   <bpmn:sequenceFlow id="e1" sourceRef="Start" targetRef="A"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="A" targetRef="B"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="B" targetRef="C"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="C" targetRef="Ende"/>
```

Abb. 55: Auszug eines BPMN-Prozesses mit sequenzieller Abfolge der Tasks A, B, C vor dem Transformationsprozess

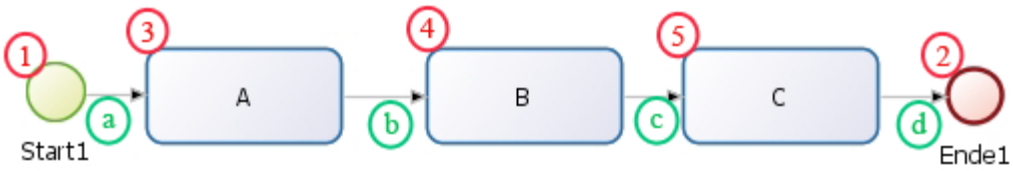
Abb. 55 zeigt einen Auszug eines BPMN-Prozesses mit drei Tasks (*bpmn:userTask* bzw. *bpmn:task* Element A, B, C) die jeweils mit einer Kante (*bpmn:sequenceFlow* Element e2, e3) sequenziell

[8] bpmn steht hier als Präfix für den Namespace in dem die Elemente eines BPMN-Prozesses definiert sind http://www.omg.org/spec/BPMN/20100524/MODEL

miteinander verbunden sind. Des Weiteren sind noch ein Start- (*bpmn:startEvent* Element Start) und eine End-Event (*bpmn:endEvent* Element Ende) im Prozess enthalten und mit der Tasksequenz verbunden (e1 bzw. e4), die den Start- und Endpunkt des Prozesses darstellen.

Transformationsphase

Am Beginn der Transformationsphase wird für jeden Task ein *bat:userPerformedPlan*[9] Element der Zwischendarstellung erzeugt. Dabei werden auch etwaige BPMN-Kantenbedingungen (*bpmn:conditionExpression* Element) berücksichtig und dafür *bat:filterConditionExpression* Kindelemente im *bat:userPerformedPlan* Element erzeugt, um später entsprechende Asbru *filter-precondition* Elemente zu erzeugen.
Abb. 56 zeigt den o.a. Prozess nachdem die Initialisierungsphase angewandt wurde. Dabei wurden *bat:plans/bat:userPerformedPlan* Elemente für jeden Task eingefügt, und eine Arbeitskopie des BPMN-Prozesses (im *bat:workingcopy* Element) erzeugt, und das *bpmn:userTask* Element für Task A wurde darin durch ein *bpmn:task* Element substituiert.




```
      <bat:plans>
        <bat:userPerformedPlan id="A" name="A"/>
        <bat:userPerformedPlan id="B" name="B"/>
        <bat:userPerformedPlan id="C" name="C"/>
      </bat:plans>
      <bat:workingcopy>...
1       <bpmn:startEvent id="Start" name="Start"/>
2       <bpmn:endEvent id="Ende" name="Ende"/>
3       <bpmn:task id="A" name="A"/>
4       <bpmn:task id="B" name="B"/>
5       <bpmn:task id="C" name="C"/>

a       <bpmn:sequenceFlow id="e1" sourceRef="Start" targetRef="A"/>
b       <bpmn:sequenceFlow id="e2" sourceRef="A" targetRef="B"/>
c       <bpmn:sequenceFlow id="e3" sourceRef="B" targetRef="C"/>
d       <bpmn:sequenceFlow id="e4" sourceRef="C" targetRef="Ende"/>
      </bat:workingcopy>...
```

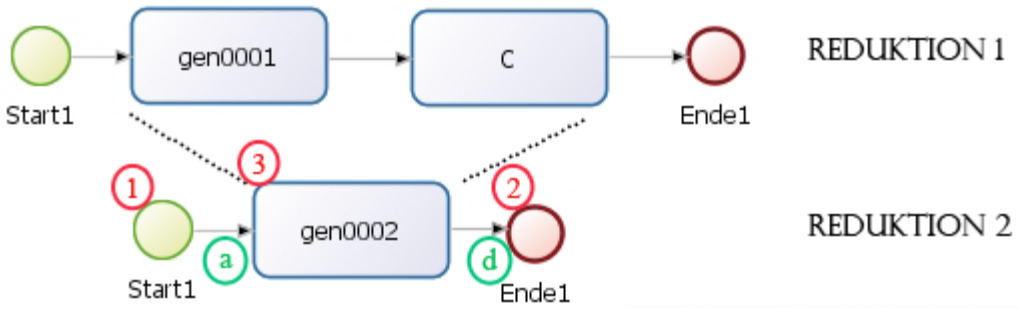
Abb. 56: Zeigt den BPMN-Prozess aus Abb. 55, nachdem die Initialisierungsphase beendet wurde. Änderungen und neue Elemente wurden fett markiert.

Anschließend wird die Verbindung zwischen zwei sequentiell ausgeführten Tasks abgebildet. Zusammenhängende Tasks charakterisieren sich, indem zwei Tasks jeweils genau eine eingehende und eine ausgehende Kante besitzen, und die ausgehende Kante des vorangegangenen Tasks die eingehende Kante des nachfolgenden Tasks ist.
Dabei werden die involvierten BPMN-Elemente (die beiden Tasks und die verbindende Kante) durch einen neuen Ersatztask ersetzt und ein Element der Zwischendarstellung (synonym: Zwischenelement) erzeugt. Als Zwischenelement dient hierfür das *bat:sequentialPlan* Element mit

[9] bat steht hier als Präfix für den Namespace der Zwischenelemente der BPMN-zu-Asbru Transformation http://www.masterarbeit/thomastschach/transform

den Kindelementen *bat:sourceRef*, das auf den Vorgängertask referenziert, und *bat:targetRef*, das auf den Nachfolgetask verweist.

Abb. 57 zeigt die wesentlichen Änderungen des BPMN-Prozesses nachdem die Mustertransformationsphase durchlaufen und zwei Reduktionsschritte angewandt wurden.




```
        <bat:plans>
          <bat:sequentialPlan id="gen0002" name="sequence gen0001 to C">
            <bat:sourceRef>gen0001</bat:sourceRef>
            <bat:targetRef>C</bat:targetRef>
          </bat:sequentialPlan>
          <bat:sequentialPlan id="gen0001" name="sequence A to B">
            <bat:sourceRef>A</bat:sourceRef>
            <bat:targetRef>B</bat:targetRef>
          </bat:sequentialPlan>
          <bat:userPerformedPlan id="A" name="A"/>
          <bat:userPerformedPlan id="B" name="B"/>
          <bat:userPerformedPlan id="C" name="C"/>
        </bat:plans>
        <bat:workingcopy>
1         <bpmn:startEvent id="Start" name="Start"/>
2         <bpmn:endEvent id="Ende" name="Ende"/>
3         <bpmn:task id="gen0002" name="gen0002"/>

a         <bpmn:sequenceFlow id="e1" sourceRef="Start" targetRef="gen0002"/>
d         <bpmn:sequenceFlow id="e4" sourceRef="gen0002" targetRef="Ende"/>
        </bat:workingcopy>
```

Abb. 57: Zeigt den BPMN-Prozess aus Abb. 55, nachdem die Mustertransformationsphase beendet wurde. Änderungen und neue Elemente wurden fett markiert

Da ein *bat:sequentialPlan* Element immer nur zwei Elemente verbinden kann, sind mehrere *bat:sequentialPlan* Elemente notwendig, um längere Abfolgen entsprechend darstellen zu können. Asbru kann Sequenzen mit drei oder mehreren aufeinanderfolgenden Plänen in einem Element darstellen. Daher werden Optimierungsschritte durchgeführt, die *bat:sequentialPlan* Elemente zu optimierten Sequenzelementen *bat:optSequentialPlan* zusammenfassen, die mit den bat:planRef Kindelementen die richtige Abfolge definieren.

```
<bat:plans>
  <bat:optSequentialPlan id="gen0002" name="gen0002">
    <bat:planRef refID="A"/>
    <bat:planRef refID="B"/>
    <bat:planRef refID="C"/>
  </bat:optSequentialPlan>
  <bat:userPerformedPlan id="A" name="A"/>
  <bat:userPerformedPlan id="B" name="B"/>
  <bat:userPerformedPlan id="C" name="C"/>
</bat:plans>
<bat:workingcopy><!-- unverändert -->...</bat:workingcopy>
```

Abb. 58: Zeigt den BPMN-Prozess aus Abb. 55 nachdem die Optimierungsphase beendet wurde. Änderungen und neue Elemente wurden fett markiert.

Abb. 58 wiederum zeigt die optimierte Sequenz in der Zwischendarstellung. Dabei wurden die beiden *bat:sequentialPlan* Elemente gen0001 und gen0002 aus der Mustertransformationsphase zu einem *bat:optSequentialPlan* Element gen0002 mit der entsprechenden Anzahl an Referenzen in der richtigen Reihenfolge zusammengefasst. Der BPMN-Prozess in der Arbeitskopie wird dabei nicht mehr verändert.

Asbru-Erzeugungsphase

bat:sequentialPlan Elemente bzw. *optSequentialPlan* Elemente können in entsprechende Asbru-Pläne übersetzt werden. Der Asbru-Plan muss dabei in seinem *plan-body* Element ein *subplans* Element enthalten, dessen *type* Attribut mit dem Wert „sequentially“ belegt ist (siehe Abb. 59). Damit wird definiert, dass die mit dem *plan-activation* Element referenzierten Subpläne (A, B, C) in der vorgegebenen sequenziellen Abfolge ausgeführt werden. Das wait-for/all Kindelement drückt aus, dass alle Subpläne erfolgreich ausgeführt werden müssen, um die Sequenz erfolgreich zu beenden.
Abb. 59 zeigt die aus den optimierten Elementen der Zwischendarstellung erzeugten Asbru XML-Elemente, die dem Mapping aus Abschnitt 4.2 bzw. Definition 14 auf Seite 75 entsprechen.

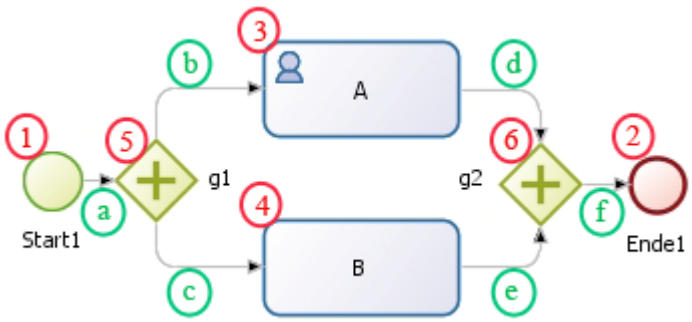
```
<plan name="gen0002" title="gen0002">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
      <plan-activation><plan-schema name="C"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A"><plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B"><plan-body><user-performed/></plan-body></plan>
<plan name="C" title="C"><plan-body><user-performed/></plan-body></plan>
```

Abb. 59: Zeigt einen Auszug der Asbru-Übersetzung des BPMN-Prozesses aus Abb. 55. Entspricht Ausgabe aus der Asbru-Generierungsphase.

4.4.1.2 Paralleler Block (Parallel Split (WCP2) / Sychronization (WCP3))

Da Asbru dem blockorientierten Kontrollflussparadigma folgt, können nur gesamte Blöcke, also Kombinationen von zwei Punkten im Kontrollfluss, die das Kontrollflusstoken auf mehrere Zweige aufteilt (Verzweigungspunkt) und wieder vereinigt (Synchronisationspunkt), übersetzt werden (siehe auch Abschnitt 4.1.1.2 und 4.1.1.3).
Beim hier implementierten parallelen Block wird der Verzweigungs- und Synchronisationspunkt jeweils durch ein paralleles BPMN-Gateway (*bpmn:parallelGateway* Element) identifiziert, die über mehrere Zweige miteinander verbunden sind. Dabei besteht jeder Zweig aus einer aus dem verzweigenden parallelen Gateway ausgehenden Kante (*bpmn:sequenceFlow* Element), die mit einem Task verbunden ist. Dieser Task ist wiederum mit dem synchronisierenden parallelen Gateway durch eine einzelne Kante verbunden. Tasks können nur einem Zweig angehören und können nicht Bestandteil mehrerer Zweige sein. Genauer ist dies in Definition 14 auf Seite 75 erläutert. Der hier beschriebene parallele Block realisiert am Verzweigungspunkt das Parallel Split Pattern (WCP2) und am Synchronisationspunkt das Synchronization Pattern (WCP3).

Abb. 60 zeigt einen Parallelen Block in der BPMN XML-Repräsentation, bei dem das verzweigende parallele Gateway g1 über zwei Zweige mit dem synchronisierenden Gateway g2 verbunden ist. Ein Zweig verbindet g1 über Kante e2, Task A und Kante e4 mit Gateway g2. Der andere Zweig wird durch Kante e3, Task B und Kante e5 gebildet. Somit werden die Tasks A und B parallel ausgeführt, und g2 synchronisiert die Kontrollflusstoken der beiden Zweige.



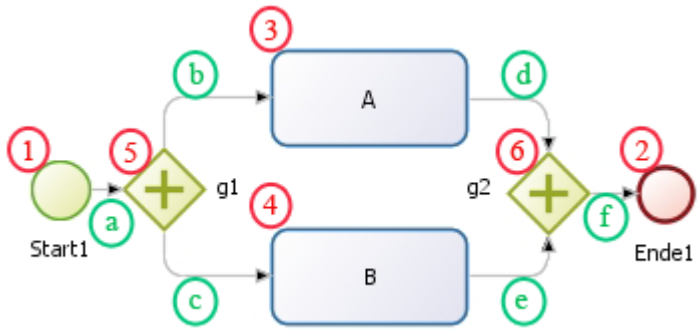

```
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:endEvent id="Ende1" name="Ende1"/>
3   <bpmn:userTask id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
5   <bpmn:parallelGateway id="g1" name="g1"/>
6   <bpmn:parallelGateway id="g2" name="g2"/>

a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="g1" targetRef="B"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e   <bpmn:sequenceFlow id="e5" sourceRef="B" targetRef="g2"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="g2" targetRef="Ende1"/>
```

Abb. 60: Auszug eines BPMN-Prozesses mit parallelem Block mit den Tasks A, B vor dem Transformationsprozess

Transformationsphase

Bei der Initialisierung werden für die diversen Task-Elemente *bat:userPerformedPlan* Elemente erzeugt und die diversen Task-Varianten in der Arbeitskopie vereinheitlicht, und dabei werden auch etwaige BPMN-Kantenbedingungen (*bpmn:conditionExpression* Element) berücksichtig.

Abb. 61 zeigt, dass in der Initialisierungsphase für jeden Task ein bat:plans/*bat:userPerformedPlan* Element eingefügt, eine Arbeitskopie (*bat:workingcopy*) des BPMN-Prozesses erzeugt und das *bpmn:userTask* Element für Task A darin durch ein *bpmn:task* Element ersetzt wurde.



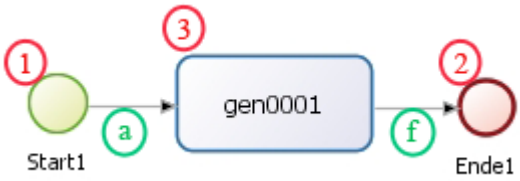

```
      ...<bat:plans>
        <bat:userPerformedPlan id="A" name="A"/>
        <bat:userPerformedPlan id="B" name="B"/>
      </bat:plans>
      <bat:workingcopy>
1       <bpmn:startEvent id="Start1" name="Start1"/>
2       <bpmn:endEvent id="Ende1" name="Ende1"/>
3       <bpmn:task id="A" name="A"/>
4       <bpmn:task id="B" name="B"/>
5       <bpmn:parallelGateway id="g1" name="g1"/>
6       <bpmn:parallelGateway id="g2" name="g2"/>

a       <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b       <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A"/>
c       <bpmn:sequenceFlow id="e3" sourceRef="g1" targetRef="B"/>
d       <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e       <bpmn:sequenceFlow id="e5" sourceRef="B" targetRef="g2"/>
g       <bpmn:sequenceFlow id="e6" sourceRef="g2" targetRef="Ende1"/>
      </bat:workingcopy>...
```

Abb. 61: Zeigt den BPMN-Prozess aus Abb. 60, nachdem die Initialisierungsphase beendet wurde. Änderungen und neue Elemente wurden fett markiert.

Ein paralleler Block charakterisiert sich dadurch, dass ein verzweigendes paralleles Gateway über einen oder mehrere Zweig(e) mit einem synchronisierenden parallelen Gateway verbunden ist. Das verzweigende parallele Gateway hat genau eine eingehende Kante und, entsprechend der Anzahl der Zweige des parallelen Blocks, eine oder mehrere ausgehende Kante(n). Das synchronisierende parallele Gateway hat, entsprechend der Anzahl der Zweige des parallelen Blocks, eine oder mehrere eingehende Kante(n) und genau eine ausgehende Kante.

Die einzelnen Zweige bestehen aus einem Task, der genau eine eingehende, die aus dem verzweigenden Gateway entspringt, und genau eine ausgehende Kante besitzt, die im

synchronisierenden Gateway endet. Enthalten die Zweige andere Muster (z.B. Verschachtelung von Blöcken), müssen diese zuerst durch vorangegangene Reduktionsschritte der Mustertransformationsphase reduziert werden und durch einen Ersatztask ersetzt werden.
Wurde nun ein paralleler Block identifiziert, wird ein *bat:parallelPlan* Element als Element der Zwischendarstellung erzeugt (Abb. 62), dessen *bat:planRef* Kindelemente die übersetzten Tasks der einzelnen Zweige referenzieren.
Das verzweigende und synchronisierende parallele Gateway wird durch einen neuen Ersatztask ersetzt, und die Kante, die vorher auf das verzweigende parallele Gateway g1 wies, verweist nun auf den neuen Ersatztask. Kanten, die aus dem synchronisierenden parallelen Gateway g2 führten, entspringen nun aus dem Ersatztask.




```
...<bat:plans>
  <bat:parallelPlan id="gen0001" name="parallel Block A B">
    <bat:planRef refID="A"/>
    <bat:planRef refID="B"/>
  </bat:parallelPlan>
  <bat:userPerformedPlan id="A" name="A"/>
  <bat:userPerformedPlan id="B" name="B"/>
</bat:plans>
<bat:workingcopy>
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:endEvent id="Ende1" name="Ende1"/>
3   <bpmn:task id="gen0001" name="gen0001"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="gen0001" targetRef="Ende1"/>
</bat:workingcopy>...
```

Abb. 62: Zeigt den BPMN-Prozess aus Abb. 60 nachdem Reduktionsschritt der Mustertransformationsphase. Änderungen und neue Elemente wurden fett markiert.

Asbru-Erzeugungsphase

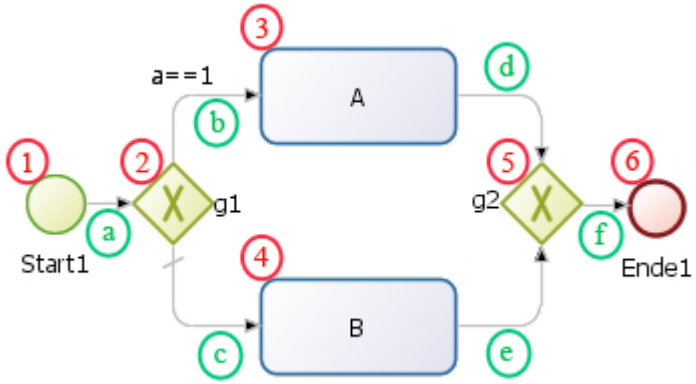
bat:parallelPlan Elemente werden in Asbru-Pläne mit einem *subplans* Element transformiert, dessen *type* Attribut mit dem Wert „parallel“ belegt ist. Jene Pläne, die die Tasks in den Zweigen repräsentieren, werden referenziert. Die wesentlichen Asbru-Elemente, mit denen der parallele Block übersetzt wird, sind in Abb. 63 zu sehen.

```
<plan name="gen0001" title="parallel Block A B">
  <plan-body>
    <subplans type="parallel">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A"><plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B"><plan-body><user-performed/></plan-body></plan>
```

Abb. 63: Zeigt einen Auszug der Asbru-Übersetzung des BPMN-Prozesses aus Abb. 60. Entspricht Ausgabe der Asbru-Erzeugungsphase

4.4.1.4 Exklusive Block (Exclusive Choice (WCP4) / Simple Merge (WCP5))

Der exklusive Block kombiniert das Exclusive Choice Pattern (WCP4) am Verzweigungspunkt mit dem Simple Merge Pattern (WCP5) am Synchronisationspunkt (siehe auch Abschnitt 4.1.1.4 und 4.1.1.5). In BPMN wird dies mit exklusiven Gateways (bpmn:exclusiveGateway Element) realisiert. Eines muss den Kontrollfluss verzweigen (g1) und eines wieder verschmelzen (g2) (siehe Abb. 64). Dabei ist jeder Zweig ausgehend vom Verzweigungspunkt über einen Task mit dem Synchronisationspunkt verbunden. Jeder Zweig besteht aus einem Task, dessen eingehende Kante den Ursprung im Verzweigungspunkt hat und dessen ausgehende Kante im Synchronisationspunkt endet (siehe auch Definition 14 S.75).




```
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:exclusiveGateway id="g1" name="g1"/>
3   <bpmn:task id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
5   <bpmn:exclusiveGateway id="g2" name="g2"/>
6   <bpmn:endEvent id="Ende1" name="Ende1"/>

a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
    <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A">
b     <bpmn:conditionExpression>a==1</bpmn:conditionExpression>
    </bpmn:sequenceFlow>
c   <bpmn:sequenceFlow id="e3" sourceRef="g1" targetRef="B"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e   <bpmn:sequenceFlow id="e5" sourceRef="B" targetRef="g2"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="g2" targetRef="Ende1"/>
```

Abb. 64: Auszug eines BPMN-Prozesses mit exklusivem Block mit den Tasks A, B vor dem Transformationsprozess

Abb. 64 zeigt einen exklusiven Block in der BPMN XML-Repräsentation, bei dem das verzweigende exklusive Gateway g1 über zwei Zweige mit dem synchronisierenden Gateway g2 verbunden ist. Ein Zweig verbindet g1 über Kante e2, Task A und Kante e4 mit Gateway g2. Der andere Zweig wird durch Kante e3, Task B und Kante e5 gebildet.

Transformationsphase

Nach den üblichen Initialisierungsmaßnahmen wurden die Tasks A und B transformiert und darin eingehende Kantenbedingungen in den Zwischenelementen berücksichtigt.
Ein exklusiver Block charakterisiert sich dadurch, dass ein verzweigendes exklusives Gateway über einen oder mehrere Zweig(e) mit einem synchronisierenden exklusiven Gateway verbunden ist. Das

verzweigende exklusive Gateway hat genau eine eingehende Kante und, entsprechend der Anzahl der Zweige des exklusiven Blocks, eine oder mehrere ausgehende Kante(n). Das synchronisierende exklusive Gateway hat, entsprechend der Anzahl der Zweige des exklusiven Blocks, eine oder mehrere eingehende Kante(n) und genau eine ausgehende Kante. Die einzelnen Zweige bestehen aus einem Task, der genau eine eingehende, die aus dem verzweigenden Gateway entspringt, und genau eine ausgehende Kante besitzt, die im synchronisierenden Gateway endet. Enthalten die Zweige andere Muster (z.B. Verschachtelung von Blöcken), müssen diese zuerst durch vorangegangene Reduktionsschritte der Mustertransformationsphase reduziert und dabei durch einen Ersatztask ersetzt werden.

Wird ein exklusiver Block identifiziert, wird ein *bat:exclusivePlan* Element als Element der Zwischendarstellung erzeugt (Abb. 65), dessen *bat:planRef* Kindelemente die übersetzten Tasks der einzelnen Zweige referenzieren. Ähnlich wie beim vorangegangenen Block werden die Gateways durch einen Task ersetzt, und Kanten entsprechend angepasst. Da die relevanten Kantenbedingungen schon in der Initialisierungsphase bzw. einem vorangegangen Reduktionsschritt übersetzt wurden, müssen sie an dieser Stelle nicht mehr berücksichtigt werden.


```
    ...<bat:plans>
      <bat:exclusivePlan id="gen0001" name="exclusive Block A B">
        <bat:planRef refID="A"/>
        <bat:planRef refID="B"/>
      </bat:exclusivePlan>
      <bat:userPerformedPlan id="A" name="A">
        <bat:filterConditionExpression>a==1</bat:filterConditionExpression>
      </bat:userPerformedPlan>
      <bat:userPerformedPlan id="B" name="B"/>
    </bat:plans>
    <bat:workingcopy>
1     <bpmn:startEvent id="Start1" name="Start1"/>
2     <bpmn:task id="gen0001" name="gen0001"/>
3     <bpmn:endEvent id="Ende1" name="Ende1"/>
a     <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
f     <bpmn:sequenceFlow id="e6" sourceRef="gen0001" targetRef="Ende1"/>
    </bat:workingcopy>...
```

Abb. 65: Zeigt den BPMN-Prozess aus Abb. 64, nachdem der exklusive Block reduziert wurde. Änderungen und neue Elemente wurden fett hervorgehoben.

Asbru-Erzeugungsphase

bat:exclusivePlan Elemente können in entsprechende Asbru-Pläne übersetzt werden. Der Asbru-Plan muss dabei in seinem *plan-body* Element ein *subplans* Element enthalten, dessen *type* Attribut mit dem Wert „any-order“ belegt ist. Außerdem muss das *wait-for* Element ein *one* Kindelement aufweisen, um die richtige Fortsetzungsstrategie zu konfigurieren. Jene Pläne, die die Tasks in den Zweigen repräsentieren, werden mit *plan-activation* bzw. *plan-schema* Elementen referenziert.
Da beim exklusiven Block erstmalig Kantenbedingungen relevant sind, weisen die referenzierten Pläne Filterbedingungen (*filter-precondition* Element in Plan A) auf. Der Ausdruck, der die Filterbedingungen darstellt, wird als Kommentar (*comment* Element) angezeigt und wird nicht

automatisch[10] übersetzt. Die wesentlichen Asbru-Elemente, mit denen der exklusive Block übersetzt worden ist, sind in Abb. 66 zu sehen.

```
<plan name="gen0001" title="exclusive Block A B">
  <plan-body>
    <subplans type="any-order">
      <wait-for><one/><wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A">
  <conditions><filter-precondition>     <comment text="a==1"/>
  </filter-precondition></conditions>
  <plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B"><plan-body><user-performed/></plan-body></plan>
```

Abb. 66: Auszug der Asbru-Übersetzung des BPMN-Prozesses aus Abb. 64. Entspricht der Ausgabe der Asbru-Generierungsphase

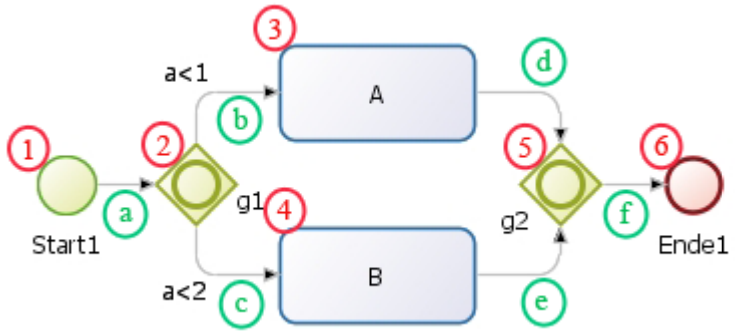
4.4.2 Erweiterte Verzweigungs- und Synchronisations- Kontrollflussmuster

Die erweiterten Verzweigungs- und Synchronisations-Kontrollflussmuster enthalten das Multiple Choice, Synchronising Merge, Multimerge und das Discriminator Pattern. Da das Multimerge Pattern von Asbru nicht unterstützt wird, wird es in diesem Abschnitt auch nicht behandelt.

4.4.2.1 Inklusiver Block (Multiple Choice (WCP6) / Sychronising Merge (WCP7))

Der inklusive Block kombiniert am Verzweigungspunkt das Multiple Choice Pattern (WCP6) mit dem Synchronising Merge Pattern (WCP7) am Synchronisationspunkt (siehe auch Abschnitt 4.1.2.1 und 4.1.2.2).
Der Aufbau und die Übersetzung dieses Blocks ähnelt dem exklusiven Block in Kapitel 4.4.1.4, deshalb wird hier nur auf die Unterschiede zum exklusiven Block hingewiesen. In BPMN wird der inklusive Block, im Unterschied zum exklusiven Block, mit inklusiven Gateways (*bpmn:inclusiveGateway* Element) realisiert, eines das den Kontrollfluss verzweigt (g1) und eines das ihn wieder verschmilzt (g2) (siehe Abb. 67 und auch Definition 14 S.75).

[10] Da das Transformieren einer BPMN-Kantenbedingung nicht im Umfang dieser Arbeit enthalten ist, muss eine gültige Asbru-Bedingung manuell durch den Benutzer erstellt werden.




```
1       <bpmn:startEvent id="Start1" name="Start1"/>
2       <bpmn:inclusiveGateway id="g1" name="g1"/>
3       <bpmn:task id="A" name="A"/>
4       <bpmn:task id="B" name="B"/>
5       <bpmn:inclusiveGateway id="g2" name="g2"/>
6       <bpmn:endEvent id="Ende1" name="Ende1"/>

a       <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
        <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A">
b         <bpmn:conditionExpression>a&lt;1</bpmn:conditionExpression>
        </bpmn:sequenceFlow>
        <bpmn:sequenceFlow id="e3" sourceRef="g1" targetRef="B"/>
c           <bpmn:conditionExpression>a&lt;2</bpmn:conditionExpression>
        </bpmn:sequenceFlow>
d       <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e       <bpmn:sequenceFlow id="e5" sourceRef="B" targetRef="g2"/>
f       <bpmn:sequenceFlow id="e6" sourceRef="g2" targetRef="Ende1"/>
```

Abb. 67: Auszug eines BPMN-Prozesses mit inklusivem Block mit den Tasks A, B vor dem Transformationsprozess

Abb. 67 zeigt einen inklusiven Block in der BPMN XML-Repräsentation bei dem das verzweigende inklusive Gateway g1 über zwei Zweige mit dem synchronisierenden Gateway g2 verbunden ist. Ein Zweig verbindet g1 über Kante e2, Task A und Kante e4 mit Gateway g2. Der andere Zweig wird durch Kante e3, Task B und Kante e5 gebildet.

Transformationsphase

Nach den üblichen Initialisierungsmaßnahmen erfolgt die Transformation ähnlich wie beim exklusiven Block im vorangegangenen Abschnitt 4.4.1.4. Allerdings wird ein *bat:inclusivePlan* Element als Element der Zwischendarstellung erzeugt (Abb. 68).


```
...<bat:plans>
  <bat:inclusivePlan id="gen0001" name="inclusive Block A B">
    <bat:planRef refID="A"/>
    <bat:planRef refID="B"/>
  </bat:inclusivePlan>
  <bat:userPerformedPlan id="A" name="A">
    <bat:filterConditionExpression>a&lt;1</bat:filterConditionExpression>
  </bat:userPerformedPlan>
  <bat:userPerformedPlan id="B" name="B">
    <bat:filterConditionExpression>a&lt;2</bat:filterConditionExpression>
  </bat:userPerformedPlan>
</bat:plans>
<bat:workingcopy>
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task id="gen0001" name="gen0001"/>
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="gen0001" targetRef="Ende1"/>
</bat:workingcopy>...
```

Abb. 68: Zeigt den BPMN-Prozess aus Abb. 67, nachdem der inklusive Block in der Mustertransformationsphase reduziert wurde. Änderungen und neue Elemente wurden fett hervorgehoben.

```
<plan name="gen0001" title="inclusive Block A B">
  <plan-body>
    <subplans type="unordered" wait-for-optional-subplans="yes">
      <wait-for><one/><wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A">
  <conditions><filter-precondition>    <comment text="a&lt;1"/>
  </filter-precondition></conditions>
  <plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B">
  <conditions><filter-precondition>    <comment text="a&lt;2"/>
  </filter-precondition></conditions>
  <plan-body><user-performed/></plan-body></plan>
```

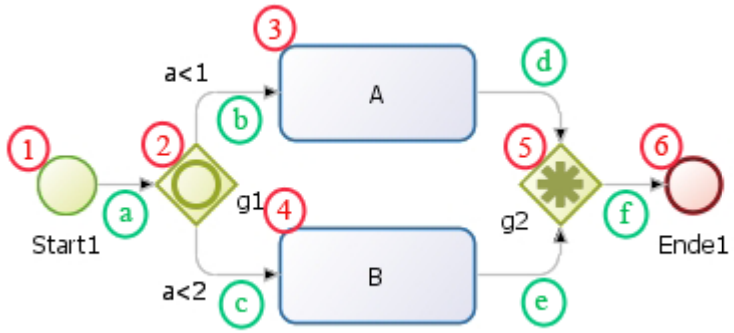
Abb. 69: Auszug der Asbru-Übersetzung des BPMN-Prozesses aus Abb. 67. Entspricht der Ausgabe der Asbru-Generierungsphase

Asbru-Erzeugungsphase

bat:inclusivePlan Elemente werden, ähnlich wie beim exklusiven Block, in Asbru-Pläne, die ein *subplans* Element enthalten, transformiert, dessen *type* Attribut jedoch mit dem Wert „*unordered*“ und dessen *wait-for-optional-subplans* Attribut mit dem Wert „*yes*“ belegt ist. Das *wait-for* Element weist ein *one* Kindelement auf und unterscheidet sich nicht vom exklusiven Block. Die wesentlichen Asbru-Elemente, mit denen der inklusive Block übersetzt wird, sind in Abb. 69 zu sehen.

4.4.2.2 Discriminator Block (Multiple Choice (WCP6) / Discriminator (WCP9))

Der Discriminator Block kombiniert das Parallel Split (WCP 2) oder das Multiple Choice Pattern (WCP6) am Verzweigungspunkt mit dem Discriminator Pattern (WCP9) am Synchronisationspunkt (siehe auch Abschnitt 4.1.2.1 und 4.1.2.4).




```
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:inclusiveGateway id="g1" name="g1"/>
3   <bpmn:task id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
    <bpmn:complexGateway id="g2" name="g2">
5     <bpmn:activationCondition>one</bpmn:activationCondition>
    </bpmn:complexGateway >
6   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
    <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A">
b     <bpmn:conditionExpression>a&lt;1</bpmn:conditionExpression>
    </bpmn:sequenceFlow>
    <bpmn:sequenceFlow id="e3" sourceRef="g1" targetRef="B"/>
c     <bpmn:conditionExpression>a&lt;2</bpmn:conditionExpression>
    </bpmn:sequenceFlow>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e   <bpmn:sequenceFlow id="e5" sourceRef="B" targetRef="g2"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="g2" targetRef="Ende1"/>
```

Abb. 70: Auszug eines BPMN-Prozesses mit Discriminator Block mit den Tasks A, B vor dem Transformationsprozess. Unterschiede zum inklusiven Block sind fett hervorgehoben.

Der Aufbau und die Übersetzung dieses Blocks ähnelt dem exklusiven Block in Kapitel 4.4.1.4, deshalb wird hier nur auf die Unterschiede zum exklusiven Block hingewiesen. In BPMN wird der Discriminator Block, im Unterschied zum exklusiven Block, mit einem parallelen oder inklusiven Gateway (*bpmn:parallelGateway* bzw. *bpmn:inclusiveGateway* Element) am Verzweigungspunkt (g1)

und einem komplexen Gateway (*bpmn:complexGateway* Element) am Synchronisationspunkt (g2) realisiert (Abb. 70). Das komplexe Gateway beinhaltet ein *bpmn:activationCondition* Element mit dem die Bedingung zur Aktivierung des Gateways angegeben wird. Unklar ist jedoch, wie diese Bedingung in BPMN auszusehen hat, damit nach Eintreffen des ersten Tokens die Nachfolgeaktivität aktiviert wird, und Token alternativer Zweige deaktiviert werden.

Abb. 70 zeigt einen Discriminator Block in der BPMN XML-Repräsentation, bei dem das verzweigende inklusive Gateway g1 über zwei Zweige mit dem synchronisierenden komplexen Gateway g2 verbunden ist. Ein Zweig verbindet g1 über Kante e2, Task A und Kante e4 mit Gateway g2. Der andere Zweig wird durch Kante e3, Task B und Kante e5 gebildet. Durch das bpmn: *activationCondition* Element mit Textinhalt „one“ in Gateway g2 wird das Discriminator Pattern gekennzeichnet (Konvention für den Prototyp), wofür es jedoch keine grafische Darstellung gibt.

Transformationsphase

Nach den üblichen Initialisierungs- und Vereinheitlichungsmaßnahmen erfolgt die Mustertransformation ähnlich wie beim exklusiven Block (Abschnitt 4.4.1.4).

Unterschiede ergeben sich bei der Identifizierung des Discriminator Blocks, da dieser durch ein verzweigendes paralleles oder inklusives Gateway (*bpmn:parallelGateway* bzw. *bpmn:inclusiveGateway* Element) und ein synchronisierendes komplexes Gateway (*bpmn:complexGateway* Element), das ein *bpmn:activationCondition* Kindelement mit dem Textinhalt „one“ beinhaltet, gebildet wird.

Die Reduktion erfolgt ähnlich wie beim exklusiven Block, nur wird hier ein *bat:discriminatorPlan* Element als Element der Zwischendarstellung erzeugt, wobei das *type* Attribut den Typ des verzweigenden Gateways des Discriminator Blocks („parallel“ oder „inclusive“) angibt (Abb. 71).


```
...<bat:plans>
  <bat:discriminatorPlan id="gen0001" name="discriminator A B" type=”inclusive”>
    <bat:planRef refID="A"/>
    <bat:planRef refID="B"/>
  </bat: discriminatorPlan >
  <bat:userPerformedPlan id="A" name="A">
    <bat:filterConditionExpression>a&lt;1</bat:filterConditionExpression>
  </bat:userPerformedPlan>
  <bat:userPerformedPlan id="B" name="B">
    <bat:filterConditionExpression>a&lt;2</bat:filterConditionExpression>
  </bat:userPerformedPlan>
</bat:plans>
<bat:workingcopy>
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task id="gen0001" name="gen0001"/>
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="gen0001" targetRef="Ende1"/>
</bat:workingcopy>...
```

Abb. 71: Zeigt den BPMN-Prozess aus Abb. 70 nachdem der Discriminator Block reduziert wurde. Änderungen und neue Elemente wurden fett hervorgehoben.

Asbru-Erzeugungsphase

bat:discriminatorPlan Elemente werden in Asbru-Pläne mit einem plan-body/subplans Element übersetzt, dessen type Attribut mit dem Wert „unordered“ (bzw. „parallel“ bei einem verzweigenden parallelen Gateway) belegt ist. Das wait-for Element weist ein one Kindelement auf und unterscheidet sich nicht vom exklusiven Block. Die wesentlichen Asbru-Elemente, mit denen der Discriminator Block übersetzt worden ist, sind in Abb. 72 zu sehen.

```
<plan name="gen0001" title="inclusive Block A B">
  <plan-body>
    <subplans type="unordered" >
      <wait-for><one/><wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A">
  <conditions><filter-precondition>    <comment text="a&lt;1"/>
  </filter-precondition></conditions>
  <plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B">
  <conditions><filter-precondition>    <comment text="a&lt;2"/>
  </filter-precondition></conditions>
  <plan-body><user-performed/></plan-body></plan>
```

Abb. 72: Auszug der Asbru-Übersetzung des BPMN-Prozesses aus Abb. 67. Entspricht der Ausgabe der Asbru-Erzeugungsphase

4.4.3 Multiple-Instanzen (MI) Kontrollflussmuster

Die Multiple-Instanzen (MI) Kontrollflussmuster enthalten das *MI without synchronization*, *MI with a priori design-time knowledge*, *MI with a priori run-time knowledge* und das *MI without a priori run-time knowledge* Pattern. Asbru bietet für keines dieser Muster eine direkte Umsetzung an. Nur für das MI with a priori design-time knowledge Pattern kann eine Lösung simuliert werden. Daher wird im folgenden Unterabschnitt nur diese Workaround-Lösung erörtert.

4.4.3.1 MI mit a priori Design-Time Wissen (MI with a priori design-time knowledge WCP13)

Mit diesem Muster wird ausgedrückt, dass innerhalb eines Prozesses oder Subprozesses mehrere Instanzen einer Aktivität erstellt werden können, und nach Beendigung aller Instanzen (synchronisierend) der Kontrollfluss weitergeführt wird. Dabei ist die Anzahl der Instanzen schon zur Design-Time bekannt (siehe auch Abschnitt 4.1.4.2).
Ein *MI mit a priori Design-Time Wissen* Muster wird in BPMN dadurch charakterisiert, dass (1) ein Task ein *bpmn:multiInstanceLoopCharacteristics* Kindelement aufweist, und (2) der Wert von dessen *behaviour* Attribut den Wert „*All*“ hat (siehe Abb. 73). Das *isSequential* Attribut gibt an, ob die einzelnen Instanzen sequentiell oder parallel ausgeführt werden sollen, und (3) über das *bpmn:loopCardinality* Element muss die Anzahl der benötigten Instanzen angegeben werden.

Asbru kann Mehrfachausführungen im Allgemeinen nicht umsetzen. Ist die Anzahl der zu erzeugenden Instanzen zur Design-Time bekannt, und findet eine Synchronisation statt, wie bei diesem Muster, dann kann es in Asbru simuliert werden. Dies geschieht, indem ein übergeordneter Plan (A bzw. B) erstellt wird, der, entsprechend der Anzahl der benötigten Instanzen, die gewünschte Anzahl an Referenzen auf einen konkreten Subplan (A_Instance bzw. B_Instance) enthält. Dieser Subplan repräsentiert die eigentliche Aktivität, und die erzeugten Instanzen werden vom referenzierende übergeordneten Plan entweder parallel oder sequentiell ausgeführt (siehe Abb. 76).


```
<bpmn:task id="A" name="A">
  <bpmn:multiInstanceLoopCharacteristics isSequential="false" behavior="All">
1   <bpmn:loopCardinality>5</bpmn:loopCardinality>
  </bpmn:multiInstanceLoopCharacteristics>
</bpmn:task>

<bpmn:task id="B" name="B">
  <bpmn:multiInstanceLoopCharacteristics isSequential="true" behavior="All">
2   <bpmn:loopCardinality>5</bpmn:loopCardinality>
  </bpmn:multiInstanceLoopCharacteristics>
</bpmn:task>
```

Abb. 73: oben parallele unten sequentielle Mehrfachausführung eines Tasks in BPMN-Notation

Abb. 73 zeigt oben einen Task mit paralleler (*isSequential* Attribut "*false*") Mehrfachausführung (Task A) und unten einen Task mit sequentieller (*isSequential* Attribut "*true*") Mehrfachausführung (Task B). Für beide Tasks sollen jeweils fünf Instanzen (*bpmn:loopCardinality* = 5) zur Laufzeit erzeugt werden. Es wird die Beendigung aller 5 Instanzen abgewartet (*behavior* Attribut = "*All*"), um den Kontrollfluss weiterzuführen.


```
<bat:plans/>
<bat:workingcopy>...
  <bat:multipleInstanceTask isSequential="false" id="A" name="A">
    <bpmn:multiInstanceLoopCharacteristics isSequential="false" behavior="All">
1     <bpmn:loopCardinality>5</bpmn:loopCardinality>
    </bpmn:multiInstanceLoopCharacteristics>
  </bat:multipleInstanceTask>
  <bat:multipleInstanceTask isSequential="true" id="B" name="B">
    <bpmn:multiInstanceLoopCharacteristics isSequential="true" behavior="All">
2     <bpmn:loopCardinality>5</bpmn:loopCardinality>
    </bpmn:multiInstanceLoopCharacteristics>
  </bat:multipleInstanceTask>...
</bat:workingcopy>
```

Abb. 74: Tasks mit Mehrfachausführung aus Abb. 73 nach der Initialisierung. Änderung und neue Elemente wurden fett hervorgehoben.

Transformationsphase

Bei der Initialisierung wird in der Arbeitskopie (*bat:workingcopy*) jedes *bpmn:task* Element, das die o.a. Merkmale aufweist, durch ein *bat:multipleInstanceTask* Element ersetzt, wobei die Attribute und Kindelemente des Tasks unverändert übernommen werden. Abb. 74 veranschaulicht die o.a. Änderungen an den Tasks mit Mehrfachausführung aus Abb. 73

Bei der Mustertransformation wird für jedes *bat:multipleInstanceTask* Element mit paralleler Mehrfachausführung (*isSequential* Attribut "*false*") ein *bat:userPerformedPlan* Element und ein *bat:parallelPlan* Element als Element der Zwischendarstellung erzeugt. Das *bat:userPerformedPlan* Element repräsentiert die Basisaktivität, von der es mehrere Instanzen geben soll. Das *bat:parallelPlan* Element, wird für die parallelen Ausführungsarten erzeugt, referenziert das *bat:userPerformedPlan* Element so oft, wie im *bpmn:loopCardinality* Element angegeben.
Abb. 75 zeigt, dass bei der Reduktion erzeugte *bat:parallelPlan* Element, das für Task A mit paralleler Mehrfachausführung erzeugt wurde, und dessen fünf *bat:planRef* Kindelemente, die auf das *bat:userPerformedPlan* Element *A_Instance* verweisen.


```
<bat:plans>
  <bat:parallelPlan id="A" name="A">
    <bat:planRef refID="A_Instance"/>
    <bat:planRef refID="A_Instance"/>
    <bat:planRef refID="A_Instance"/>
    <bat:planRef refID="A_Instance"/>
    <bat:planRef refID="A_Instance"/>
  </bat:parallelPlan >
  <bat:userPerformedPlan id="A_Instance" name="A_Instance"/>

  <bat:optSequentialPlan id="B" name="B">
    <bat:planRef refID="B_Instance"/>
    <bat:planRef refID="B_Instance"/>
    <bat:planRef refID="B_Instance"/>
    <bat:planRef refID="B_Instance"/>
    <bat:planRef refID="B_Instance"/>
  </bat:optSequentialPlan>
  <bat:userPerformedPlan id="B_Instance" name="B_Instance"/>
</bat:plans>
<bat:workingcopy>...
1   <bpmn:task id="A" name="A">
2   <bpmn:task id="B" name="B">
</bat:workingcopy>...
```

Abb. 75: Tasks aus Abb. 73 nachdem beide durch die Mustertransformation reduziert wurden. Änderungen und neue Elemente wurden fett hervorgehoben.

Die Transformation des *bat:multipleInstanceTask* Element mit sequentieller Mehrfachausführung (*isSequential* Attribut "*true*") erfolgt ähnlich wie bei der parallelen Variante. Es wird jedoch statt dem *bat:parallelPlan* ein *bat:optSequentialPlan* Element erzeugt.
Für Task B mit sequentieller Mehrfachausführung wurde ein *bat:optSequentialPlan* Element (Abb. 75) erzeugt, das fünf Mal auf Element *B_Instance* verweist.

Asbru-Erzeugungsphase

Die Asbru-Erzeugungsphase übersetzt die *bat:userPerformedPlan, bat:parallelPlan* und *bat:optSequentialPlan* Elemente wie schon in Abschnitt 4.4.1.1 bzw. Abschnitt 4.4.1.2 beschrieben.

Abb. 76 zeigt die wesentlichen Asbru-Elemente, mit denen die Tasks A und B aus Abb. 73 übersetzt wurden.

```
<plan name="A" title="A">
  <plan-body>
    <subplans type="parallel">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A_Instance"/></plan-activation> <!--5-mal -->
      ...
    </subplans>
  </plan-body>
</plan>
<plan name="A_Instance" title="A_Instance">
  <plan-body><user-performed/></plan-body>
</plan>

<plan name="B" title="B">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="B_Instance"/></plan-activation> <!--5-mal -->
      ...
    </subplans>
  </plan-body>
</plan>
<plan name="B_Instance" title="B_Instance">
  <plan-body><user-performed/></plan-body>
</plan>
```

Abb. 76: Auszug aus Asbru-Übersetzung der Task aus Abb. 73

4.4.4 Zustandsbasierte Kontrollflussmuster

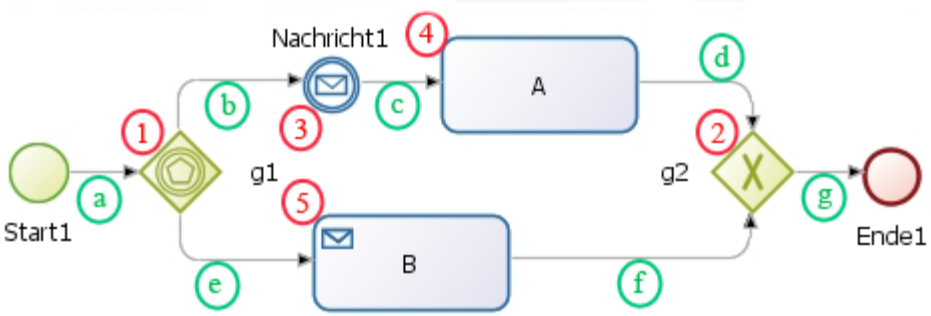
Die Zustandsbasierten Kontrollflussmuster enthalten das Deferred Choice, Interleaved Parallel Routing und das Milestone Pattern. Das Milestone Pattern wird von Asbru nicht unterstützt. Die Beschreibung der ersten beiden Patterns erfolgt in den folgenden Abschnitten.

4.4.4.1 Deferred Choice (WCP16)

Mit dem Deferred Choice Pattern kann ein Verzweigungspunkt im Prozess definiert werden, dessen Entscheidung, welcher Zweig aktiviert werden soll, von der Umgebung beeinflusst wird und nicht vom Prozess selbst abhängt (siehe auch Abschnitt 4.1.3.1). Z.B. wird durch eine Benutzereingabe entschieden, welcher Zweig ausgeführt werden soll.

BPMN realisiert dieses Muster, indem ein Ereignis-basiertes Gatway (*bpmn:eventBasedGateway* Element) am Verzweigungspunkt mit einem exklusiven Gateway (*bpmn:exclusiveGateway* Element) am Synchronisationspunkt zu einem Block kombiniert wird (siehe Abb. 77 und auch Definition 14 S.75). Die einzelnen Zweige des Blocks können ein Zwischenereignis (werfend *bpmn:intermediateThrowEvent* Element oder konsumierent: *bpmn:intermediateCatchEvent* Element), ein Zwischenereignis in sequentieller Abfolge mit einem Task (auch Varianten davon), einem Empfangen (*bpmn:receiveTask* Element) oder Senden Task (*bpmn:sendTask* Element) enthalten. Abhängig vom *eventGatewayTyp* Attributwert des *bpmn:eventBasedGateway* Element,

darf nur ein einzelner Zweig des Deferred Choice Blocks (eventGatewayTyp="Exclusive") oder können mehrere Zweige parallel aktiviert werden (eventGatewayTyp="Parallel").
Asbru hat die Möglichkeit, die Ausführung eines Plans von der Bestätigung durch einen Benutzer abhängig zu machen. Dies erfolgt über das *confirmation-required* Attribut der Filterbedingung (*filter-precondition* Element) des Plans (siehe Abb. 81). Auf diese Weise kann Asbru das Deferred Choice Pattern umsetzen.



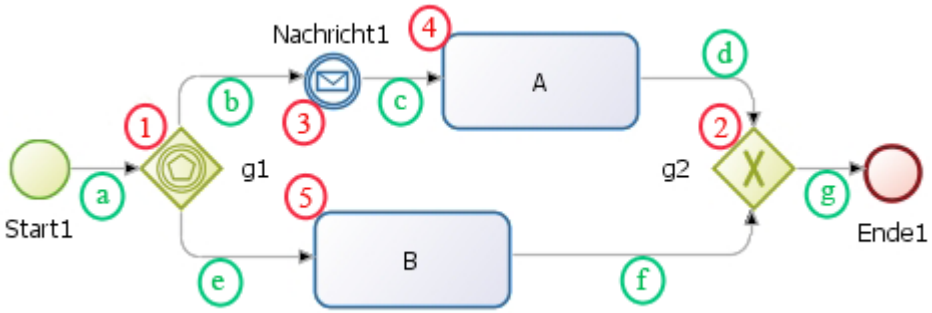

```
...
1   <bpmn:eventBasedGateway id="g1" name="g1"
        eventGatewayType="Exclusive" instantiate="true"/>
2   <bpmn:exclusiveGateway id="g2" name="g2"/>

3   <bpmn:intermediateCatchEvent id="Nachricht1" name="Nachricht1"/>
4   <bpmn:task id="A" name="A"/>
5   <bpmn:receiveTask id="B" name="B"/>

a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="Nachricht1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Nachricht1" targetRef="A"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e   <bpmn:sequenceFlow id="e5" sourceRef="g1" targetRef="B"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="B" targetRef="g2"/>
g   <bpmn:sequenceFlow id="e7" sourceRef="g2" targetRef="Ende1"/>...
```

Abb. 77: Deferred Choice Block mit einem konsumierenden Ereignis Nachricht1 gefolgt von Task A (Zweig 1) und einem Empfang Task B (Zweig 2)

Abb. 77 zeigt einen Deferred Choice Block in BPMN XML-Notation, der durch das Ereignis-basierte Gateway g1 (*bpmn:eventBasedGateway* Element) und das exklusive Gateway g2 (*bpmn:exclusiveGateway* Element) gebildet wird und zwei Zweige besitzt. Zweig 1 verbindet g1 über das Zwischenereignis Nachricht1 (*bpmn:intermediateCatchEvent* Element) und Task A mit g2. Verbindende Kanten sind hier e2-e4. Zweig 2 verbindet g1 über Empfang Task B (*bpmn:receiveTask* Element) mit g2 und verwendet hierfür die Kanten e5 und e6. Start- und Endzustand des Prozesses (Start1 und Ende1) sind aus Platzgründen in der Abbildung nicht angeführt. Sie sind jedoch über die Kanten e1 und e7 mit dem Deferred Choice Block verbunden.

Transformationsphase

Neben den üblichen Initialisierungsmaßnahmen werden für Empfang und Sende Tasks *bat:userPerformedPlan* Elemente in der Zwischendarstellung erzeugt, die ein *bat:eventTriggered* Kindelement enthalten.




```
        <bat:plans>
          <bat:userPerformedPlan id="A" name="A"/>
          <bat:userPerformedPlan id="B" name="B">
             <bat:eventTriggered eventID=""/>
          </bat:userPerformedPlan>
        </bat:plans>
        <bat:workingcopy>...
1         <bpmn:eventBasedGateway id="g1" name="g1"
              eventGatewayType="Exclusive" instantiate="true"/>
2         <bpmn:exclusiveGateway id="g2" name="g2"/>

3         <bpmn:intermediateCatchEvent id="Nachricht1" name="Nachricht1"/>
4         <bpmn:task id="A" name="A"/>
5         <bpmn:task id="B" name="B"/>

a         <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b         <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="Nachricht1"/>
c         <bpmn:sequenceFlow id="e3" sourceRef="Nachricht1" targetRef="A"/>
d         <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="g2"/>
e         <bpmn:sequenceFlow id="e5" sourceRef="g1" targetRef="B"/>
f         <bpmn:sequenceFlow id="e6" sourceRef="B" targetRef="g2"/>
g         <bpmn:sequenceFlow id="e7" sourceRef="g2" targetRef="Ende1"/>...
        </bat:workingcopy>...
```

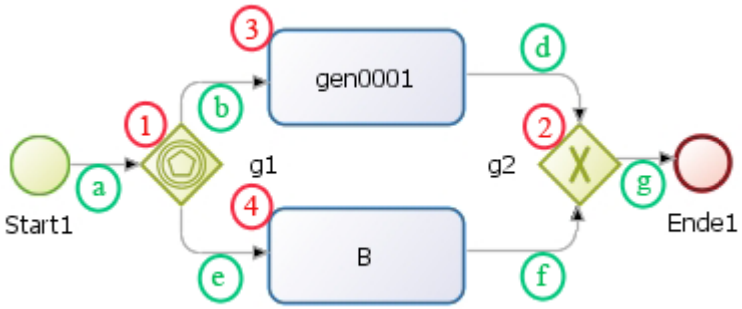
Abb. 78: der Deferred Choice Block aus Abb. 77 nach den Modifikationen der Initialisierungsphase. Änderung und neue Elemente sind fett markiert.

Wie oben beschrieben zeigt Abb. 78 die Transformationen der Initialisierung auf den Deferred Choice Block. Für die Tasks A, B wird jeweils ein *bat:userPerformedPlan* Element erzeugt, wobei B zusätzlich ein *bat:eventTriggered* Kindelement enthält, da es sich hier um einen Empfang Task handelt. In der Arbeitskopie des BPMN-Prozesses wird das Empfang Task B (*bpmn:receiveTask* Element) durch das *bpmn:task* Element ersetzt, um die Taskvarianten zu vereinheitlichen.

Ein Deferred Choice Block charakterisiert sich dadurch, dass ein verzweigendes Ereignis-basiertes Gateway über einen oder mehrere Zweig(e) mit einem synchronisierenden exklusiven Gateway verbunden ist. Das verzweigende Ereignis-basierte Gateway hat genau eine eingehende Kante und, entsprechend der Anzahl der Zweige des Deferred Choice Blocks, eine oder mehrere ausgehende Kante(n). Das synchronisierende exklusive Gateway hat, entsprechend der Anzahl der Zweige des Deferred Choice Blocks, eine oder mehrere eingehende Kante(n) und genau eine ausgehende Kante. Die einzelnen Zweige bestehen aus einem Zwischenereignis (werfend *bpmn:intermediateThrowEvent* Element oder konsumierent *bpmn:intermediateCatchEvent* Element), ein Zwischenereignis in sequentieller Abfolge mit einem Task, einem Empfangen (*bpmn:receiveTask* Element) oder Senden Task (*bpmn:sendTask* Element). Diese Komponenten haben genau eine eingehende Kante, die aus dem Ereignis-basierten Gateway entspringt, und genau eine ausgehende Kante, die im

synchronisierenden Gateway endet. Enthalten die Zweige andere Muster (z.B. Verschachtelung von Blöcken), müssen diese zuerst durch vorangegangene Reduktionsschritte der Mustertransformationsphase reduziert werden und durch einen Ersatztask ersetzt werden.

Bei der Reduktion der Sequenz eines Zwischenereignisses mit einem Task werden die beiden Knoten (Nachricht1 u. A) und die verbindende Kante durch einen Ersatztask (*bat:eventTask* Element gen0001) ersetzt und etwaige Kantenreferenzen modifiziert (siehe Abb. 79).
Als Zwischenelement wird das *bat:planActivationPlan* Element (gen0001) erzeugt, das den Task referenziert. Weiters enthält es ein *bat:eventTriggered* Element, das später die Benutzerbestätigung in der Filterbedingung aktiviert. Abb. 79 veranschaulicht den eben beschriebenen Reduktionsschritt.




```
        <bat:plans>
          <bat:planActivationPlan id="gen0001" name="ET_Sequence Nachricht1 A">
            <bat:eventTriggered eventID="Nachricht1"/>
            <bat:planRef refID="A"/>
          </bat:planActivationPlan>
          <bat:userPerformedPlan id="A" name="A"/>
          <bat:userPerformedPlan id="B" name="B">
            <bat:eventTriggered eventID=""/>
          </bat:userPerformedPlan></bat:plans>
        <bat:workingcopy>...
1         <bpmn:eventBasedGateway id="g1" name="g1"
                eventGatewayType="Exclusive" instantiate="true"/>
2         <bpmn:exclusiveGateway id="g2" name="g2"/>
3         <bat:evenTask id=" gen0001" name=" gen0001"/>
4         <bpmn:task id="B" name="B"/>
a         <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b         <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="gen0001"/>
c         <bpmn:sequenceFlow id="e4" sourceRef="gen0001" targetRef="g2"/>
d         <bpmn:sequenceFlow id="e5" sourceRef="g1" targetRef="B"/>
e         <bpmn:sequenceFlow id="e6" sourceRef="B" targetRef="g2"/>
f         <bpmn:sequenceFlow id="e7" sourceRef="g2" targetRef="Ende1"/>...
        </bat:workingcopy>...
```

Abb. 79: Reduktionschritt der Sequenz eines Zwischenergebnisses (Nachricht1) und eines Task (A) zu einem bat:eventTask Element (gen0001). Änderungen und neue Elemente sind fett hervorgehoben.

Ein Deferred Choice Block kann reduziert werden, wenn in all seinen Zweigen jeweils ein erlaubtes Element enthalten ist und dies mit der darin eingehenden Kante mit dem Ereignis-basierten Gateway am Verzweigungspunkt und mit dessen ausgehenden Kante mit dem exklusiven Gateway am Synchronisationspunkt verbunden ist. Unter erlaubten Elementen versteht man Zwischenereignisse, *bat:eventTask* Elemente, Empfang oder Sende Tasks.

Der Reduktionsschritt erfolgt ähnlich wie beim parallelen Block in Abschnitt 4.4.1.2. Die Gateways werden durch einen Ersatztask (gen0002) ersetzt und Kantenreferenzen der in den Block eingehenden und ausgehenden Kante modifiziert (siehe Abb. 80).
Als Element der Zwischendarstellung wird ein *bat:eventBlock* Element erzeugt, das die einzelnen Zweige referenziert (siehe Abb. 80, gen0002).


```
        <bat:plans>
          <bat:eventBlock id="gen0002" name="eventBlock gen0001 B"
              eventGatewayType="Exclusive">
           <bat:planRef refID="gen0001"/>
           <bat:planRef refID="B"/>
          </bat:eventBlock>
          <bat:planActivationPlan id="gen0001" name="ET_Sequence Nachricht1 A">
            <bat:eventTriggered eventID="Nachricht1"/>
            <bat:planRef refID="A"/>
          </bat:planActivationPlan>
          <bat:userPerformedPlan id="A" name="A"/>
          <bat:userPerformedPlan id="B" name="B">
            <bat:eventTriggered eventID=""/>
          </bat:userPerformedPlan>
        </bat:plans>
        <bat:workingcopy>...
1         <model:task id="gen0002"name="eventBlock Nachricht1 B"/>
a         <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0002"/>
b         <bpmn:sequenceFlow id="e7" sourceRef="gen0002" targetRef="Ende1"/>...
</bat:workingcopy>...
```

Abb. 80: zeigt den Deferred Choice Block aus Abb. 77 nach den beiden Reduktionsschritten der Mustertransformations-phase. Änderungen und neue Elemente sind fett hervorgehoben.

Asbru-Erzeugungsphase

Für die Übersetzung des ***bat:eventBlock*** Elements gibt es 2 Varianten, nämlich die exklusive und die parallele Variante.

- Bei der exklusiven (sequenziellen) Variante (Abb. 81, Plan *gen0002*) soll nur ein einzelner Zweig ausgeführt werden können. Deshalb wird das *bat:eventBlock* Element in einen Plan transformiert, dessen ***plan-body*** Element ein *subplans* Element enthält, dessen *type* Attribut den Wert „*any-order*“ hat. Dadurch kann nur ein Zweig gleichzeitig aktiviert werden. Weiters muss das *subplans* Element ein *wait-for* Element ein *one* Kindelement aufweisen. Dadurch reicht die erfolgreiche Beendigung eines einzelnen Subplans aus, um den Plan erfolgreich zu beenden. Die Subpläne und somit die einzelnen Zweige werden über die *plan-activation* und *plan-schema* Elemente referenziert.
- In der parallelen Variante (Abb. 82, Plan *gen0002*) wird ein Plan erzeugt, dessen *plan-body* Element ein *subplans* Element enthält. Das *subplans* Element hat ein ***type*** Attribut mit dem Wert „*parallel*“ und ein *wait-for-optional-subplans* Attribut mit dem Wert „*yes*“. Die restlichen Kindelemente unterscheiden sich nicht im Bezug zur exklusiven Übersetzungsvariante.

```
<plan name="gen0002" title="eventBlock gen0001 B">
  <plan-body>
    <subplans type="any-order">
      <wait-for><one/></wait-for>
      <plan-activation><plan-schema name="gen0001"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="gen0001" title="ET_Sequence Nachricht1 A">
  <conditions>
    <filter-precondition confirmation-required="yes">
      <comment text="1=1"/>
    </filter-precondition>
  </conditions>
  <plan-body><plan-activation><plan-schema name="A"/></plan-activation></plan-body>
</plan>
<plan name="A" title="A"><plan-body><user-performed/></plan-body></plan>
<plan name="B" title="B">
  <conditions>
    <filter-precondition confirmation-required="yes">
      <comment text="1=1"/>
    </filter-precondition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan>
```

Abb. 81: Auszug der Asbru-Übersetzung des sequentiellen Deferred Choice Blocks aus Abb. 77

```
<plan name="gen0002" title="eventBlock gen0001 B">
  <plan-body>
    <subplans type="parallel" wait-for-optional-subplans="yes">
      <wait-for><one/></wait-for>
      <plan-activation><plan-schema name="gen0001"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
```

Abb. 82: Asbru-Übersetzung des bat:EventBlock Elements analog zu gen003 aus Abb. 80 in der parallelen Variante. Unterschiede zur exklusiven Variante wurden fett hervorgehoben.

Das *bat:planActivationPlan* Element wird in einen Plan (Abb. 81, Plan *gen0001*) transformiert, dessen *plan-body* Element ein *plan-activation* Element enthält, welches mit dem *plan-schema* Element den referenzierten Plan aktiviert. In Abb. 81 ist es Plan A. Um die Aktivierung des Plans durch die Benutzerbestätigung zu aktivieren, muss das *confirmation-required* Attribut im *filter-precondition* Element mit dem Wert „*yes*“ belegt sein. Da die Aktivierung des Plans nur von der Benutzerbestätigung abhängen soll, wird für den Filterbedingungsausdruck (1=1 im *comment* Element) eine Tautologie angegeben.
Die Übersetzung für das *bat:userPerformedPlan* Element mit und ohne *bat:eventTriggered* Element kann Abb. 81 entnommen werden und folgt dem Mapping in Abschnitt 4.2.1.

4.4.4.2 Interleaved (Parallel) Routing (WCP17 bzw. WCP40)

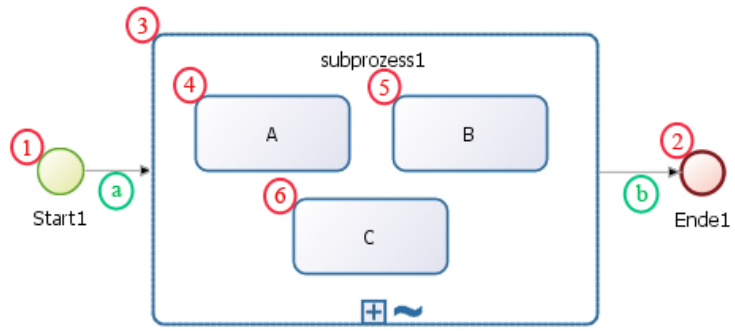
Das Interleaved Parallel Routing Pattern (WCP17) wird dadurch charakterisiert, dass eine Menge von Aktivitäten in beliebiger Reihenfolge ausgeführt werden, jedoch jede Aktivität genau einmal aktiviert werden muss, und die Ausführung strikt sequentiell erfolgt. Als zusätzliche Anforderung wird eine partielle Ordnung zwischen Tasks definiert, die bei der Ausführung eingehalten werden muss (siehe auch Abschnitt 4.1.3.2).
Das Interleaved Routing (WCP40) stellt eine vereinfachte Form des Interleaved Parallel Routing Pattern dar, bei dem die partielle Ordnung zwischen den Tasks entfällt. D.h. es wird eine Menge an Aktivitäten in beliebiger Reihenfolge ausgeführt, jedoch jede Aktivität genau einmal und die Ausführung erfolgt strikt sequentiell.

BPMN unterstützt das Interleaved Parallel Routing Pattern nicht, da die partielle Ordnung von Sequenzen und Gruppen von Aktivitäten, nicht wie in WCP 17 definiert, unterstützt wird.
BPMN kann jedoch das *Interleaved Routing* Pattern mit einem Ad-Hoc Subprozess (*bpmn:adHocSubProcess* Element) indirekt darstellen (siehe Abb. 83 und auch Definition 14 S.75). Die Menge an Aktivitäten, die in beliebiger Reihenfolge ausgeführt werden sollen, werden als Kindelemente des *bpmn:adHocSubProcess* Elements spezifiziert. Die strikt sequentielle Ausführung der Aktivitäten wird durch das *ordering* Attribut mit dem Wert „Sequential“ definiert. Weiters wird eine Beendigungsbedingung (*bpmn:completionCondition* Element) angegeben, die nach jeder erfolgreichen Beendigung einer Subaktivität überprüft wird, und die Abarbeitung des Ad-Hoc Subprozesses beendet, falls sie erfüllt ist. Unklar ist jedoch wie die Beendigungsbedingung in BPMN formuliert werden soll, damit alle Subaktivitäten genau einmal ausgeführt werden. Dies ist notwendig, da, lt. BPMN-Spezifikation, jede Subaktivität eines Ad-Hoc Subprozesses beliebig oft ausgeführt werden kann und sich standardmäßig nicht konform zu WCP 40 verhält.

Asbru kann das Interleaved Parallel Routing Pattern als auch das Interleaved Routing Pattern umsetzen. Es kann ein Plan erstellt werden, der Subpläne referenziert und eine sequentielle Abarbeitung zulässt. Die Subpläne stellen die Ad-Hoc Aktivitäten dar und werden erst nach einer Benutzerbestätigung bzw. durch Filterbedingungen aktiviert. Die partielle Ordnung wird über Filterbedingungen modelliert, die die Aktivierung eines Subplans von der Beendigung eines bestimmten anderen Subplans abhängig macht.

Da mit BPMN nur das Interleaved Routing Pattern (WCP40) modelliert werden kann, kann das Transformationssystem auch nur dieses in ein Asbru-Modell überführen.

Abb. 83 zeigt einen Ad-Hoc Subprozess subprozess1 (*bpmn:adHocSubProcess* Element), mit den Subtasks A, B und C (*bpmn:task* Element) als Kindelemente. Das *ordering* Attribut mit dem Wert „*Sequential*“ besagt, dass die Subtasks sequentiell ausgeführt werden müssen. Die Kanten e1 bzw. e2 (*bpmn:sequenceFlow* Element) führen in den Ad-Hoc Subprozess hinein bzw. aus ihm heraus. Partielle Ordnung unter den Subtasks ist hier nicht vorhanden, da keine Subtasks mit Kanten verbunden sind. Für das Transformationssystem wird folgende Konvention definiert, da dies in BPMN nicht eindeutig spezifiziert wird: Weist eine Beendigungsbedingung (*bpmn:completionCondition* Kindelement) in einem Ad-Hoc Subprozess den Wert „All“ auf, dann muss jeder Subtask genau einmal ausgeführt werden.



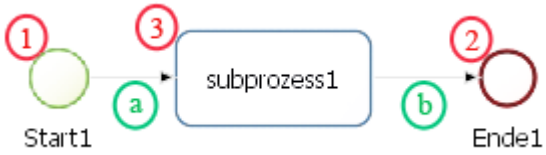

```
1       <bpmn:startEvent id="Start1" name="Start1"/>
2       <bpmn:endEvent id="Ende1" name="Ende1"/>

        <bpmn:adHocSubProcess id="subprozess1"      name="subprozess1"
                  triggeredByEvent="false" ordering="Sequential"
                   cancelRemainingInstances="true">
3         <bpmn:completionCondition>All</bpmn:completionCondition >
          4  <bpmn:task id="A" name="A"/>
          5  <bpmn:task id="B" name="B"/>
          6  <bpmn:task id="C" name="C"/>
        </bpmn:adHocSubProcess>

a       <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="subprozess1"/>
b       <bpmn:sequenceFlow id="e2" sourceRef="subprozess1" targetRef="Ende1"/>
```

Abb. 83: Ad-Hoc Subprozess in BPMN XML-Notation. Die Tasks A, B, C sollen in beliebiger Abfolge ausgeführt werden.

Transformationsphase

Nach den üblichen Initialisierungs- und Vereinheitlichungsmaßnahmen kann ein Ad-Hoc Subprozess durch das BPMN-zu-Asbru Transformationssystem reduziert werden, wenn ein Ad-Hoc Subprozess (*bpmn:adHocSubProcess* Element) nur *bpmn:task* Kindelemente enthält, die nicht durch Kanten verbunden sind, und der zusätzlich ein *bpmn:completionCondition* Element mit dem Wert „All“ aufweist.
Beim Reduktionsschritt wird das *bpmn:adHocSubProcess* Element mit allen Kindelementen durch einen Ersatztask (*bpmn:task* Element) ersetzt.
Als Element der Zwischendarstellung wird das *bat:adhocBlock* Element eingefügt, das mit seinen *bat:planRef* Elementen die Ad-Hoc Subpläne referenziert.
In die für die Ad-Hoc Subpläne in der Initialisierungsphase (bzw. durch einen vorherigen Reduktionsschritt) erzeugten Elemente werden durch die Mustertransformationsphase zusätzlich noch *bat:userTriggered* Kindelemente eingefügt, um später die Benutzerbestätigung in der Asbru Filterbedingung zu aktivieren.

In Abb. 84 wird der o.a. Reduktionsschritt veranschaulicht. Der Ad-Hoc Subprozess subprozess1 (*bpmn:adHocSubProcess* Element) und alle seine Kindelemente Tasks A, B, C (*bpmn:task* Elemente) werden in der Arbeitskopie entfernt und durch einen Ersatztask *subprozess1* ersetzt.
Weiters wird das ***bat:adhocBlock*** Element erzeugt, das mit den ***bat:planRef*** Elementen die *bat:userPerformedPlan* Elemente A, B, C als Subpläne referenziert.




```
        <bat:plans>
          <bat:adhocBlock id="subprozess1" name="adhocBlock A B C"
                          ordering="Sequential" cancelRemainingInstances="true">
            <bat:planRef refID="A"/>
            <bat:planRef refID="B"/>
            <bat:planRef refID="C"/>
          </bat:adhocBlock>
          <bat:userPerformedPlan id="A" name="A"/>
          <bat:userPerformedPlan id="B" name="B"/>
          <bat:userPerformedPlan id="C" name="C"/>
        </bat:plans>
        <bat:workingcopy>
1         <bpmn:startEvent id="Start1" name="Start1"/>
2         <bpmn:endEvent id="Ende1" name="Ende1"/>
3         <bpmn:task id="subprozess1" name="subprozess1"/>
a         <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="subprozess1"/>
b         <bpmn:sequenceFlow id="e2" sourceRef="subprozess1" targetRef="Ende1"/>
        </bat:workingcopy>
```

Abb. 84: Ad-Hoc Subprozess aus Abb. 83 nach den Modifikationen der Mustertransformationsphase. Änderungen und neue Elemente wurden fett hervorgehoben.

Asbru-Erzeugungsphase

Bei der Transformation des ***bat:adhocBlock*** Elements in Asbru-Pläne wird ein Plan (Abb. 85, subprozess1) erzeugt, dessen ***plan****-body* Element ein *subplans* Element enthält, dessen *type* Attribut den Wert „*any-order*“ hat. Weiters muss das *subplans* Element ein *wait-for* Element aufweisen mit einem *all* Kindelement. Dadurch müssen alle Subpläne erfolgreich beendet werden, damit der Plan erfolgreich beendet wird. Die Subpläne werden über die *plan-activation* und ***plan****-schema* Elemente referenziert.

Die Übersetzung für das ***bat:userPerformedPlan*** Element (A, B, C) mit *bat:userTriggered* Kindelement kann Abb. 85 (Plan A, B, C) und Abb. 81 entnommen werden.

```
<plan name="subprozess1" title="adhocBlock A B C">
  <plan-body>
    <subplans type="any-order" >
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="B"/></plan-activation>
      <plan-activation><plan-schema name="C"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A" title="A">
  <conditions>
    <filter-precondition confirmation-required="yes">
      <comment text="1=1"/>
    </filter-precondition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan><plan name="B" title="B">
  <conditions>
    <filter-precondition confirmation-required="yes">
      <comment text="1=1"/>
    </filter-precondition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan><plan name="C" title="C">
  <conditions>
    <filter-precondition confirmation-required="yes">
      <comment text="1=1"/>
    </filter-precondition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan>
```

Abb. 85: Auszug der Asbru-Übersetzung des Ad-Hoc Subprozesses aus Abb. 83.

4.4.5 Abbruch Kontrollflußmuster

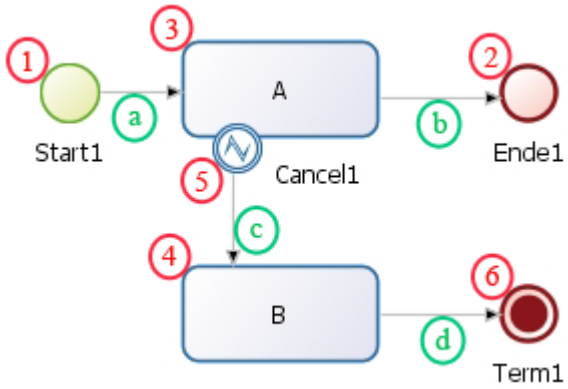
Die Kategorie der Abbruch Kontrollflussmuster enthalten das Cancel Task (WCP19) und Cancel Case Pattern (WCP20), deren Umsetzung in den folgenden Unterabschnitten erläutert wird.

4.4.5.1 Cancel Task (WCP19)

Mit dem Cancel Task Muster kann abgebildet werden, dass eine Aktivität, die zuvor gestartet wurde, abgebrochen wird und danach der normale Prozessablauf bzw. ein etwaiger alternativer Prozessablauf ausgeführt wird (siehe auch Abschnitt 4.1.6.1).

BPMN kann dieses Muster mit Hilfe eines anhaftenden, konsumierenden Fehler-Zwischenereignisses realisieren, das an einen Task angehaftet wird (siehe auch Definition 14 S.75). Dem anhaftenden Fehler-Zwischenereignis kann auch eine Kante entspringen, über die der alternative Prozessablauf aktiviert wird, der vom normalen Kontrollfluss abweicht. Wird jedoch an einem Task nur ein Fehler-Zwischenereignis angehaftet, ohne einen Alternativzweig zu definieren, dann wird im Fehlerfall die Aktivität abgebrochen und das Kontrollflusstoken an die im normalen Kontrollfluss nachfolgenden

Knoten weitergegeben. In dieser Arbeit werden nur alternative Prozessabläufe betrachtet, die in einem Terminierungsendereignis enden, da Asbru ein Einmünden des alternativen in den regulären Prozessablauf nicht abbilden kann.
In BPMN wird ein anhaftendes Zwischenereignis als *bpmn:boundaryEvent* Element dargestellt, das an jenen Knoten anhaftet, auf den das *attachedToRef* Attribut verweist (siehe Abb. 86). Durch das *bpmn:errorEventDefinition* Kindelement wird der Typ des Zwischenereignisses definiert, nämlich als ein Fehler-Zwischenereignis. Um einen Alternativzweig zu definieren, können Kanten (bpmn:sequenceFlow Elemente) aus dem Fehler-Zwischenereignis führen.
Der alternative Zweig muss in dieser Arbeit in einem Terminierungsendereignis (*bpmn:endEvent* Elment mit *bpmn:terminateEventDefinition* Kindelement) enden.




```
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:endEvent id="Ende1" name="Ende1"/>
3   <bpmn:task id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
    <bpmn:boundaryEvent id="Cancel1" name="Cancel1" attachedToRef="A">
5     <bpmn:errorEventDefinition id="eventdef-Cancel1" .../>
    </bpmn:boundaryEvent>
    <bpmn:endEvent id="Term1" name="Term1">
6     <bpmn:terminateEventDefinition id="_e8Doo"/>
    </bpmn:endEvent>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="A"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="A" targetRef="Ende1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Cancel1" targetRef="B"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="B" targetRef="Term1"/>
```

Abb. 86: Task A mit anhaftendem Fehler-Zwischenereignis Cancel1, dessen Alternativzweig über Task B in Terminierung Endereignis Term1 endet. Prozess in BPMN XML-Notation.

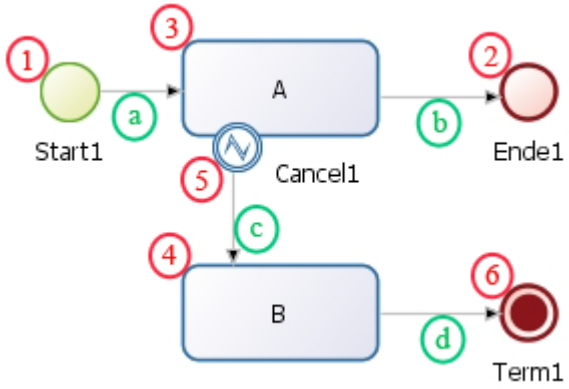
Abb. 86 zeigt einen BPMN-Prozess mit einem Cancel Task Muster. Nachdem der Prozess gestartet (*bpmn:startEvent* Element Start1) wird, wird Task A (*bpmn:task* Element) aktiviert, nach dessen erfolgreicher Beendigung wird der Prozess beendet (*bpmn:endEvent* Element Ende1). Das Fehler-Zwischenereignis Cancel1 (*bpmn:boundaryEvent* Element) haftet an Task A, indem das *attachedToRef* Attribut darauf verweist.
Weiters ist ein alternativer Zweig definiert, der Task B aktiviert und nach dessen Beendigung den Prozess via Terminierungsendereignis Term1 (*bpmn:endEvent* Element mit *bpmn:terminateEventDefinition* Kindelement) beendet, wenn in Task A ein Fehlerereignis geworfen wird.

Asbru kann das Cancel Task Pattern durch eine Abbruchbedingung in einem Plan umsetzen. D.h. dem Plan der die Aktivität mit dem anhaftenden Fehler-Zwischenereignis repräsentiert, wird eine

Abbruchbedingung (abort-condition Element) hinzugefügt, die erfüllt ist, wenn ein boolscher Parameter true wird. Ist sie erfüllt, wird der Plan abgebrochen, und es kann ein etwaiger alternativer Prozessablauf ausgeführt werden. Mehr Details dazu werden in der Beschreibung der Asbru-Erzeugungsphase gezeigt.

Transformationsphase

In der Arbeitskopie werden bei der Initialisierung Tasks, denen ein Fehler-Zwischenereignis anhaftet, durch ein *bat:errorEventTask* Element ersetzt, das die Attribute und Kindelemente des Tasks unverändert übernimmt.
Weiters wird für jedes anhaftende Fehler-Zwischenereignis (*bpmn:boundaryEvent/ bpmn:errorEventDefinition* Element) ein Abbruchparameter (bat:abortParameter Element) in der Zwischendarstellung erzeugt. In der Arbeitskopie referenzieren Fehler-Zwischenereignisse durch das *abortedByParameter* Attribut den zugehörigen Parameter.
Für jeden Prozess bzw. Subprozess (*bpmn:process* bzw. *bpmn:subProcess* Element), der ein Terminierungsendereignis enthält, wird eine Abbruchvariable (*bat:abortVariable* Element) erzeugt. In der Arbeitskopie referenzieren (Sub)Prozesse durch das *abortedByVariable* Attribut die zugehörige Abbruchvariable. Die Abbruchvariable wird erzeugt, um das Terminierungsendereignis zu simulieren, und wird nur indirekt bei der Umsetzung des Cancel Task Musters benötigt.
In Abb. 87 werden die oben beschriebenen Modifikationen an dem Beispielprozess veranschaulicht.




```
<bat:variables><bat:abortVariable id="abortvar_Prozess1"/></bat:variables>
<bat:parameters><bat:abortParameter id="abortpara_Cancel1"/></bat:parameters>
<bat:plans>
  <bat:userPerformedPlan id="B" name="B"/>
<bat:plans>
<bat:workingcopy>
  <bpmn:process id="Prozess1" name="Prozess1"
        abortedByVariable="abortvar_Prozess1">
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:endEvent id="Ende1" name="Ende1"/>
3   <bat:errorEventTask userTriggered="false"  id="A" name="A"/>
4   <bpmn:task id="B" name="B"/>
    <bpmn:boundaryEvent id="Cancel1" name="Cancel1"
5         attachedToRef="A"  abortedByParameter="abortpara_Cancel1">
      <bpmn:errorEventDefinition id="eventdef-Cancel1" errorRef="Cancel1FC"/>
    </bpmn:boundaryEvent>
    <bpmn:endEvent id="Term1" name="Term1">
6     <bpmn:terminateEventDefinition id="_e8Doo"/>
    </bpmn:endEvent>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="A"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="A" targetRef="Ende1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Cancel1" targetRef="B"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="B" targetRef="Term1"/>
  </bpmn:process>
<bat:workingcopy>...
```

Abb. 87: BPMN-Prozess mit Cancel Task Muster aus Abb. 86 nach den Initialisierungsmaßnahmen. Äderungen und neue Elemente wurden fett hervorgehoben.

Das Cancel Task Muster wird nun durch ein vorhandenes *bat:errorEventTask* Element in der Arbeitskopie identifiziert. Es wird reduziert, indem das *bat:errorEventTask* Element und das anhaftende Fehlerereignis durch einen Ersatztask ersetzt werden. Ist ein alternativer Zweig vorhanden, müssen auch dessen Elemente aus der Arbeitskopie entfernt werden.

Bei der Reduktion werden mehrere Zwischenelemente erzeugt (siehe auch Abb. 88). Ist kein alternativer Zweig (Basisvariante) definiert, werden folgende Elemente der Zwischendarstellung erzeugt:

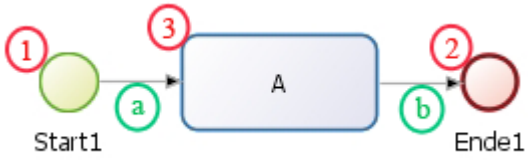
- *bat:userPerformedPlan* Element: Es repräsentiert jene Aktivität, an der das Fehler-Zwischenereignis anhaftet. Es enthält ein *bat:abortConditionParameter* Kindelement und verweist auf den korrespondierenden Abbruchparameter des *bat:errorEventTask* Elements in

der Arbeitskopie. Später wird daraus eine Asbru Abbruchbedingung generiert, die den Plan abbricht, falls der Abbruchparameter den Wert „true“ annimmt.

- *bat:planActivationPlan* Element: Es wird verwendet, um auf den normalen und den optionalen alternativen Zweig zu verweisen. Mit dem *bat:planRef* Kindelement wird jenes Element referenziert, das den normalen Zweig repräsentiert, und mit dem *bat:onabortRef* Kindelement den alternative Zweig. Ist kein alternativer Zweig definiert, entfällt das *bat:onabortRef* Element.

Ist ein alternativer Zweig vorhanden, sind zusätzliche Elemente notwendig. Ist eine Aktivität im Alternativzweig des Cancel Task Musters definiert, dann referenziert das *bat:onabortRef* Element des *bat:planActivationPlan* Elements auf das *bat:sequentialPlan* Element. Ist keine Aktivität im Alternativzweig definiert, dann referenziert das *bat:onabortRef* Element das *bat:endEventPlan* Element.

Abb. 88 zeigt die oben beschriebenen Änderungen bei der Mustertransformation am BPMN-Prozess. Aus der Arbeitskopie (*bat:workingcopy* Element) wurden das *bat:errorEventTask* Element A, das *bpmn:boundaryEvent* Element Cancel1, die Alternativaktivität Task B (*bpmn:task* Element), das Terminierungsendereignis Term1 (*bpmn:endEvent* Element) und die Kanten des alternativen Zweigs e3, e4 entfernt, und statt ihrer ein abstrakter Ersatztask A (*bpmn:task* Element mit selber ID wie das *bat:errorEventTask* Element) eingefügt.




```
<bat:variables><bat:abortVariable id="abortvar_Prozess1"/></bat:variables>
<bat:parameters><bat:abortParameter id="abortpara_Cancel1"/></bat:parameters>
<bat:plans>
  <bat:planActivationPlan id="A" name="A">
    <bat:planRef refID="A_normalBranch"/>
    <bat:onabortRef refID="A_alternativeBranch"/>
  </bat:planActivationPlan>
  <bat:userPerformedPlan id="A_normalBranch" name="A_normalBranch">
    <bat:abortConditionParameter>abortpara_Cancel1</bat:abortConditionParameter>
  </bat:userPerformedPlan>
  <bat:sequentialPlan id="A_alternativeBranch" name="A_alternativeBranch">
    <bat:sourceRef>B</bat:sourceRef>
    <bat:targetRef>Term1</bat:targetRef>
  </bat:sequentialPlan>
  <bat:endEventPlan id="Term1" name="Term1">
    <bat:terminationVariable variableRef="abortvar_Prozess1"/>
  </bat:endEventPlan>
  <bat:userPerformedPlan id="B" name="B"/>
<bat:plans>
<bat:workingcopy>
  <bpmn:process id="Prozess1" name="Prozess1"
        abortedByVariable="abortvar_Prozess1">
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:endEvent id="Ende1" name="Ende1"/>
3   <bpmn:task id="A" name="A"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="A"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="A" targetRef="Ende1"/>
  </bpmn:process>
<bat:workingcopy>...
```

Abb. 88: BPMN-Prozess mit Cancel Task Muster aus Abb. 86 nach den Modifikationen der Mustertransformationsphase. Äderungen und neue Elemente wurden fett hervorgehoben.

Asbru-Erzeugungsphase

In der Asbru-Erzeugungsphase werden aus den Elementen der Zwischendarstellung (*bat:planActivationPlan, bat:userPerformedPlan, bat:sequentialPlan* und *bat:endEventPlan* Element) Asbru-Pläne erstellt.

```
<plan name="A" title="A">
  <plan-body>
    <plan-activation><plan-schema name="A_normalBranch"/>
      <on-abort>
        <plan-activation><plan-schema name="A_alternativeBranch"/></plan-activation>
      </on-abort>
    </plan-activation>
  </plan-body>
</plan>
<plan name="A_normalBranch" title="A_normalBranch">
  <conditions>
    <abort-condition>
      <parameter-proposition parameter-name="abortpara_Cancel1">
        <value-description type="equal"><qualitative-constant value="true"/>
        </value-description>...
      </parameter-proposition>
    </abort-condition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan>
<plan name="A_alternativeBranch" title="A_alternativeBranch">
  <plan-body><subplans type="sequentially">
    <wait-for><all/></wait-for>
    <plan-activation><plan-schema name="B"/></plan-activation>
    <plan-activation><plan-schema name="Term1"/></plan-activation>
  </subplans></plan-body>
</plan>
<plan name="Term1" title="Term1">
  <plan-body>
    <!-- bricht übergeordneten Prozess (Prozess1) ab -->
    <variable-assignment variable="abortvar_Prozess1">
      <qualitative-constant value="true"/>
    </variable-assignment>
  </plan-body>
</plan>
<plan name="B" title="B"><plan-body><user-performed/></plan-body></plan>
```

Abb. 89: Asbru-Übersetzung der Pläne des Cancel Task Musters aus Abb. 86

Für das *bat:planActivationPlan* Element wird ein Plan (Abb. 89 Plan A) generiert, das mit dem *plan-body/plan-activation* Element den normalen Zweig (Plan A_normalBranch) aktiviert. Wird dieser abgebrochen, d.h. die Abbruchbedingung ist erfüllt, dann wird der alternative Zweig (Plan A_alternativeBranch) aktiviert, der durch das *on-abort/plan-activation* Element referenziert wird.

Das *bat:userPerformedPlan* Element wird zu einem Plan (Abb. 89 Plan A_normalBranch, B) übersetzt, das ein *plan-body/user-performed* Element enthält. Im Plan A_normalBranch wird auch noch eine Abbruchbedingung (*conditions/abort-condition* Element) definiert, die erfüllt ist, wenn Parameter abortpara_Cancel1 (*parameter-proposition* Element) den Wert true (*qualitative-constant* Element) annimmt.
Aus dem *bat:sequentialPlan* Element (A_alternativeBranch) wird ein Asbru-Plan A_alternativeBranch (Abb. 89) erzeugt, der die alternative Aktivität (Plan B) und anschließend das Terminierungsendereignis (Plan Term1) sequentiell hintereinander ausführt (*subplans* Element, Attribut type="sequentially").

Das *bat:endEventPlan* Element wird zu einem Asbru-Plan Term1 übersetzt, in dem der Variablen abortvar_Prozess1 (*variable-assignment* Element) der Wert true (*qualitative-constant* Element) zugewiesen wird. Dadurch wird die Abbruchbedingung des übergeordneten Prozesses Prozess1, der den gesamten Prozessablauf umschließt, erfüllt und beendet.

```
<domain-defs><domain name="mainDomain">
...
<parameter-def name="abortpara_Cancel1" required="no" type="Boolean">
  <raw-data-def mode="manual" use-as-context="no"/>
</parameter-def>
...
</domain></domain-defs>
<library-defs>
  <qualitative-scale-def name="Boolean">
    <qualitative-entry entry="false"/>
    <qualitative-entry entry="true"/>
  </qualitative-scale-def>
  <variable-def name="abortvar_Prozess1">
    <scalar-def type="Boolean">
      <initial-value>
        <qualitative-constant value="false"/>
      </initial-value>
    </scalar-def>
  </variable-def>
</library-defs>
```

Abb. 90: Asbru-Übersetzung der Parameter- und Variablendefinition des Cancel Task Musters aus Abb. 86

Bei der Transformation des Cancel Task Musters müssen noch Parameter- und Variablendefinitionen (Abb. 90) erstellt werden, damit die oben beschriebenen Parameter und Variablen anwendbar sind.

4.4.5.2 Cancel Case (WCP20)

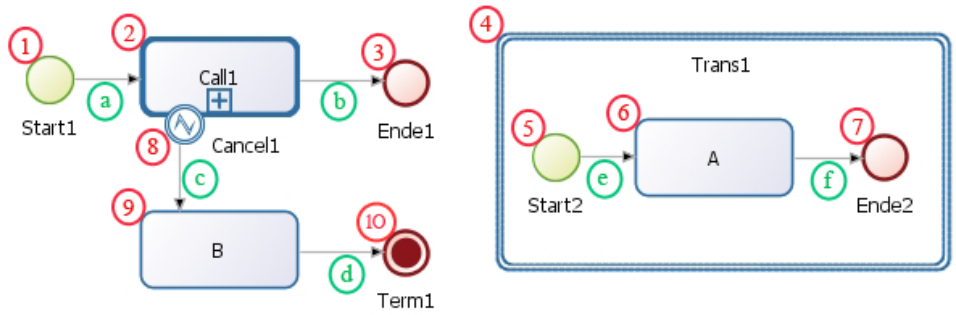
Das Cancel Case Muster bildet ab, dass eine Subprozess, der zuvor gestartet wurde, abgebrochen wird, und danach eventuell im aufrufenden Prozess ein alternativer Prozessablauf durchlaufen wird (siehe auch Abschnitt 4.1.6.2).

BPMN kann dieses Muster mit Hilfe eines anhaftenden, konsumierenden Fehler-Zwischenereignisses realisieren, das an eine Transaktion (*bpmn:transaction* Element) oder eine Call-Aktivität angehaftet wird (siehe auch Definition 14 S.75). Dem anhaftenden Fehler-Zwischenereignis kann auch eine Kante entspringen, über die der alternative Prozessablauf aktiviert wird, der vom normalen Kontrollfluss abweicht. In dieser Arbeit werden nur alternative Prozessabläufe betrachtet, die in einem Terminierungsendereignis enden, da Asbru ein Einmünden des alternativen in den regulären Prozessablauf nicht abbilden kann. Wird jedoch an einer Transaktion bzw. einer Call-Aktivität nur ein Fehler-Zwischenereignis angehaftet, ohne einen Alternativzweig zu definieren, dann wird im Fehlerfall der Teilprozess abgebrochen und das Kontrollflusstoken an die im normalen Kontrollfluss nachfolgenden Knoten weitergegeben.
In BPMN wird ein anhaftendes Zwischenereignis als *bpmn:boundaryEvent* Element dargestellt, das an jenen Knoten anhaftet, auf den das *attachedToRef* Attribut verweist (siehe Abb. 91). Durch das *bpmn:errorEventDefinition* Kindelement wird der Typ des Zwischenereignisses definiert, nämlich als

ein Fehler-Zwischenereignis. Um einen Alternativzweig zu definieren, können Kanten (bpmn:sequenceFlow Elemente) aus dem Fehler-Zwischenereignis führen.
Der alternative Zweig muss in dieser Arbeit in einem Terminierungsendereignis enden (*bpmn:endEvent* Elment mit *bpmn:terminateEventDefinition* Kindelement), da dieser durch ein Endereignis beendet werden muss und nicht in den regulären Prozessablauf münden darf.

Abb. 91 zeigt einen BPMN-Prozess mit dem Cancel Case Muster. Wenn der Prozess gestartet wird (*bpmn:startEvent* Element Start1), wird Call-Aktivität Call1 (*bpmn:callActivity* Element), welche die Transaktion Trans1 (*bpmn:transaction* Element) aufruft, aktiviert. Nach dessen erfolgreicher Beendigung wird der Prozess beendet (*bpmn:endEvent* Element Ende1). Das Fehler-Zwischenereignis Cancel1 (*bpmn:boundaryEvent* Element) haftet an Call-Aktivität Call1, indem das *attachedToRef* Attribut darauf verweist.
Die Transaktion Trans1 besteht aus einem Task A, der bei Aufruf des Subprozesses aktiviert wird, den Start- bzw. Endereignis Start2 bzw. Ende2 und den Kanten e5, e6, die die Knoten verbinden.
Weiters ist ein alternativer Zweig definiert, der Task B (*bpmn:task* Element) aktiviert und nach dessen Beendigung den Prozess via Terminierungsendereignis Term1 (*bpmn:endEvent* Element mit *bpmn:terminateEventDefinition* Kindelement) beendet, wenn in Task A ein Fehlerereignis geworfen wird.

Asbru kann das Cancel Case Pattern durch eine Abbruchbedingung in einem Plan umsetzen. D.h. dem Plan, der die Transaktion mit Fehler-Zwischenereignis repräsentiert, wird eine Abbruchbedingung (abort-condition Element) hinzugefügt, die erfüllt, ist wenn ein boolscher Parameter true wird. Ist sie erfüllt, wird der Plan abgebrochen, und es kann ein alternativer Zweig – falls vorhanden – ausgeführt werden. Mehr Details dazu werden in der Beschreibung der Asbru-Erzeugungsphase gezeigt.




```
<bpmn:process id="Prozess1" name="Prozess1" >
1     <bpmn:startEvent id="Start1" name="Start1"/>
2     <bpmn:callActivity id="Call1" name="Call1" calledElement="Trans1"/>
3     <bpmn:endEvent id="Ende1" name="Ende1"/>

      <bpmn:transaction id="Trans1" name="Trans1" triggeredByEvent="false"
5       <bpmn:startEvent id="Start2" name="Start2"/>
4     6   <bpmn:task id="A" name="A"/>
7       <bpmn:endEvent id="Ende2" name="Ende2"/>
      </bpmn:transaction>

      <bpmn:boundaryEvent id="Cancel1" name="Cancel1" attachedToRef="Call1">
8       <bpmn:errorEventDefinition id="eventdef-Cancel1"/>
      </bpmn:boundaryEvent>
9     <bpmn:task id="B" name="B"/>
      <bpmn:endEvent id="Term1" name="Term1">
10      <bpmn:terminateEventDefinition id="_7MU0w"/>
      </bpmn:endEvent>

a     <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call1"/>
b     <bpmn:sequenceFlow id="e2" sourceRef="Call1" targetRef="Ende1"/>
c     <bpmn:sequenceFlow id="e3" sourceRef="Cancel1" targetRef="B"/>
d     <bpmn:sequenceFlow id="e4" sourceRef="B" targetRef="Term1"/>
e     <bpmn:sequenceFlow id="e5" sourceRef="Start2" targetRef="A"/>
f     <bpmn:sequenceFlow id="e6" sourceRef="A" targetRef="Ende2"/>
</bpmn:process>
```

Abb. 91: BPMN-Prozess mit Cancel Case Muster in BPMN XML-Notation.

Transformationsphase

Neben den üblichen Initialisierungs- und Vereinheitlichungsmaßnahmen bei Tasks hat die Initialisierung hier im wesentlichen drei weitere Aufgaben, nämlich Zwischenelemente für (1) Parameter- und Variablendefinitionen zu erzeugen, (2) die Call-Aktivität durch einen Task zu ersetzen und (3) das anhaftende Fehler-Zwischenereignis von der Call-Aktivität an die Transaktion (*bpmn:transaction* Element) anzuhaften.

(1)

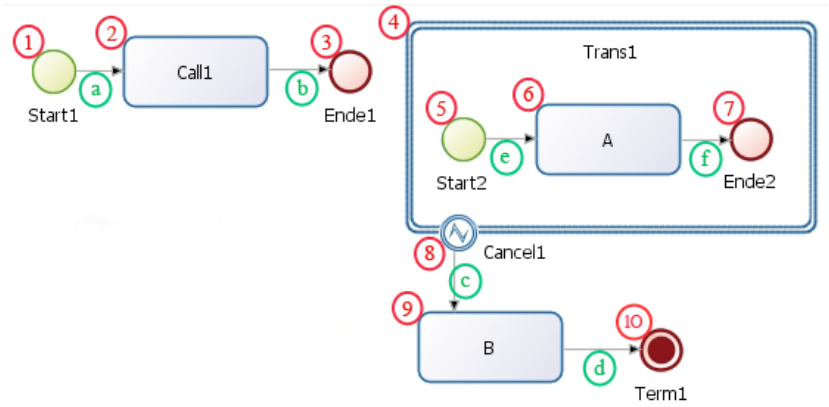
a. Abbruchparameter: Weiters wird für jedes anhaftende Fehler-Zwischenereignis (*bpmn:boundaryEvent/bpmn:errorEventDefinition* Element) ein Abbruchparameter (bat:abortParameter Element) in der Zwischendarstellung erzeugt. In der Arbeitskopie referenzieren Fehler-Zwischenereignisse durch das *abortedByParameter* Attribut den zugehörigen Parameter. Abb. 92 zeigt, dass in der

Arbeitskopie das Fehler-Zwischenereignis (Cancel1) den korrespondierenden Abbruchparameter (abortpara_Cancel1) referenziert.

b. Abbruchvariablen: Für jeden Prozess bzw. Subprozess (*bpmn:process* bzw. *bpmn:subProcess* Element), der ein Terminierungsendereignis enthält, wird eine Abbruchvariable (*bat:abortVariable* Element) erzeugt. In der Arbeitskopie referenzieren (Sub)Prozesse, durch das *abortedByVariable* Attribut, die zugehörige Abbruchvariable. Die Abbruchvariable wird erzeugt, um das Terminierungsendereignis zu simulieren und wird nur indirekt bei der Umsetzung des Cancel Task Musters benötigt. Abb. 92 zeigt, dass in der Arbeitskopie der Prozess (Prozess1) bzw. die Transaktion (Trans1) auf die korrespondierende Abbruchvariable (abortvar_Prozess1 bzw. abortvar_Trans1) referenziert.

(2) Für jede Call-Aktivität (*bpmn:callActivity* Element) wird ein *bat:planActivationPlan* Element erzeugt, das mit seinem *bat:planRef* Kindelement auf die zu aktivierende Transaktion verweist. In der Arbeitskopie wird dabei die Call-Aktivität durch einen Ersatztask ersetzt, damit umschließende Pattern reduziert werden können. Abb. 92 veranschaulicht dies an Call-Aktivität Call1.

(3) Anhaftende Fehler-Zwischenereignisse (*bpmn:boundaryEvent* Element mit *bpmn:errorEventDefinition* Kindelement), die an einer Call-Aktivität anhaften, haften nach den Initialisierungsmaßnahmen der Transaktion an, auf die die Call-Aktivität verwiesen hat. D.h., das *attachedToRef* Attribut des *bpmn:boundaryEvent* Elements verweist nun auf ein *bpmn:transaction* Element. Abb. 92 zeigt, dass das Fehler-Zwischenereigniss Cancel1 in der Arbeitskopie nun an Transaktion Trans1 anhaftet.




```
<bat:variables>
  <bat:abortVariable id="abortvar_Prozess1"/>
  <bat:abortVariable id="abortvar_Trans1"/></bat:variables>
<bat:parameters><bat:abortParameter id="abortpara_Cancel1"/></bat:parameters>
<bat:plans>
  <bat:userPerformedPlan id="A" name="A"/>
  <bat:planActivationPlan id="Call1" name="Call1">
    <bat:planRef refID="Trans1"/></bat:planActivationPlan>
  <bat:userPerformedPlan id="B" name="B"/>
<bat:plans>
<bat:workingcopy>
  <bpmn:process id="Prozess1" name="Prozess1"
                abortedByVariable="abortvar_Prozess1">
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task id="Call1" name="Call1" calledElement="Trans1"/>
3   <bpmn:endEvent id="Ende1" name="Ende1"/>

    <bpmn:transaction id="Trans1" name="Trans1" triggeredByEvent="false"
                      abortedByVariable="abortvar_Trans1">
5     <bpmn:startEvent id="Start2" name="Start2"/>
4   6 <bpmn:task id="A" name="A"/>
7     <bpmn:endEvent id="Ende2" name="Ende2"/>
    </bpmn:transaction>
    <bpmn:boundaryEvent id="Cancel1" name="Cancel1" attachedToRef="Trans1"
                        abortedByParameter="abortpara_Cancel1">
8     <bpmn:errorEventDefinition id="eventdef-Cancel1"/>
    </bpmn:boundaryEvent>
9   <bpmn:task id="B" name="B"/>
    <bpmn:endEvent id="Term1" name="Term1">
10    <bpmn:terminateEventDefinition id="_7MU0w"/>
    </bpmn:endEvent>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="Call1" targetRef="Ende1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Cancel1" targetRef="B"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="B" targetRef="Term1"/>
e   <bpmn:sequenceFlow id="e5" sourceRef="Start2" targetRef="A"/>
f   <bpmn:sequenceFlow id="e6" sourceRef="A" targetRef="Ende2"/>
  </bpmn:process><bat:workingcopy>...
```

Abb. 92: Cancel Case Muster aus Abb. 91 nach den Initialisierungsmaßnahmen. Änderungen und neue Elemente wurden fett hervorgehoben

Die Reduktion des Cancel Case Musters erfolgt in zwei Schritten. Dabei wird vorausgesetzt, dass der Prozess in der Transaktion TransX, dem das Fehler-Zwischenereignis anhaftet, durch vorherige Reduktionsschritte zu einem einzelnen Task X reduziert wurde. Im Transaktionsfaltungsschritt wird die Transaktion TransX aus der Arbeitskopie entfernt, und das ihm anhaftende Fehler-Zwischenereignis wird an Task X angehaftet. Dies entspricht somit einem Sonderfall des Cancel Task Musters.
Somit wird inm Errortask-Reduktionsschritt ähnlich wie beim Cancel Task Muster (vergleiche Abschnitt 4.4.5.1) vorgegangen. D.h., es werden die am Muster beteiligten Elemente entfernt und Zwischenelemente erzeugt, die bei Abbruch der Transaktion einen etwaigen alternativen Prozessablauf aktivieren.
Die beiden Reduktionsschritte werden in folgenden Unterabschnitten detaillierter beschrieben.

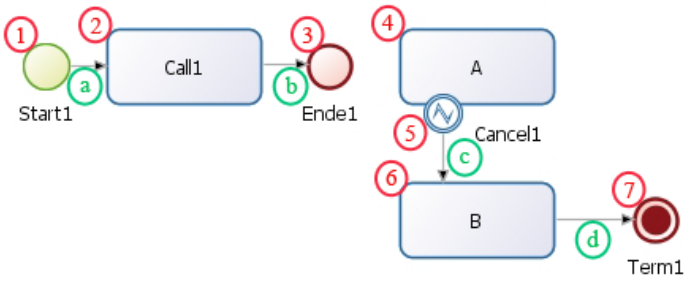
Transaktionsfaltungsschritt

Das Cancel Case Muster wird dadurch identifiziert bzw. der Transaktionsfaltungsschritt ausgeführt, indem einer Transaktion ein Fehler-Zwischenereignis anhaftet, der Teilprozess in der Transaktion aus einem einzelnen Task besteht, dessen Vorgängerknoten ein Startereignis und dessen Nachfolgeknoten ein Endereignis ist (siehe Abb. 92 4-8).
Zuerst wird in der Arbeitskopie die Transaktion TransX zusammengefaltet, d.h. das Transaktions Element und alle darin befindlichen Kindelemente außer Task X werden entfernt. Eine Transaktion kann nur zusammengefaltet werden, wenn ein Task (X), ein Start- und ein Endeereignis enthalten sind, und zwei Kanten die drei Knoten verbinden.
Anschließend wird Task X durch ein *bat:errorEventTask* Element ersetzt, das, zusätzlich zu den im Task vorhandenen Attributen, das *plansid* und *userTriggered* Attribut einfügt. Das *plansid* Attribut wird verwendet, wenn das *id* Attribut des Zwischenelements von der ursprünglichen ID des BPMN-Elements abweichen soll. Hier hält es die Referenz auf die eben entfernte Transaktion. Das *userTriggered* Attribut hat den Wert „true“, wenn das *bat:errorEventTask* Element ein *bpmn:sendTask* oder *bpmn:receiveTask* Element ersetzt, da es später die Benutzerbestätigung in der Filterbedingung des Asbru-Plans aktiviert. Sonst hat das *userTriggered* Attribut den Wert „false“.

Weiters wird in der Arbeitskopie das Fehler-Zwischenereignis das TransX anhaftete, durch Ändern des *attachedToRef* Attributs, an Task X (also jetzt *bat:errorEventTask* Element X) angehaftet.

Da Task X die Transaktion repräsentiert und ihm das Fehler-Zwischenereignis anhaftet, muss im korrespondierenden Zwischenelement (ist schon vorhanden) ein *bat:abortConditionParameter* Element eingefügt werden, welches auf den entsprechenden Abbruchparameter verweist. Der referenzierte Abbruchparameter korrespondiert mit dem des anhaftenden Fehler-Zwischenelements. Später wird daraus eine Abbruchbedingung generiert, die erfüllt ist, wenn der referenzierte boolsche Abbruchparameter den Wert true aufweist.

Abb. 93 veranschaulicht auszugsweise den Transaktionsfaltungsschritt an dem BPMN-Prozess aus Abb. 91. Das *bat:userPerformedPlan* Element A erhält ein *bat:abortConditionParameter* Kindelement und verweist auf den Abbruchparameter abortpara_Cancel1.
In der Arbeitskopie (*bat:workingcopy* Element) wird bpmn:task Element A durch *bat:errorEventTask* Element ersetzt, das zusätzlich das *plansid* Attribut, das auf die Transaktion Trans1 verweist, und das *userTriggered* Attribut (false, da A ein bpmn:task Element war) erhalten hat.
Weiters haftet Fehler-Zwischenelement Cancel1 (bpmn:boundaryEvent) nicht mehr an Trans1 sondern an A (attachedToRef="A").




```
</bat:plans>...
  <bat:userPerformedPlan id="A" name="A">

<bat:abortConditionParameter>abortpara_Cancel1</bat:abortConditionParameter>...
  </bat:userPerformedPlan>
  ...
</bat:plans>
</bat:workingcopy>...
4   <bat:errorEventTask plansid="Trans1" id="A" name="A"
            userTriggered="false"/>....
5   <bpmn:boundaryEvent id="Cancel1" name="Cancel1" attachedToRef="A"
        abortedByParameter="abortpara_Cancel1">...</bpmn:boundaryEvent>
  ...
</bat:workingcopy>
```

Abb. 93: Zeigt die Änderung beim falten der Transaktion Trans1 durch die Initialisierungsphase. Änderungen und neue Elemente wurden fett hervorgehoben.

Errortask-Reduktionsschritt

Der Errortask-Reduktionsschritt wird ausgeführt, wenn ein *bat:errorEventTask* Element in der Arbeitskopie vorhanden ist. Dabei werden die am Muster beteiligten Elemente aus der Arbeitskopie entfernt, und entsprechende Elemente der Zwischendarstellung erzeugt, die den Abbruch der Transaktion registrieren und einen etwaigen alternativen Prozesszweig aktivieren.

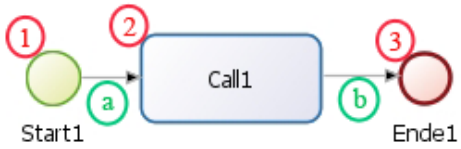
Nach dem Transaktionsfaltungsschritt wird im Errortask-Reduktionsschritt das *bat:errorEventTask* Element (A), das Fehler-Zwischenereignis (Cancel1), bei einem alternativen Zweig, die Alternativaktivität (B), das Terminierungsendereignis (Term1) und die verbindenden Kanten (e3, e4) aus der Arbeitskopie entfernt.

Bei der Reduktion werden mehrere Zwischenelemente erzeugt (siehe auch Abb. 94). Ist ein alternativer Zweig definiert, müssen auch dafür Zwischenelemente generiert werden, die noch hinzukommen.
Ist kein alternativer Zweig (Basisvariante) definiert, wird ein *bat:planActivationPlan* Element in der Zwischendarstellung erzeugt. Es wird verwendet, um auf den normalen und den optional vorhandenen alternativen Zweig zu verweisen. Mit dem *bat:planRef* Kindelement wird jenes Element für den normalen Zweig referenziert und mit dem *bat:onabortRef* Kindelement der alternative Zweig. Ist kein alternativer Zweig definiert, entfällt das *bat:onabortRef* Element.

Ist ein alternativer Zweig vorhanden, sind zusätzliche Elemente notwendig. Ist eine Aktivität im Alternativzweig des Cancel Case Musters definiert, dann referenziert das *bat:onabortRef* Element des *bat:planActivationPlan* Elements auf das *bat:sequentialPlan* Element. Ist keine Aktivität im Alternativzweig definiert, dann referenziert das *bat:onabortRef* Element auf das *bat:endEventPlan* Element.

Abb. 94 zeigt die oben beschriebenen Änderungen des Errortask-Reduktionsschritts am BPMN-Prozess.
Aus der Arbeitskopie (*bat:workingcopy* Element) wurden das *bat:errorEventTask* Element A, das *bpmn:boundaryEvent* Element Cancel1, die Alternativaktivität Task B (*bpmn:task* Element), das Terminierung Endereignis Term1 (*bpmn:endEvent* Element) und die Kanten des alternativen Zweigs e3, e4 entfernt.

In der Zwischendarstellung wurden mehrere Elemente generiert. Das *bat:planActivationPlan* Element Trans1 referenziert mit dem *bat:planRef* Kindelement *bat:userPerformedPlan* Element A, das den normalen Prozessablauf repräsentiert, während mit dem *bat:onabortRef* Kindelement auf den alternativen Zweig verwiesen wird, nämlich auf Element Trans1_alternativeBranch.
A ist ein *bat:userPerformedPlan* Element mit einer Abbruchbedingung (*bat:abortConditionParameter* Element), die auf dem boolschen Parameter abortpara_Cancel1 basiert.
Trans1_alternativeBranch ist ein *bat:sequentialPlan* Element, das die Alternativaktivität B und danach das Terminierung Endereignis Term1 (*bat:endEventPlan* Element) ausführt, welches auf die Abbruchvariabel abortvar_Prozess1 (*bat:terminationVariable* Element) verweist.




```
<bat:variables>
  <bat:abortVariable id="abortvar_Prozess1"/>
  <bat:abortVariable id="abortvar_Trans1"/>
</bat:variables>
<bat:parameters><bat:abortParameter id="abortpara_Cancel1"/></bat:parameters>
<bat:plans>
  <bat:planActivationPlan id="Trans1" name="Trans1">
    <bat:planRef refID="A"/>
    <bat:onabortRef refID="Trans1_alternativeBranch"/>
  </bat:planActivationPlan>
  <bat:sequentialPlan id="Trans1_alternativeBranch"
                      name=" Trans1_alternativeBranch">
    <bat:sourceRef>B</bat:sourceRef>
    <bat:targetRef>Term1</bat:targetRef>
  </bat:sequentialPlan>
  <bat:endEventPlan id="Term1" name="Term1">
    <bat:terminationVariable variableRef="abortvar_Prozess1"/>
  </bat:endEventPlan>
  <bat:userPerformedPlan id="A" name="A">
    <bat:abortConditionParameter>abortpara_Cancel1</bat:abortConditionParameter>
  </bat:userPerformedPlan>
  <bat:planActivationPlan id="Call1" name="Call1">
    <bat:planRef refID="Trans1"/></bat:planActivationPlan>
  <bat:userPerformedPlan id="B" name="B"/>
</bat:plans>
<bat:workingcopy>
  <bpmn:process id="Prozess1" name="Prozess1"
                abortedByVariable="abortvar_Prozess1">
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task id="Call1" name="Call1" calledElement="Trans1"/>
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="Call1" targetRef="Ende1"/>
  </bpmn:process>
</bat:workingcopy>...
```

Abb. 94: Cancel Case Muster aus Abb. 91 nach den Modifikationen des Errortask-Reduktionsschritt der Mustertransformationsphase. Änderungen und neue Elemente wurden fett hervorgehoben

Asbru-Generierungsphase

In der Asbru-Erzeugungsphase werden aus den Elementen der Zwischendarstellung (*bat:planActivationPlan, bat:userPerformedPlan, bat:sequentialPlan* und *bat:endEventPlan* Element) Asbru-Pläne erstellt.

```
<plan name="Call1" title="Call1">
  <plan-body><plan-activation><plan-schema name="Trans1"/></plan-activation></plan-body>
</plan>
<plan name="Trans1" title="Trans1">
  <plan-body>
    <plan-activation><plan-schema name="A"/>
      <on-abort>
        <plan-activation><plan-schema name="Trans1_alternativeBranch"/></plan-activation>
      </on-abort>
    </plan-activation>
  </plan-body>
</plan>
<plan name="A" title="A">
  <conditions>
    <abort-condition>
      <parameter-proposition parameter-name="abortpara_Cancel1">
        <value-description type="equal"><qualitative-constant value="true"/>
        </value-description>...
      </parameter-proposition>
    </abort-condition>
  </conditions>
  <plan-body><user-performed/></plan-body>
</plan>
<plan name="Trans1_alternativeBranch" title="Trans1_alternativeBranch">
  <plan-body><subplans type="sequentially">
    <wait-for><all/></wait-for>
    <plan-activation><plan-schema name="B"/></plan-activation>
    <plan-activation><plan-schema name="Term1"/></plan-activation>
  </subplans></plan-body>
</plan>
<plan name="Term1" title="Term1">
  <plan-body>
    <variable-assignment variable="abortvar_Prozess1">
      <qualitative-constant value="true"/>
    </variable-assignment>
  </plan-body>
</plan>
<plan name="B" title="B"><plan-body><user-performed/></plan-body></plan>
```

Abb. 95: Asbru-Übersetzung der Pläne des Cancel Task Musters aus Abb. 91

Für das *bat:planActivationPlan* Element wird ein Plan (Abb. 95 Plan Trans1, Call1) generiert, der mit dem *plan-body/plan-activation* Element den normalen Zweig (Plan A) aktiviert. Wird dieser abgebrochen, d.h. die Abbruchbedingung ist erfüllt, dann wird der alternative Zweig Plan Trans1_alternativeBranch aktiviert, der durch das *on-abort/plan-activation* Element referenziert wird.

Das *bat:userPerformedPlan* Element wird zu einem Plan (Abb. 95 Plan A, B) übersetzt, der ein *plan-body/user-performed* Element enthält. In Plan A wird auch noch eine Abbruchbedingung (*conditions/abort-condition* Element) definiert, die erfüllt ist, wenn Parameter abortpara_Cancel1 (*parameter-proposition* Element) den Wert true (*qualitative-constant* Element) annimmt.

Für das *bat:sequentialPlan* Element wird ein Plan (Abb. 95 Plan Trans1_alternativeBranch) erzeugt, der die alternative Aktivität (Plan B) und anschließend das Terminierungsendereignis (Plan Term1) sequentiell hintereinander ausführt (*subplans* Element, Attribut type="sequentially").

Das *bat:endEventPlan* Element wird zu einem Plan (Abb. 95 Plan Term1) übersetzt, in dem der Variablen abortvar_Prozess1 (*variable-assignment* Element) der Wert true (*qualitative-constant* Element) zugewiesen wird.

```
<domain-defs>
  <domain name="mainDomain">
...
<parameter-def name="abortpara_Cancel1" required="no" type="Boolean">
  <raw-data-def mode="manual" use-as-context="no"/>
</parameter-def>
...
</domain></domain-defs>
<library-defs>
  <qualitative-scale-def name="Boolean">
    <qualitative-entry entry="false"/>
    <qualitative-entry entry="true"/>
  </qualitative-scale-def>
  <variable-def name="abortvar_Prozess1">
    <scalar-def type="Boolean">
      <initial-value>
        <qualitative-constant value="false"/>
      </initial-value>
    </scalar-def>
  </variable-def>
  <variable-def name="abortvar_Trans1">
    <scalar-def type="Boolean">
      <initial-value>
        <qualitative-constant value="false"/>
      </initial-value>
    </scalar-def>
  </variable-def>
</library-defs>
```

Abb. 96: Asbru-Übersetzung der Parameter- und Variablendefinition des Cancel Task Musters aus Abb. 91

Bei der Transformation des Cancel Case Musters müssen noch Parameter- und Variablendefinitionen (Abb. 96) erstellt werden, damit die oben beschriebenen Parameter und Variablen anwendbar sind.

4.4.6 Strukturierte Schleifen (Structured loop WCP21)

Das Structured Loop Pattern ermöglicht es, eine Aktivität, mehrere Aktivitäten oder einen Teilprozess beliebig oft hintereinander zu aktivieren. Ob eine (weitere) Iteration ausgeführt wird, ist meist an eine Schleifenbedingung geknüpft, die dafür erfüllt sein muss. Sie wird entweder vor oder nach dem Schleifendurchlauf geprüft (siehe auch [14]). Um das Kriterium der Strukturiertheit zu erfüllen, darf nur ein einzelner Ein- und Ausstiegspunkt in dem Schleifenkonstrukt vorhanden sein.

In BPMN gibt es drei unterschiedliche Möglichkeiten strukturierte Schleifen zu bilden:

- Schleifen Task: Ein Task, der mit einer Schleifen-Markierung versehen ist, kann ebenfalls mehrmals hintereinander aktiviert werden. Um einen Task zu markieren, enthält das bpmn:task Element ein *bpmn:standardLoopCharacteristics* Kindelement (siehe Abb. 97).

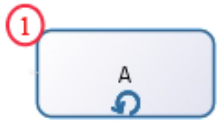
- Schleifen Subprozess: Eine Schleifen-Markierung kann auch dazu verwendet werden, um eine Call Aktivität (*bpmn:callActivity* Element) zu markieren, damit der darin referenzierte Subprozess (bpmn:subprocess Element) iterativ aktiviert wird. Dabei enthält das *bpmn:callActivity* Element ebenfalls ein *bpmn:standardLoopCharacteristics* Kindelement (siehe Abb. 100).

- Zyklische Kantenfolge: Zwei exklusive Gateways sind über zwei gegenläufige Kantenfolgen miteinander verbunden. Das zweite Gateway g2 verzweigt den Kontrollfluss, abhängig von der Schleifenbedingung, und führt entweder eine weitere Iteration durch oder verlässt die Schleife. Das erste Gateway g1 vereinigt den Kontrollfluss vor dem Schleifenkörper und leitet das von g2 kommende Kontrollflusstoken wieder in die vorwärtslaufende Kantenfolge. Wird die Schleifenbedingung am Ende der Iteration überprüft (Repeat-Schleife), dann ist die zu wiederholende Aktivität (mehrere davon oder ein Subprozess) in der vorwärtsläufigen Kantenfolge platziert, die von g1 nach g2 gerichtet ist. Wird die Schleifenbedingung am Anfang der Iteration überprüft (While-Schleife), dann ist die zu wiederholende Aktivität in der rückläufigen Kantenfolge (g2 nach g1) enthalten. Eine formale Definition davon gibt es in Definition 14 auf Seite 75

In folgenden Unterabschnitten wird die Transformation der einzelnen Phase für die eben ausgeführten Schleifenvarianten genauer erörtert.

4.4.6.1 Schleifen Task

Der Schleifen Task ist die einfachste Variante, um eine einzelne Aktivität mehrmals in einer Schleife auszuführen und wird in der BPMN XML-Notation durch ein *bpmn:standardLoopCharacteristics* Kindelement im *bpmn:task* Element gekennzeichnet (siehe Abb. 97). Das *bpmn:standardLoopCharacteristics* Element hat ein *testBefore* Attribut und ein *bpmn:loopCondition* Element. Mit dem *testBefore* Attribut wird spezifiziert, ob die Schleifenbedingung vor (testBefore="true") bzw. nach (testBefore="false") der Iteration überprüft wird. Die Schleifenbedingung wird über das *bpmn:loopCondition* Element angegeben und muss erfüllt sein, um ein weitere Iteration durchzuführen.




```
1  <bpmn:task id="A" name="A">
     <bpmn:standardLoopCharacteristics testBefore="false">
       <bpmn:loopCondition>a==1</bpmn:loopCondition>
     </bpmn:standardLoopCharacteristics>
   </bpmn:task>
```

Abb. 97: Schleifen Task A in BPMN XML-Notation.

Der in Abb. 97 abgebildete Schleifen Task A wird iterativ wiederholt, solange die Schleifenbedingung (a==1 im *bpmn:loopCondition* Element) erfüllt ist. Schleifen Task A wird mindestens einmal ausgeführt, da die Schleifenbedingung erst am Ende der Iteration (testBefore="false") getestet wird.

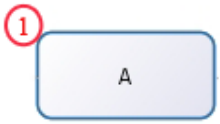
Transformationsphase

Während der Initialisierung wird der den Schleifen Task in der Arbeitskopie nicht verändert, allerdings werden zwei (bei While Schleife) bzw. drei (bei Repeat Schleife) Elemente der Zwischendarstellung erzeugt.
Diese zwei bzw. drei Elemente sind:

- *bat:sequentialPlan* Element: Dieses Element wird nur bei einer Repeat-Schleife benötigt. Wird die Schleifenbedingung nach der Iteration geprüft, muss eine Sequenz (A in Abb. 98) erzeugt werden, um die Repeat-Schleife mit einer While-Schleife zu simulieren. Somit wird die erste obligatorische Iteration durch eine Sequenz imitiert, die zuerst die zu wiederholende Aktivität (A_loopbody) ausführt und danach die While-Schleife (A_whileLoop) aufruft. Dies ist nötig, da Asbru nur eine While-Schleife (*iterative-plan* Element) unterstützt, die die Schleifenbedingung vor der Iteration überprüft.

- *bat:userPerformedPlan* Element: Dies stellt die zu wiederholende Aktivität (A_loopbody in Abb. 98) dar, die vom *bat:sequentialPlan* bzw. *bat:iterativePlan* Element referenziert wird.

- *bat:iterativePlan* Element: Damit wird die While-Schleife abgebildet. Mit dem *bat:do-repeatedly/bat:planRef* Element wird die zu wiederholende Aktivität (A_loopbody in Abb. 98) referenziert. Mit dem *bat:loop-conditions/bat:termination-condition* Element wird die Abbruchbedingung angegeben, die erfüllt sein muss, um die Schleife zu verlassen. Im *bat:loop-conditions/bat:continuation-condition* Element wird die Fortsetzungsbedingung (a==1) spezifiziert, die erfüllt sein muss, um eine weitere Iteration durchzuführen. Es darf jedoch nur in einem der beiden Elemente eine Bedingung angegeben sein.




```
<bat:plans>
  <bat:sequentialPlan id="A" name="A">
    <bat:sourceRef>A_loopbody</bat:sourceRef>
    <bat:targetRef>A_whileLoop</bat:targetRef>
  </bat:sequentialPlan>
  <bat:iterativePlan id="A_whileLoop" name="A_whileLoop">
    <bat:do-repeatedly><bat:planRef refID="A_loopbody"/></bat:do-repeatedly>
    <bat:loop-conditions>
      <bat:termination-condition/>
      <bat:continuation-condition>a==1</bat:continuation-condition>
    </bat:loop-conditions>
  </bat:iterativePlan>
  <bat:userPerformedPlan id="A_loopbody" name="A_loopbody"/>
<bat:plans>
<bat:workingcopy>...
1   <bpmn:task id="A"name="A"/>...
</bat:workingcopy>
```

Abb. 98: Schleifen Task aus Abb. 97 nach den Initialisierungsmaßnahmen. Änderungen und neue Elemente wurden fett hervorgehoben.

Nun kann direkt mit der Erzeugung des Asbru-Codes weitergemacht werden.

Asbru-Erzeugungsphase

In der Asbru-Erzeugungsphase werden aus den o.a. Elementen der Zwischendarstellung entsprechende Asbru-Pläne erzeugt. Das *bat:sequentialPlan* und bat:userPerformedPlan Element wird, wie schon öfter in den vorangegangenen Abschnitten (z.B. Abschnitt 4.4.1.1) beschrieben, transformiert (siehe Abb. 99 Plan A bzw. Plan A_loopbody).

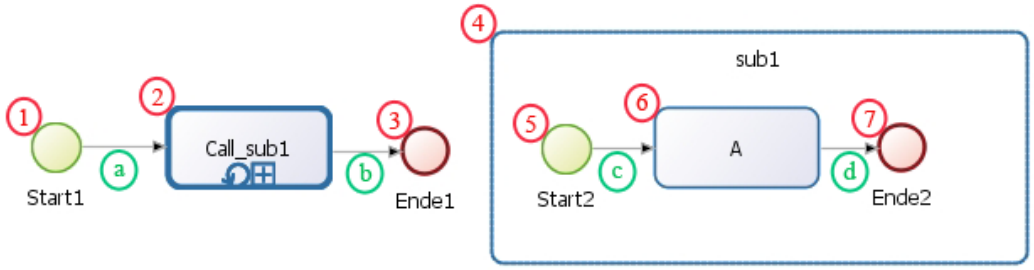
```
<plan name="A" title="A">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A_loopbody"/></plan-activation>
      <plan-activation><plan-schema name="A_whileLoop"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="A_whileLoop" title="A_whileLoop">
  <plan-body>
    <iterative-plan>
      <do-repeatedly>
        < plan-activation><plan-schema name="A_loopbody"/></plan-activation>
      </do-repeatedly>
      <termination-condition><comment text="!( a==1 )"/></termination-condition>
    </iterative-plan>
  </plan-body>
</plan>
<plan name="A_loopbody" title="A_loopbody">
  <plan-body><user-performed/></plan-body>
</plan>
```

Abb. 99: Zeigt die Asbru-Übersetzung des Schleifen Tasks aus Abb. 97.

Für das *bat:iterativePlan* Element wird ein Asbru Plan (Abb. 99 Plan A_whileLoop) erzeugt, dessen *plan-body* Element ein *iterative-plan* Element enthält. Das *iterative-plan* Element enthält das *do-repeatedly/plan-activation/plan-schema* Element, mit dem der zu wiederholende Plan referenziert wird, und das *termination-condition* Element, in dem die Abbruchbedingung, mittels *comment* Element, angegeben wird. Der Ausdruck der Abbruchbedingung entstammt entweder dem *bat:termination-condition* Element oder dem negierten Ausdruck des *bat:continuation-condition* Elements des *bat:iterativePlan* Zwischenelements.

4.4.6.2 Schleifen Subprozess

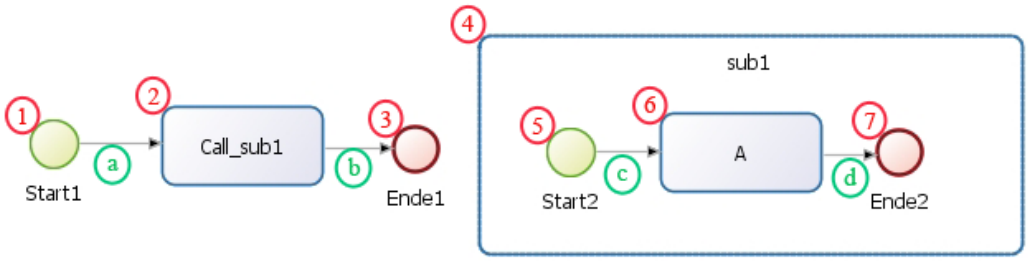
Der Schleifen Subprozess wird verwendet, um einen Teilprozess, also eventuell mehr als eine Aktivität, beliebig oft iterativ auszuführen. Die Call-Aktivität (*bpmn:callActivity* Element), die einen Teilprozess referenziert und aufruft, wird dadurch wiederholt, indem sie ein *bpmn:standardLoopCharacteristics* Kindelement enthält. Das *bpmn:standardLoopCharacteristics* Element wurde bereits im vorigen Abschnitt 4.4.6.1 genauer beschrieben, deshalb wird hier darauf verzichtet.




```
1   <bpmn:startEvent id="Start1" name="Start1"/>
    <bpmn:callActivity="Call_sub1" name="Call_sub1" calledElement="sub_process1">
      <bpmn:standardLoopCharacteristics loopMaximum="-1">
2       <bpmn:loopCondition>a==1</bpmn:loopCondition>
      </bpmn:standardLoopCharacteristics>
    </bpmn:callActivity>
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
    <bpmn:subProcess id="sub1" name="sub1" triggeredByEvent="false">
      5 <bpmn:startEvent id="Start2" name="Start2"/>
4     6 <bpmn:task id="A" name="A"/>
      7 <bpmn:endEvent id="Ende2" name="Ende2"/>
    </bpmn:subProcess>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call_sub1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="Call_sub1" targetRef="Ende1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Start2" targetRef="A"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="Ende2"/>
```

Abb. 100: Schleifen Subprozess in BPMN XML-Notation. Subprozess sub1 wird über die Call-Activität Call_sub1 iterativ aufgerufen.

Der in Abb. 100 gezeigte Prozess besteht aus dem Elternprozess, der aus Start- (Start1), Endereignis (Ende1) und der Call-Aktivität (Call_sub1) besteht, und dem Subprozess (sub1), der Start- (Start2), Endereignis (Ende2) und den Task (A) enthält. Call_sub1 ruft sub1 iterativ auf, solange die Schleifenbedingung (a==1) erfüllt ist.




```
<bat:plans>
  <bat:sequentialPlan id="Call_sub1" name="Call_sub1">
    <bat:sourceRef>sub1</bat:sourceRef>
    <bat:targetRef>Call_sub1_whileLoop</bat:targetRef>
  </bat:sequentialPlan>
  <bat:iterativePlan id="Call_sub1_whileLoop" name="Call_sub1_whileLoop">
    <bat:do-repeatedly><bat:planRef refID="sub1"/></bat:do-repeatedly>
    <bat:loop-conditions>
      <termination-condition/>
      <bat:continuation-condition>a==1</bat:continuation-condition>
    </bat:loop-conditions>
  </bat:iterativePlan>
  <bat:userPerformedPlan id="A" name="A"/>
<bat:plans>
<bat:workingcopy>
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task id="Call_sub1" name="Call_sub1" />
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
    <bpmn:subProcess id="sub1" name="sub1" triggeredByEvent="false">
5     <bpmn:startEvent id="Start2" name="Start2"/>
4   6   <bpmn:task id="A" name="A"/>
7     <bpmn:endEvent id="Ende2" name="Ende2"/>
    </bpmn:subProcess>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call_sub1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="Call_sub1" targetRef="Ende1"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="Start2" targetRef="A"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="A" targetRef="Ende2"/>
<bat:workingcopy>...
```

Abb. 101: Schleifen Subprozess aus Abb. 100 nach den Modifikationen der Initialisierungsphase. Änderungen und neue Elemente wurden fett markiert.

Transformationsphase

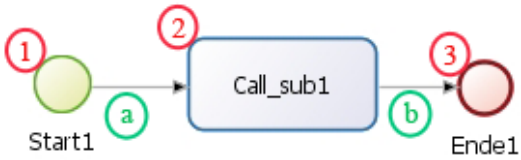
Neben den üblichen Initialisierungsmaßnahmen wird, ähnlich wie beim Schleifentask, ein *bat:iterativePlan* Zwischenelement (Call_sub1_whileLoop in Abb. 101) erzeugt, das eine While-Schleife abbildet. Mit dem *bat:do-repeatedly/bat:planRef* Element wird der zu wiederholende Subprozess (sub1) referenziert. Mit dem *bat:loop-conditions/bat:termination-condition* Element wird die Abbruchbedingung angegeben, die erfüllt sein muss, um die Schleife zu verlassen. Im *bat:loop-conditions/bat:continuation-condition* Element wird die Fortsetzungsbedingung (a==1) spezifiziert, die erfüllt sein muss, um eine weitere Iteration durchzuführen. Es darf jedoch nur in einem der beiden Elemente eine Bedingung angegeben sein.

Wird die Schleifenbedingung nach der Iteration geprüft, muss zusätzlich eine Sequenz (*bat:sequentialPlan* Element Call_sub1 in Abb. 101) erzeugt werden, um die Repeat-Schleife mit einer While-Schleife zu simulieren.

Das *bat:iterativePlan* Element verweist nach den Initialisierungsmaßnahmen auf das *bpmn:subProcess* Element sub1, da der im Subprozess enthaltenen Prozess erst durch die Mustertransformation reduziert werden muss. Der Subprozess in Abb. 101 besteht allerdings nur aus Task A bzw. dem korrespondierenden Zwischenelement und könnte, in diesem speziellen Fall, direkt referenziert werden, da dessen Transformation bereits abgeschlossen ist (*bat:userPerformedPlan* Element A).

Durch die Mustertransformation wird der Subprozess (Prozess der im *bpmn:subProcess* Element enthalten ist) entsprechend den anwendbaren Reduktionsschritten zu einem einzelnen Task reduziert. D.h. enthält der Subprozess nur mehr ein Start- und ein Endereignis, einen Task, der den Teilprozess repräsentiert, und zwei Kanten, die die drei Knoten verbinden, dann kann das *bpmn:subProcess* Element mit allen seinen Kindelementen und den dazugehörigen Kanten aus der Arbeitskopie entfernt werden.

Jenes Zwischenelement, das den Subprozess repräsentiert (Abb. 102 *bat:userPerformedPlan* Element A), soll nun durch das *bat:iterativePlan* und *bat:sequentialPlan* Element referenziert werden. Daher müssen Referenzen auf das *bpmn:subProcess* Element (sub1) und der dafür in der Initialisierungsphase erzeugten Zwischenelemente geändert werden.




```
<bat:plans>
  <bat:sequentialPlan id="Call_sub1" name="Call_sub1">
    <bat:sourceRef>A</bat:sourceRef>
    <bat:targetRef>Call_sub1_whileLoop</bat:targetRef>
  </bat:sequentialPlan>
  <bat:iterativePlan id="Call_sub1_whileLoop" name="Call_sub1_whileLoop">
    <bat:do-repeatedly><bat:planRef refID="A"/></bat:do-repeatedly>
    <bat:loop-conditions>
      <termination-condition/>
      <bat:continuation-condition>a==1</bat:continuation-condition>
    </bat:loop-conditions>
  </bat:iterativePlan>
  <bat:userPerformedPlan id="A" name="A"/>
<bat:plans>
<bat:workingcopy>
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:task="Call_sub1" name="Call_sub1" />
3   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="Call_sub1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="Call_sub1" targetRef="Ende1"/>
<bat:workingcopy>
```

Abb. 102: Schleifen Subprozess aus Abb. 100 nach der Mustertransformation. Änderungen und neue Elemente wurden fett markiert.

Abb. 102 zeigt, dass die ursprünglichen Referenzen auf das *bpmn:subProcess* Element (sub1) in den *bat:iterativePlan* und *bat:sequentialPlan* Elementen auf das *bat:userPerformedPlan* Element A geändert wurden.

Asbru-Erzeugungsphase

In der Asbru-Erzeugungsphase werden aus den o.a. Elementen der Zwischendarstellung entsprechende Asbru-Pläne erzeugt. Das *bat:sequentialPlan* und *bat:userPerformedPlan* Element wird, wie schon öfter in den vorangegangenen Abschnitten (z.B. Abschnitt 4.4.1.1) beschrieben, transformiert (siehe Abb. 103 Plan A bzw. Plan Call_sub1).

Für das *bat:iterativePlan* Element wird ein Asbru Plan (Abb. 103 Plan Call_sub1_whileLoop) erzeugt, dessen *plan-body* Element ein *iterative-plan* Element enthält. Das *iterative-plan* Element enthält das *do-repeatedly/plan-activation/plan-schema* Element, mit dem der zu wiederholende Plan referenziert wird, und das *termination-condition* Element, in dem die Abbruchbedingung mittels *comment* Element angegeben wird. Der Ausdruck der Abbruchbedingung entstammt entweder dem *bat:termination-condition* Element oder dem negierten Ausdruck des *bat:continuation-condition* Elements des *bat:iterativePlan* Zwischenelements.

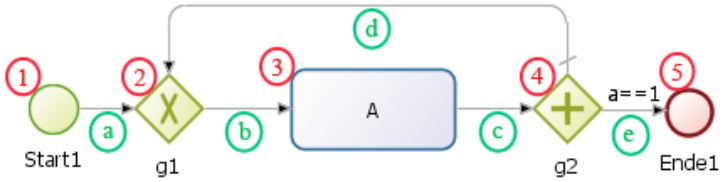
```
<plan name="Call_sub1" title="Call_sub1">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="Call_sub1_whileLoop"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="Call_sub1_whileLoop" title="Call_sub1_whileLoop">
  <plan-body>
    <iterative-plan>
      <do-repeatedly>
        <plan-activation><plan-schema name="A"/></plan-activation>
      </do-repeatedly>
      <termination-condition><comment text="!( a==1 )"/></termination-condition>
    </iterative-plan>
  </plan-body>
</plan>
<plan name="A" title="A"><plan-body><user-performed/></plan-body></plan>
```

Abb. 103: Zeigt die Asbru-Übersetzung des Schleifen Subprozess aus Abb. 100

4.4.6.3 Zyklische Kantenfolge

Eine weitere implementierte Möglichkeit einen Task (oder mehrere davon) iterativ auszuführen, ist die zyklische Kantenfolge. Hier wird mit einer Kantenfolge ein Kreis im Grafen definiert, der es ermöglicht, eine oder mehrere Aktivität(en) mehrmals hintereinander zu aktivieren. Hat dieser Kreis genau einen Ein- und Ausstiegspunkt, dann kann er mit Asbrus *iterative-plan* Element dargestellt werden, da die Schleife strukturiert ist. Wie auch bei den anderen in Abschnitt 4.4.6 beschriebenen Schleifenvarianten, kann auch hier modelliert werden, ob die Schleifenbedingung vor (While-Schleife) oder nach der Iteration (Repeat-Schleife) überprüft wird. Sind die Aktivitäten in der vorwärtsgerichteten Kantenfolge (Zweig) zwischen Ein- und Ausstiegspunkt angeordnet, dann handelt es sich um eine Repeat-Schleife. Sind die Aktivitäten jedoch im rückläufigen Zweig zwischen Aus- und Einstiegspunkt angeordnet, dann handelt es sich um eine While-Schleife.

BPMN definiert eine Schleife durch zwei exklusive Gateways, wobei das eine als Ein- (g1) und das zweite als Ausstiegspunkt (g2) der Schleife dient (siehe Abb. 104). Der Kreis im Grafen teilt sich in zwei Kantenfolgen, nämlich die vorwärtsgerichtete (e2, e3 in Abb. 104) und die rückläufige (e4). Ist die zu wiederholende Aktivität im vorwärtsgerichteten Zweig platziert, wie in Abb. 104, dann handelt es sich um eine Repeat-Schleife. Wäre sie im rückläufigen Zweig platziert, würde es sich um eine While-Schleife handeln. Die Ausstiegs- bzw. Schleifenbedingungen werden über Kantenbedingungen der vom Ausstiegspunkt (g2) ausgehenden Kanten (Ausstiegsbedingung in e5) realisiert.




```
1   <bpmn:startEvent id="Start1" name="Start1"/>
2   <bpmn:exclusiveGateway id="g1" name="g1"/>
3   <bpmn:task id="A" name="A"/>
4   <bpmn:exclusiveGateway id="g2" name="g2"/>
5   <bpmn:endEvent id="Ende1" name="Ende1"/>
a   <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
b   <bpmn:sequenceFlow id="e2" sourceRef="g1" targetRef="A"/>
c   <bpmn:sequenceFlow id="e3" sourceRef="A" targetRef="g2"/>
d   <bpmn:sequenceFlow id="e4" sourceRef="g2" targetRef="g1"/>
    <bpmn:sequenceFlow id="e5" sourceRef="g2" targetRef="Ende1">
e     <bpmn:conditionExpression>a==1</bpmn:conditionExpression>
    </bpmn:sequenceFlow>
```

Abb. 104: Durch zyklische Kantenfolge definierte Repeat-Schleife in BPMN XML-Notation.

Transformationsphase

Als Ein- bzw. Ausstiegspunkt in die zyklische Kantenfolge dienen zwei exklusive Gateways, wobei das eine als Ein- (g1) und das zweite als Ausstiegspunkt (g2) der Schleife dient (siehe Abb. 104 Der Kreis im Grafen teilt sich in zwei Kantenfolgen, nämlich die vorwärtsgerichtete (e2, e3 in Abb. 104) und die rückläufige (e4). Ist die zu wiederholende Aktivität im vorwärtsgerichteten Zweig platziert, wie in Abb. 104, dann handelt es sich um eine Repeat-Schleife. Wäre sie im rückläufigen Zweig platziert, würde es sich um eine While-Schleife handeln.
Nach den üblichen Initialisierungs- und Vereinheitlichungsmaßnahmen kann mit der Mustertransformation begonnen werden. Dabei werden die beiden exklusiven Gateways (g1, g2) und die darin eingschlossenen, beteiligten Elemente durch eine Ersatztask (gen0001) ersetzt und Kanten korrigiert.

Ähnlich wie beim Schleifentask, wird ein *bat:iterativePlan* Zwischenelement (gen0001_whileLoop in Abb. 105) erzeugt, das eine While-Schleife abbildet. Mit dem *bat:do-repeatedly/bat:planRef* Element wird der zu wiederholende Task (A) referenziert. Mit dem *bat:loop-conditions/bat:termination-condition* Element wird die Abbruchbedingung angegeben, die erfüllt sein muss, um die Schleife zu verlassen. Im *bat:loop-conditions/bat:continuation-condition* Element wird die Fortsetzungsbedingung (a==1) spezifiziert, die erfüllt sein muss, um eine weitere Iteration durchzuführen. Es darf jedoch nur in einem der beiden Elemente eine Bedingung angegeben sein.

Wird die Schleifenbedingung nach der Iteration geprüft, muss zusätzlich eine Sequenz (*bat:sequentialPlan* Element gen0001 in Abb. 105) erzeugt werden, um die Repeat-Schleife mit einer While-Schleife zu simulieren. Abb. 105 veranschaulicht die eben beschriebenen Modifikationen.


```
        <bat:plans>
          <bat:sequentialPlan id="gen0001" name="gen0001">
            <bat:sourceRef>A</bat:sourceRef>
            <bat:targetRef>gen0001_whileLoop</bat:targetRef>
          </bat:sequentialPlan>
          <bat:iterativePlan id="gen0001_whileLoop" name="gen0001_whileLoop">
            <bat:do-repeatedly><bat:planRef refID="A"/></bat:do-repeatedly>
            <bat:loop-conditions>
              <bat:termination-condition>a==1</bat:termination-condition>
              <bat:continuation-condition/>
            </bat:loop-conditions>
          </bat:iterativePlan>
          <bat:userPerformedPlan id="A" name="A"/>
        <bat:plans>
        <bat:workingcopy>
1         <bpmn:startEvent id="Start1" name="Start1"/>
2         <bpmn:task id="gen0001" name="gen0001"/>
3         <bpmn:endEvent id="Ende1" name="Ende1"/>
a         <bpmn:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
          <bpmn:sequenceFlow id="e5" sourceRef="gen0001" targetRef="Ende1">
e           <bpmn:conditionExpression>a==1</bpmn:conditionExpression>
          </bpmn:sequenceFlow>
        <bat:workingcopy>
```

Abb. 105: Die Repeat-Schleife aus Abb. 104 nach der Mustertransformation. Änderungen und neue Elemente sind fett hervorgehoben.

Asbru-Erzeugungsphase

In der Asbru-Erzeugungsphase werden aus den o.a. Elementen der Zwischendarstellung entsprechende Asbru-Pläne erzeugt. Das *bat:sequentialPlan* und bat:userPerformedPlan Element wird, wie schon öfter in den vorangegangenen Abschnitten (z.B. Abschnitt 4.4.1.1) beschrieben, transformiert (siehe Abb. 106 Plan gen0001 bzw. Plan A).

Für das *bat:iterativePlan* Element wird ein Asbru Plan (Abb. 106 Plan gen0001_whileLoop) erzeugt, dessen *plan-body* Element ein ***iterative**-plan* Element enthält. Das *iterative-**plan*** Element enthält das *do-repeatedly/plan-**activation/plan**-schema* Element, mit dem der zu wiederholende Plan referenziert wird und das ***termination**-condition* Element, in dem die Abbruchbedingung mittels *comment* Element angegeben wird. Der Ausdruck der Abbruchbedingung entstammt entweder dem *bat:termination-condition* Element oder dem negierten Ausdruck des *bat:**continuation**-condition* Elements des *bat:iterativePlan* Zwischenelements.

```
<plan name="gen0001" title="gen0001">
  <plan-body>
    <subplans type="sequentially">
      <wait-for><all/></wait-for>
      <plan-activation><plan-schema name="A"/></plan-activation>
      <plan-activation><plan-schema name="gen0001_whileLoop"/></plan-activation>
    </subplans>
  </plan-body>
</plan>
<plan name="gen0001_whileLoop" title="repeatLoop A_whileLoop">
  <plan-body>
    <iterative-plan>
      <do-repeatedly>
        <plan-activation><plan-schema name="A"/></plan-activation>
      </do-repeatedly>
      <termination-condition><comment text="a==1"/></termination-condition>
    </iterative-plan>
  </plan-body>
</plan>
<plan name="A" title="A"><plan-body><user-performed/></plan-body></plan>
```

Abb. 106: Auszug der Asbru-Übersetzung der Repeat-Schleife aus Abb. 104

5 Evaluierung des Prototypen

Nachdem die Arbeitsweise des Transformationssystems in den vorangegangen Abschnitten anhand der einzelnen Kontrollflussmuster gezeigt wurde, wird in diesem Teil der Arbeit beschrieben, wie die Korrektheit des Prototyps überprüft wird.

Dabei werden Beispiele transformiert und die erzeugten Asbru-Leitlinien, im Asbru XML-Format, einer syntaktischen und semantischen Prüfung unterzogen. Die syntaktische Prüfung erfolgt, indem die generierte Leitlinie gegen die Asbru-DTD validiert wird. Die semantische Prüfung wird durch manuellen Vergleich der Leitlinie mit dem BPMN-Prozess erreicht, wobei der Fokus auf der Äquivalenz des Kontrollfluss liegt.

Die Arbeitsweise des Transformationssystems wurde an aussagekräftigen Muster-Testfällen getestet, anhand deren die Arbeitsweise des Transformationssystems überprüft wurde. Dies sind BPMN-Prozesse, die aus übersetzbaren Varianten des jeweiligen Kontrollflussmusters bestehen und die Übersetzung von Normal- und Sonderfällen einzelner Muster abdecken.
Abgesehen von den Muster-Testfällen wurde ein BPMN-Prozess als Testfall herangezogen, der mehrere verschachtelte Kontrollflussmuster beinhaltet, um so das musterübergreifende Verhalten des Prototyps zu testen.
Weiters wurde eine real einsetzbare Leitlinie („Prostate Cancer“-Leitlinie) herangezogen, um das Transformationssystem an einer realen medizinischen Leitlinie zu testen.

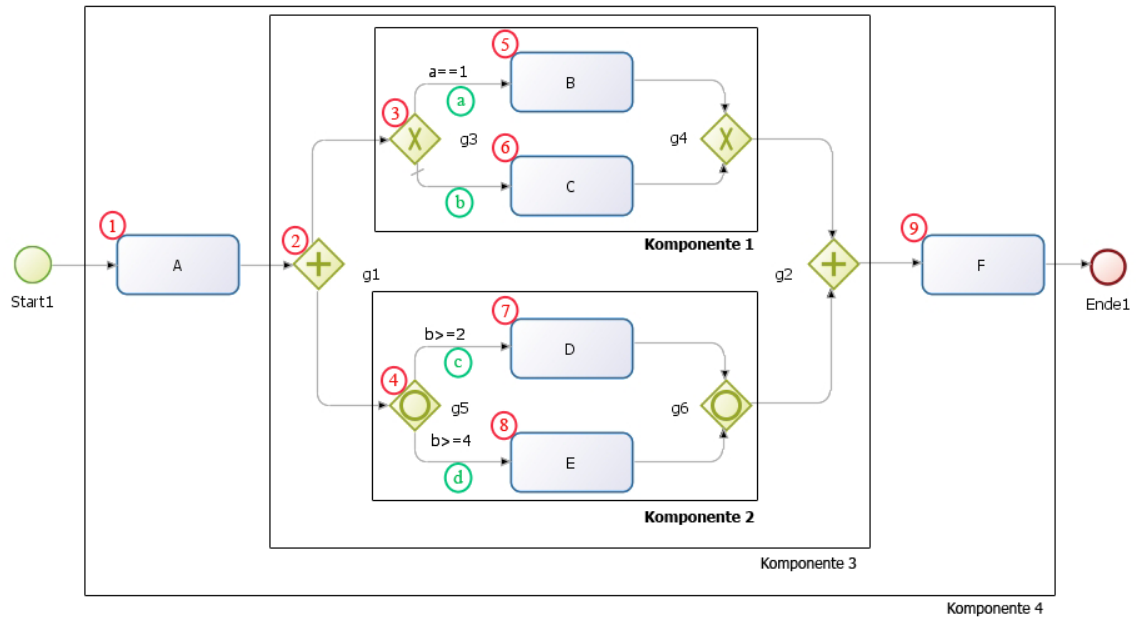
In den folgenden beiden Abschnitten wird das Vorgehen und die Ergebnisse anhand eines Beispiels mit verschachtelten Kontrollflussmustern (Kapitel 5.1) und der „Prostate Cancer“-Leitlinie (Kapitel 5.2) gezeigt.

5.1 Verschachtelte Kontrollflussmuster

BPMN-Prozesse bestehen in der Regel nicht nur aus einem einzelnen Kontrollflussmuster, sondern es werden meist mehrere Muster verwendet, die ineinander verschachtelt oder nebeneinander vorkommen.
Abb. 107 zeigt einen BPMN-Prozess, der aus vier ineinander verschachtelte Komponenten besteht, die jeweils ein Pattern darstellen. Komponente 1 und 2 stellen einen exklusiven bzw. inklusiven Block dar, während Komponente 3 und 4 einen parallelen Block bzw. das Sequence-Pattern ergeben.

Das Transformationssystem kann auch BPMN-Prozesse mit mehreren verschachtelten Kontrollflussmustern übersetzen. In der Mustertransformationsphase werden transformierbare Muster identifiziert. Wird mehr als ein Muster identifiziert, dann wird anhand einer Prioritätenliste entschieden, welches ausgeführt wird. In dem darauf folgenden Reduktionsschritt der Mustertransformationsphase wiederholt sich diese Auswahl wieder. Wie schon in den vorangegangenen Kapiteln (4.4.1 bis 4.4.6) beschrieben, werden reduzierte Muster in der Arbeitskopie des zu transformierenden BPMN-Prozesses durch eine Ersatz-Task ersetzt. Dadurch können weitere Muster transformierbar werden, die davor nicht reduzierbar waren. Im Beispiel in Abb. 107 sind Komponente 1 und 2 direkt reduzierbar, Komponente 3 wird transformierbar, nachdem Komponente 1 und 2 reduziert wurden. Komponente 4 kann nach der Reduktion von Komponente 3 übersetzt werden.
Im Transformationssystem wird die Identifikation der Muster durch XSLT-Variable realisiert, denen XPath-Ausdrücke zugeordnet sind. Diese XPath-Ausdrücke werden vom XSLT-Prozessor ausgewertet, bevor die Transformationsregeln ausgeführt werden. Anhand dieser Variablen ist ersichtlich, welche Muster transformierbar sind und welche nicht.

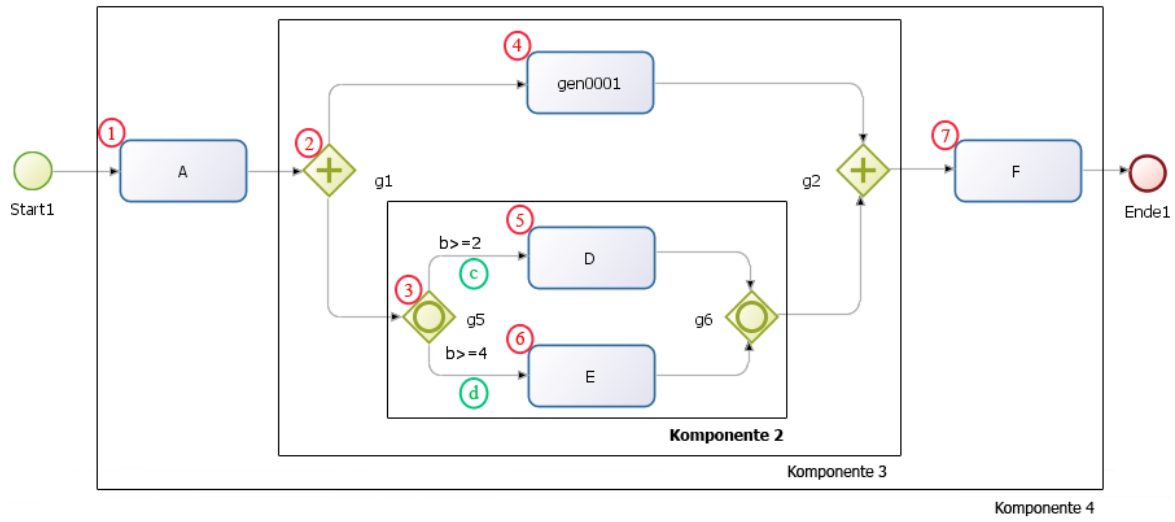
Die Prioritätenliste, in welcher die Muster ausgeführt werden, ist durch eine Switch-Anweisung realisiert, wobei jeder Zweig für die Transformation eines Musters steht.
Im Folgenden wird die Funktionsweise des Transformationssystems am Beispiel in Abb. 107 gezeigt.




```
        <bat:plans>
            <bat:userPerformedPlan id="A" name="A"/>...
            <bat:userPerformedPlan id="F" name="F"/>
        </bat:plans>
        <bat:workingcopy>...
1         <model:task id="A" name="A"/>
2         <model:parallelGateway id="g1" name="g1"/>
3         <model:exclusiveGateway id="g3" name="g3"/>
4         <model:inclusiveGateway id="g5" name="g5"/>
5         <model:task id="B" name="B"/>
6         <model:task id="C" name="C"/>
7         <model:task id="D" name="D"/>
8         <model:task id="E" name="E"/>
          <model:exclusiveGateway id="g4" name="g4"/>
          <model:inclusiveGateway id="g6" name="g6"/>
          <model:parallelGateway id="g2" name="g2"/>
9         <model:task id="F" name="F"/>
                ...
          <model:sequenceFlow id="e5" sourceRef="g3" targetRef="B">
a           <model:conditionExpression>a==1</model:conditionExpression>
          </model:sequenceFlow>
b         <model:sequenceFlow id="e6" sourceRef="g3" targetRef="C"/>
          <model:sequenceFlow id="e7" sourceRef="g5" targetRef="D">
c           <model:conditionExpression>b&gt;=2</model:conditionExpression>
          </model:sequenceFlow>
          <model:sequenceFlow id="e8" sourceRef="g5" targetRef="E">
d           <model:conditionExpression>b&gt;=4</model:conditionExpression>
          </model:sequenceFlow>...
        </bat:workingcopy>
```

Abb. 107: Multi-Pattern Beispiel mit vier verschachtelten Pattern. Jede Komponente stellt ein Kontrollflusspattern dar, das während des Transformationsprozesses reduziert wird. Aktuell reduzierbare Muster *Komponente 1* und *2*.

Da die Transformation der einzelnen Muster bereits in den vorangegangenen Kapiteln erörtert wurde, beschränkt sich die Beschreibung der Transformationsschritte auf die Mustertransformationsphase und wird am o.a. Beispiel veranschaulicht. Dabei liegt der Fokus auf den Auswirkungen auf die verbleibenden Muster/Komponenten, die bei der Reduktion eines Musters bzw. einer Komponente entstehen.




```
<bat:plans>
  <bat:exclusivePlan id="gen0001" name="exclusive Block B C">
    <bat:planRef refID="B"/>
    <bat:planRef refID="C"/>
  </bat:exclusivePlan>...
</bat:plans>
<bat:workingcopy>...
1   <model:task id="A" name="A"/>
2   <model:parallelGateway id="g1" name="g1"/>
3   <model:inclusiveGateway id="g5" name="g5"/>
4   <model:task id="gen0001" name="exclusive Block B C"/>
5   <model:task id="D" name="D"/>
6   <model:task id="E" name="E"/>
    <model:parallelGateway id="g2" name="g2"/>
    <model:inclusiveGateway id="g6" name="g6"/>
7   <model:task id="F" name="F"/>
    ...
    <model:sequenceFlow id="e3" sourceRef="g1" targetRef="gen0001"/>
    <model:sequenceFlow id="e4" sourceRef="g1" targetRef="g5"/>
    <model:sequenceFlow id="e7" sourceRef="g5" targetRef="D">
c     <model:conditionExpression>b&gt;=2</model:conditionExpression>
    </model:sequenceFlow>
    <model:sequenceFlow id="e8" sourceRef="g5" targetRef="E">
d     <model:conditionExpression>b&gt;=4</model:conditionExpression>
    </model:sequenceFlow>
    <model:sequenceFlow id="e13" sourceRef="gen0001" targetRef="g2"/>...
</bat:workingcopy>
```

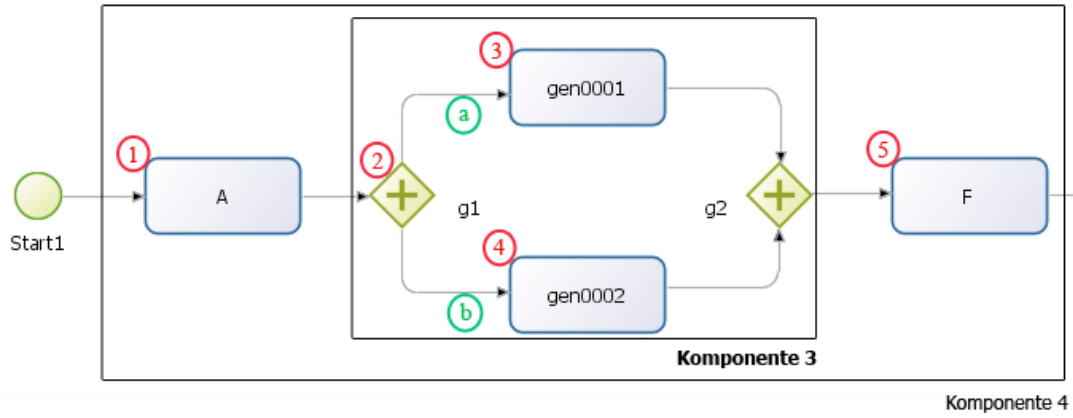
Abb. 108: Multi-Pattern Beispiel aus Abb. 107 nachdem Komponente 1 (exklusiver Block (WCP 4/5)) reduziert wurde. Aktuell reduzierbare Muster *Komponente 2*. Änderungen u. neue Elemente wurden fett hervorgehoben.

Komponente 1 (exklusiver Block (WCP4/5))

Abb. 107 zeigt, dass der exklusive Block (*Komponente 1*) und der inklusive Block (*Komponente 2*), beide, reduzierbar sind (siehe Kapitel 4.4.1.4 und 0.). Da der exklusive Block in der Implementierung höher priorisiert wurde, wird er zuerst reduziert (siehe Abb. 108).

Komponente 2 (inklusiver Block (WCP6/7))

Der inklusive Block (*Komponente 2*) ist nun als einziges Muster reduzierbar, da die Reduktion von Komponente 1 allein noch nichts an der Transformierbarkeit der verbleibenden Komponenten 3 und 4 ändert. Nach der Reduktion von *Komponente 2* sieht der Beispielprozess wie in Abb. 109 aus.




```
        <bat:plans>
          <bat:inclusivePlan id="gen0002" name="inclusive Block D E" >
                <bat:planRef refID="D"/>
                <bat:planRef refID="E"/>
          </bat:inclusivePlan>...
        </bat:plans>
        <bat:workingcopy>...
1         <model:task id="A" name="A"/>
2         <model:parallelGateway id="g1" name="g1"/>
3         <model:task id="gen0001" name="exclusive Block B C"/>
4         <model:task id="gen0002" name="inclusive Block D E"/>
          <model:parallelGateway id="g2" name="g2"/>
5         <model:task id="F" name="F"/>
          ...
          <model:sequenceFlow id="e2" sourceRef="A" targetRef="g1"/>
a         <model:sequenceFlow id="e3" sourceRef="g1" targetRef="gen0001"/>
b         <model:sequenceFlow id="e4" sourceRef="g1" targetRef="gen0002"/>
          <model:sequenceFlow id="e14" sourceRef="gen0002" targetRef="g2"/>
          <model:sequenceFlow id="e13" sourceRef="gen0001" targetRef="g2"/>
          <model:sequenceFlow id="e15" sourceRef="g2" targetRef="F"/>...
        </bat:workingcopy>
```

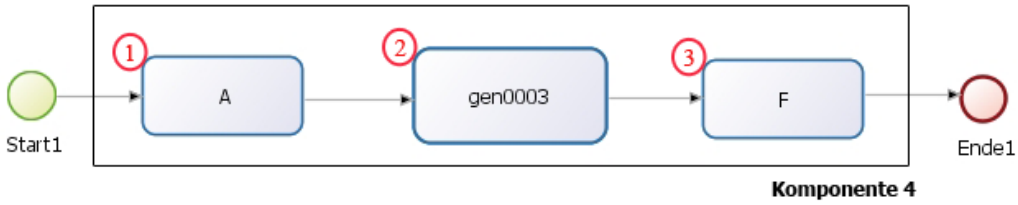
Abb. 109: Multi-Pattern Beispiel aus Abb. 108 nachdem Komponente 2 (inklusiver Block (WCP 6/7)) reduziert wurde. Aktuell reduzierbare Muster *Komponente 3*. Änderungen u. neue Elemente wurden fett hervorgehoben.

Komponente 3 (paralleler Block (WCP2/3))

Durch die Transformation der *Komponenten 1* und *2* wird nun auch *Komponente 3* reduzierbar, da die Tasks *gen0001* und *gen0002* die *Komponenten 1* und *2* ersetzen (siehe auch Kapitel 4.4.1.2). Die Ergebnisse dieses Reduktionsschritts ist in Abb. 110 ersichtlich.

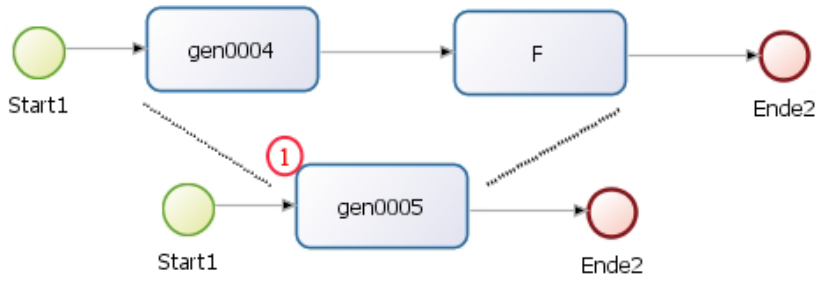
Komponente 4 (Sequence (WCP1))

Abb. 110 zeigt eine Sequenz von Tasks A, gen003, F, die durch Transformieren von *Komponente 3* entstanden ist. Diese kann in zwei Reduktionsschritten (siehe Kapitel 4.4.1.1) zu einem einzelnen Task reduziert werden (Abb. 111). Dabei ersetzt Task gen0004 die Tasks A und gen0003 und Task gen0005 die Tasks gen0004 und F.




```
<bat:plans>
  <bat:parallelPlan id="gen0003" name="parallel Block gen0001 gen0002">
    <bat:planRef refID="gen0001"/>
    <bat:planRef refID="gen0002"/>
  </bat:parallelPlan>...
</bat:plans>
<bat:workingcopy>...
  <model:startEvent id="Start1" name="Start1"/>
1 <model:task id="A" name="A"/>
2 <model:task id="gen0003" name="parallel Block gen0001 gen0002"/>
3 <model:task id="F" name="F"/>
  <model:endEvent id="Ende1" name="Ende1"/> ...
  <model:sequenceFlow id="e2" sourceRef="A" targetRef="gen0003"/>
  <model:sequenceFlow id="e15" sourceRef="gen0003" targetRef="F"/>...
</bat:workingcopy>
```

Abb. 110: Multi-Pattern Beispiel aus Abb. 109 nachdem Komponente 3 (paralleler Block (WCP 2/3)) reduziert wurde. Aktuell reduzierbare Muster *Komponente 4.* Änderungen u. neue Elemente wurden fett hervorgehoben.




```
<bat:plans>
  <bat:sequentialPlan id="gen0005" name="sequence gen0004 to F">
    <bat:sourceRef>gen0004</bat:sourceRef>
    <bat:targetRef>F</bat:targetRef>
    </bat:sequentialPlan>
  <bat:sequentialPlan id="gen0004" name="sequence A to gen0003">
    <bat:sourceRef>A</bat:sourceRef>
    <bat:targetRef>gen0003</bat:targetRef>
  </bat:sequentialPlan>...
</bat:plans>
<bat:workingcopy>
  <model:startEvent id="Start1" name="Start1"/>
1 <model:task id="gen0005" name="sequence gen0004 to F"/>
  <model:endEvent id="Ende1" name="Ende1"/>
  <model:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0005"/>
  <model:sequenceFlow id="e16" sourceRef="gen0005" targetRef="Ende1"/>
</bat:workingcopy>
```

Abb. 111: Multi-Pattern Beispiel aus Abb. 110 nachdem Komponente 4 (Sequence (WCP 1)) in zwei Schritten reduziert wurde. Änderungen u. neue Elemente wurden fett hervorgehoben.

Das Beispiel zeigt, dass das Transformationssystem auch BPMN-Prozesse mit mehreren verschiedenen, verschachtelten Kontrollflussmustern transformieren kann, sofern das Transformationssystem jedes einzelne Muster reduzieren kann.

5.2 „Prostate Cancer“ Leitlinie

Die „Prostate Cancer“ Leitlinie ist eine NCCN[11] Leitlinie zur Behandlung von Patienten mit Prostatakrebs und stellt eine von interdisziplinären Experten ausgearbeitete medizinische Leitlinie dar. Auf Basis dieser Leitlinie (siehe [42]) wurde ein BPMN 2.0 Modell[12] erstellt, das wir mit dem Transformationssystem in ein Asbru-Modell überführen und evaluieren.

Beschreibung der „Prostatate Cancer“-Leitlinie:
Der Hauptprozess ist in 49 Teilprozesse (38 *model:subProcess* und 11 *model:adHocSubProcess* Elemente) unterteilt, in denen 171 Task Elemente verwendet werden.
Es wurden die Kontrollflussmuster Sequence (WCP1, 87 Mal), ExclusiveChoice (WCP4) mit Simple Merge (WCP5, 49 Mal), Multiple Choice (WCP6) mit Synchronising Merge (WCP7, 6 Mal), Interleaved

[11] National Comprehensive Cancer Network
[12] Wir danken Dr. Mar Marcos von der Knowledge Engineering Group der Universitat Jaume I (Castellon, Spanien), dass Sie uns das BPMN 2.0 Modell der NCCN Prostate Cancer Leitlinie, das in Ihrer Gruppe entwickelt wurde, zur Verfügung gestellt hat.

Parallel Routing (WCP40, 11 Mal), Structured Loop (WCP21, 15 Mal) verwendet. Das Transformationssystem übersetzt die „Prostatate Cancer“-Leitlinie in 194 Reduktions- und 36 Optimierungsschritte. Diese Zahlen sollen die Komplexität der Leitlinie untermauern, da eine detaillierte Beschreibung auf Grund der Größe hier nicht möglich ist.

Die syntaktische Prüfung der Asbru-Übersetzung der „Prostatate Cancer“-Leitlinie gegen die Asbru-DTD ergab, dass syntaktische Fehler vorhanden sind, die durch nicht übersetzte Bedingungen entstehen. D.h. Kantenbedingungen im BPMN-Prozess sollten als Bedingungen in Asbru-Plänen (z.B. Filterbedingungen (*filter-condition* Elemente) siehe auch 3.1.2) umgesetzt werden. Dies ist jedoch nicht Teil dieser Arbeit und des implementierten Prototyps. Dadurch werden syntaktische Fehler, auf Grund der unvollständigen Bedingungen, verursacht. Dies ist jedoch ein allgemeines Problem, das bei jeder Leitlinienübersetzung auftreten kann und nicht nur speziell bei der „Prostatate Cancer“-Leitlinie.

Die semantische Prüfung ergab, dass die generierte Asbru-Datei den Kontrollfluss äquivalent zum Eingangsprozess, entsprechend dem Mapping aus Abschnitt 4.2, abbildet. Da in der Praxis der in den Abschnitten 4.4.1 bis 4.4.6 vorgestellten Pattern auch Sonderfälle vorkommen, mussten davor folgende zwei Kontrollflussmuster erweitert und die Implementierung adaptiert werden, damit die „Prostatate Cancer“-Leitlinie vollständig übersetzt werden konnte:

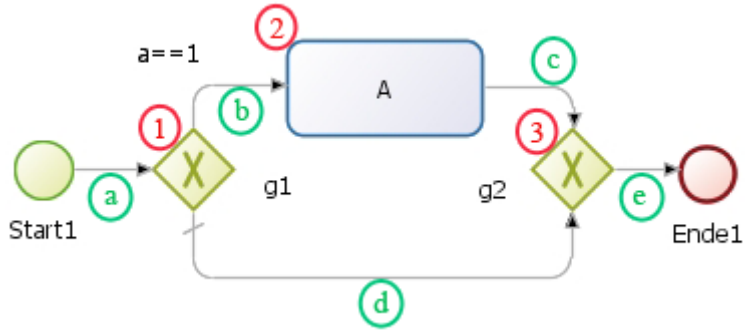
1) Exklusiver Block (WCP 4 u. 5) mit „Bypass“ und
2) Interleaved Routing (WCP 40) mit Task mit verbundenem Endereignis

In den folgenden beiden Abschnitten werden diese Adaptionen im Detail besprochen.

5.2.1.1 Exklusiver Block (WCP 4 u. 5) mit „Bypass“

Im Unterschied zum konventionellen exklusiven Block (siehe Abschnitt 4.4.1.4) in BPMN gibt es eine bzw. auch mehrere Kante(n), die jeweils eine direkte Verbindung zwischen dem verzweigenden und dem synchronisierenden exklusiven Gateway herstellt. D.h. der exklusive Block enthält mindestens einen Zweig, der keinen Task enthält (Abb. 112 Marke d: Kante e4). Dies wird im Weiteren als Bypass-Kante bzw. Bypass-Zweig bezeichnet und wird verwendet, um den exklusiven Block zu verlassen ohne eine Aktivität auszuführen. Die Bypass-Kante kann aktiviert werden falls Bedingungen der anderen Zweige nicht zutreffen (Default-Kante wie in Abb. 112), oder es muss eine dezidierte Bypass-Bedingung erfüllt sein.

Diese Variante des exklusiven Blocks kann mit Asbru *subplans* Element nicht ohne Weiteres dargestellt werden und bedarf einer gesonderten Behandlung. Da mit dem *subplans* Element kein leerer Default-Zweig festgelegt werden kann und nur dezidierte Pläne referenziert werden können, wird für jede Bypass-Kante ein Dummy-Plan generiert. Diese Dummy-Pläne können vom *subplans* Element referenziert werden und auch die etwaig vorhandene Bypass-Bedingung der Bypass-Kante umsetzen.
Da sich der exklusive Block mit Bypass vom konventionellen exklusiven Block nur in der Mustertransformationsphase unterscheidet, wird auch nur diese beschrieben.




```
        <bat:workingcopy>...
1         <model:exclusiveGateway id="g1" name="g1"/>
2         <model:task id="A" name="A"/>
3         <model:exclusiveGateway id="g2" name="g2"/>...

a         <model:sequenceFlow id="e1" sourceRef="Start1" targetRef="g1"/>
          <model:sequenceFlow id="e2" sourceRef="g1" targetRef="A">
b           <model:conditionExpression>a==1</model:conditionExpression>
          </model:sequenceFlow>
c         <model:sequenceFlow id="e3" sourceRef="A" targetRef="g2"/>
d         <model:sequenceFlow id="e4" sourceRef="g1" targetRef="g2"/>
e         <model:sequenceFlow id="e5" sourceRef="g2" targetRef="Ende1"/>...
        </bat:workingcopy>
```

Abb. 112: Auszug eines BPMN-Prozesses mit exklusivem Block mit Bypass mit dem Tasks A vor dem Transformationsprozess

Transformationsphase

Durch die üblichen Initialisierungsmaßnahmen werden Tasks und Kantenbedingung, der darin eingehenden Kanten, übersetzt.

Ein exklusiver Block mit Bypass charakterisiert sich dadurch, dass ein verzweigendes exklusives Gateway über einen oder mehrere Zweig(e) mit einem synchronisierenden exklusiven Gateway verbunden ist. Das verzweigende exklusive Gateway hat genau eine eingehende Kante und, entsprechend der Anzahl der Zweige des exklusiven Blocks, eine oder mehrere ausgehende Kante(n). Das synchronisierende exklusive Gateway hat, entsprechend der Anzahl der Zweige des exklusiven Blocks, eine oder mehrere eingehende Kante(n) und genau eine ausgehende Kante. Die einzelnen Zweige bestehen aus einem Task, der genau eine eingehende, die aus dem verzweigenden Gateway entspringt, und genau eine ausgehende Kante besitzt, die im synchronisierenden Gateway endet. Mindestens ein Zweig, der Bypass-Zweig, besteht aus einer Kante, die aus dem verzweigenden Gateway entspringt und im synchronisierenden Gateway endet und optional eine Bypass-Bedingung (*model:conditionExpression* Kindelement) enthält.

Enthalten die Zweige andere Muster (z.B. Verschachtelung von Blöcken), müssen diese zuerst durch vorangegangene Reduktionsschritte der Mustertransformationsphase reduziert und dabei durch einen Ersatztask ersetzt werden.


```
        <bat:plans>
          <bat:exclusivePlan id="gen0001" name="exclusive Block A" >
            <bat:planRef refID="A"/>
            <bat:planRef refID="DummyPlan_e4"/>
          </bat:exclusivePlan>
          <bat:userPerformedPlan id="DummyPlan_e4" name="DummyPlan_e4">
            <bat:filterConditionExpression>!( a==1 )</bat:filterConditionExpression>
           </bat:userPerformedPlan>
          <bat:userPerformedPlan id="A" name="A">
            <bat:filterConditionExpression>a==1</bat:filterConditionExpression>
          </bat:userPerformedPlan>
        </bat:plans>
        <bat:workingcopy>...
1         <model:task id="A" name="A"/>....

a         <model:sequenceFlow id="e1" sourceRef="Start1" targetRef="gen0001"/>
e         <model:sequenceFlow id="e5" sourceRef="gen0001" targetRef="Ende1"/>...
        </bat:workingcopy>
```

Abb. 113: Zeigt den BPMN-Prozess aus Abb. 112, nachdem der exklusive Block mit Bypass reduziert wurde. Änderungen und neue Elemente wurden fett hervorgehoben.

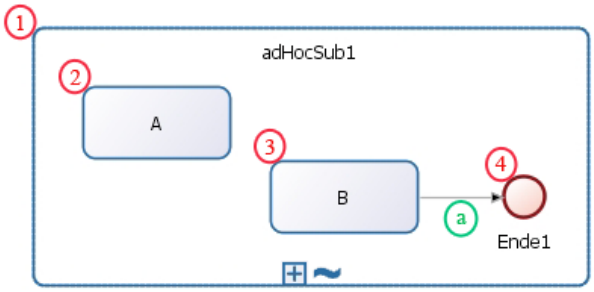
Wird ein exklusiver Block mit Bypass bei der Mustertransformation identifiziert, wird ein *bat:exclusivePlan* Element als Element der Zwischendarstellung erzeugt (Abb. 113 gen001), dessen *bat:planRef* Kindelemente die übersetzten Tasks der einzelnen Zweige (A) und die Dummy-Pläne (DummyPlan_e4) referenzieren. Für jede Bypass-Kante wird ein *bat:userPerformedPlan* Element als Dummy-Plan erzeugt (Abb. 113 DummyPlan_e4), das ein etwaiges *bat:filterConditionExpression* Kindelement mit der Bypass-Bedingung enthält. Ist die Bypass-Bedingung der Kante leer, muss sie die negierte disjunktive Verknüpfung aller anderen Verzweigungsbedingungen enthalten, da der Dummy-Plan aktiviert werden soll, wenn alle anderen Bedingungen nicht erfüllt sind.
Wie beim konventionellen exklusiven Block wird bei der Reduktion aus dem BPMN-Prozess in der Arbeitskopie das verzweigende und synchronisierende exklusive Gateway, die Tasks in den einzelnen Zweigen und die im Block enthaltenen Kanten entfernt. Stattdessen wird ein neuer Ersatztask (Task gen0001) eingefügt und die Kante, die vorher in den exklusiven Block wies bzw. die daraus herausführt, verweist auf den bzw. entspringt dem neuen Ersatztask.

In Abb. 113 ist zu sehen, wie der Reduktionschritt den exklusiven Block mit Bypass aus dem BPMN-Prozess in der Arbeitskopie entfernt, und er durch einen Ersatztask gen0001 substituiert wird. Dabei werden die exklusiven Gateways g1, g2, die Tasks A und die Kanten e2-e4 entfernt. Außerdem wird bei der Kante e1 die Zielreferenz und bei Kante e5 die Ursprungsreferenz mit der Referenz auf gen0001 aktualisiert. Weiters wurde das *bat:plans/bat:exclusivePlan* Element eingefügt, das mit seinen *bat:planRef* Kindelementen auf die Pläne A und DummyPlan_e4 referenziert. Für die Bypass-Kante e4 wurde ein *bat:userPerformedPlan* Element DummyPlan_e4 erzeugt, dessen *bat:filterConditionExpression* Kindelement die negierte disjunktiv verknüpften Verzweigungsbedingungen des Blocks enthält (!(a==1)).

Die Übersetzung der Elemente der Zwischendarstellung in Asbru-Elemente erfolgt wie im konventionellen exklusiven Block (Abschnitt 4.4.1.4).

5.2.1.2 Interleaved Routing (WCP 40) mit Task mit verbundenem Endereignis

Das konventionelle Interleaved Routing Muster (WCP 40) wird in BPMN durch einen AdHoc-Subprozess ausgedrückt, der durch das Transformationssystem reduziert werden kann, wenn ausschließlich Tasks, jedoch keine Kanten, Start- oder Endereignisse darin enthalten sind (siehe Abschnitt 4.4.4.2).



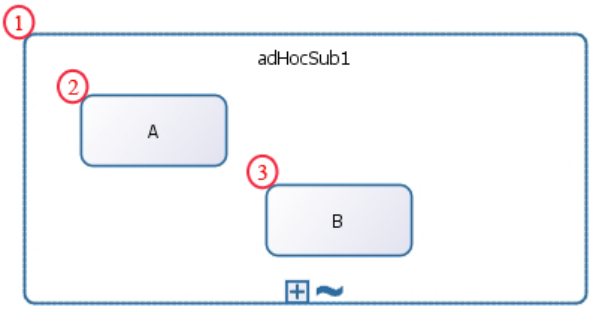

```
<workingcopy>...
  <model:adHocSubProcess id="adHocSub1" name="adHocSub1">
2   <model:task id="A" name="A"/>
3   <model:task id="B" name="B"/>
1 4 <model:endEvent id="Ende1" name="Ende1"/>
a   <model:sequenceFlow id="e2" sourceRef="B" targetRef="Ende1"/>

    <model:completionCondition id="sid-ce29111f">false</model:completionCondition>
  </model:adHocSubProcess>...
</workingcopy>
```

Abb. 114: AdHoc Subprozess mit Task B mit verbundenem Endereignis vor dem Endereignis-Reduktionshilfsschritt der Mustertransformationsphase

Die „Prostatate Cancer“-Leitlinie enthält auch AdHoc-Subprozesse, in denen auch Tasks vorkommen, die untereinander verbunden sind und dadurch Teilprozesse bilden, die aus verschiedene Kontrollflussmuster bestehen und in einem Endereignis enden. Durch vorangegangene Reduktionsschritte werden die Teilprozesse zwar zu einem einzelnen Tasks (z.B. Task B in Abb. 114) reduziert, dieser besitzt jedoch eine ausgehende Kante (e2), die in einem Endereignis (Ende1) endet. Dadurch wird der AdHoc-Subprozess durch die konventionelle Variante des Interleaved Routing Musters nicht reduziert.

Abb. 114 zeigt einen AdHoc-Subprozess adHocSub1, der die Tasks A, B und das Endereignis Ende1 enthält. Weiters ist Task B über eine ausgehende Kante e2 mit dem Endereignis verbunden. Task B könnte z.B. für einen durch vorangegangene Reduktionsschritte reduzierten Teilprozess stehen.

Als Lösung wird ein *Endereignis-Reduktionshilfsschritt* eingeführt, der das Endereignis und die darin eingehende Kante entfernt, wenn ein einzelner Task in einem AdHoc-Subprozess keine eingehende Kante aufweist und eine einzelne ausgehende besitzt, die in einem Endereignis endet. Der *Endereignis-Reduktionshilfsschritt* verändert den BPMN-Prozess in der Arbeitskopie dahingehend, dass er in einem darauf folgenden Reduktionsschritt,wie das konventionelle Interleaved Routing Muster reduziert werden kann. Daher werden keine Elemente der Zwischendarstellung erzeugt oder manipuliert.




```
<workingcopy>...
  <model:adHocSubProcess id="adHocSub1" name="adHocSub1">
2     <model:task id="A" name="A"/>
3     <model:task id="B" name="B"/>

1
      <model:completionCondition id="sid-ce29111f">false</model:completionCondition>
  </model:adHocSubProcess>...
</workingcopy>
```

Abb. 115: AdHoc Subprozess aus Abb. 114 nach dem Endereignis-Reduktionshilfsschritt bei der Mustertransformation

Abb. 115 zeigt, wie das Endereignis Ende1 und die Kante e2 aus dem AdHoc-Subprozess adHocSub1 entfernt wurde. Somit kann der AdHoc-Subprozess in einem weiteren Reduktionsschritt transformiert werden.

6 Schlussbetrachtung und weitere Studien

Medizinische Leitlinien sind Dokumente, die eingesetzt werden, um die Diagnose und Behandlung von Patienten qualitativ zu verbessern und kosteneffizienter zu gestallten [1]. Neben der rein textuellen Beschreibung oder in Form von Entscheidungsbäumen, liegen medizinische Leitlinien auch als Prozessdiagramme vor [2].
BPMN 2.0 ist eine standardisierte, grafische Notation, um Prozessabläufe zu definieren, und kann genutzt werden, um Prozessdiagramme für medizinische Leitlinien zu modellieren. BPMN ist im Bereich der Visualisierung von Geschäftsprozessen weit verbreitet und bietet eine gute Modellierungsinfrastruktur durch viele Modellierungswerkzeuge.
Asbru wurde speziell entwickelt, um medizinische Leitlinien zu modellieren und zu beschreiben. Asbru-Modelle können durch einen Interpreter ausgeführt werden und sollen so das medizinische Personal bei der Diagnose und Behandlung unterstützen [3]. Asbru wird noch durch wenige Modellierungswerkzeuge unterstützt und ist noch wenig verbreitet.

Ziel dieser Arbeit ist ein Transformationssystem zu implementieren, das BPMN-Modelle in Asbru-Modelle überführt. Somit könnten Leitlinien in standardisierten BPMN-Prozessen erstellt werden, dessen Notation leicht zu erlernen ist und durch ausgereifte Werkzeuge unterstützt wird, und können trotzdem durch den Asbru-Interpreter, nach der Übersetzung, ausgeführt werden.

Um dieses Transformationssystem zu implementieren, bedarf es einer (1) theoretischen Analyse der unterstützen Kontrollflussmuster, (2) eines Mappings zwischen BPMN-Elementen und semantisch äquivalenten Asbru-Konstrukten und (3) der Auswahl einer Transformationsstrategie. Ein weiterer Beitrag dieser Arbeit ist (4) die Implementierung eines Software-Prototyps des Transformationssystems, basierend auf XSLT-Stylesheets, das die ausgewählte Strategie und das definierte Mapping in einer ausführbaren Form umsetzt.

(1) Um zu Analysieren, wie und welche Kontrollflussmuster von BPMN bzw. Asbru unterstützt werden und in einem Transformationssystem umsetzbar sind, wurden bereits vorhandene musterbasierte Analysen von BPMN und Asbru in dieser Arbeit gegenüber gestellt (Kapitel 4.1). Dabei ergibt sich, dass von den 20 untersuchten Kontrollflussmustern 14 von beiden Modellen, 2 von keinem Modell und 4 nur von BPMN unterstützt werden (Tabelle 1).

(2) Wichtig für eine korrekte Transformation zwischen den zwei Prozessmodellen ist ein Mapping, welches BPMN-Entitäten semantisch gleichwertigen Asbru-Konstrukten gegenüberstellt. Dieses Mapping (Kapitel 4.2) entstand durch Analyse der BPMN- und Asbru-Spezifikation und beschränkt sich auf Elemente, die zur Darstellung des Kontrollflusses benötigt werden. Nicht für jedes BPMN-Element konnte ein semantisch äquivalentes Asbru-Element gefunden werden. Jedoch konnte für jene Elemente, die zur Darstellung der o.a. Muster benötigt werden, ein Mapping gefunden werden. Aus der Analyse kann gefolgert werden, dass einzelne BPMN-Gateways nicht übersetzt werden können, sondern nur damit gebildete Blöcke. Weiters ist zu erkennen, dass Asbru über keine Mechanismen verfügt, die BPMN-Events abbilden können. Manche Event-Typen können jedoch durch manuelle Eingaben oder boolsche Variablen simuliert werden.

(3) Da BPMN dem graforientierten und Asbru dem blockorientierten Paradigma folgt, um den Kontrollfluss darzustellen, ist eine Transformation von BPMN- zu Asbru-Modellen nicht trivial. Da in BPMN der Kontrollfluss über Kanten an die nachfolgenden Tasks weitergegeben wird, sind Prozessabläufe modellierbar, die mit strukturierten Asbru-Komponenten nicht gebildet werden können.

Dies betrifft im allgemeinen Schleifen (Kapitel 3.2.3.1), die mehr als einen Ein- oder Ausstiegspunkt in den Schleifenkörper besitzen, oder Blöcke (Kapitel 3.2.3.2), die am Synchronisationspunkt einen anderen Gateway-Typ aufweisen als am Verzweigungspunkt.
Diese Arbeit untersucht Transformationssysteme und –strategien mit ähnlicher Problematik, hat deren Lösungsansätze als Teil des State-of-the-Art-Berichts (Kapitel 3.2.5) zusammengefasst und auf Anwendbarkeit auf das BPMN-zu-Asbru Transformationssystem überprüft (Kapitel 4.3). Dabei wurden drei Strategien analysiert, die sich bei der Übersetzung diverser graforientierter Prozessmodelle zu BPEL-Modellen bewährt haben.
Die Wahl der im Software-Prototyp zu implementierenden Transformationsstrategie fiel auf die *Structure-Identification* Strategie. Dabei werden nur Konstrukte des graforientierten BPMN-Quellmodells übersetzt, die auf eine strukturierte Komponente im blockorientierten Asbru-Zielmodell abgebildet werden können. Dies hat den Vorteil, dass der übersetzte Asbru-Code gut lesbar ist und dadurch besser überprüft werden kann. Die beiden anderen Strategien, nämlich *Element-Preservation* und *Event-Condition-Action-Rule*, wären aufgrund der Asbru-Spezifikation zwar theoretisch anwendbar, durch die fehlende Interpreter Unterstützung essentieller Asbru-Elemente jedoch nicht praktisch überprüfbar.

(4) Im Anschluss an die theoretischen Vorarbeiten wurde ein Software-Prototyp implementiert, der die unterstützten Kontrollflussmuster identifizieren kann und das Mapping und die Transformationsstrategie praktisch umsetzt.
Der Prototyp Transformiert das BPMN-Eingangsmodell in einer ersten Transformationsphase in eine Zwischendarstellung, die bereits vereinfacht Asbru Struktur abbildet und optimiert werden kann. In einer zweiten Asbru-Erzeugungsphase wird die Zwischendarstellung in Asbru-Code übersetzt. Diese Phasen werden durch ein bzw. mehrfaches Anwenden von XSLT-Stylesheets realisiert.
Das implementierte Transformationssystem kann die o.a. unterstützten Kontrollflussmuster, meist in einer Variante, identifizieren und entsprechend dem definierten Mapping in ein Asbru-Modell umsetzen. Es können auch BPMN-Prozesse mit mehreren verschiedenen verschachtelten Mustern transformiert werden. Der Prototyp kann jedoch keine BPMN-Kantenbedingungen in Asbru- Bedingungskonstrukte übersetzen, da dies nicht in der abgegrenzten Zielsetzung dieser Arbeit enthalten ist.

In dieser Arbeit wurden theoretische Grundlagen und Analysen durchgeführt, um ein BPMN-zu-Asbru Transformationssystem zu realisieren, und ein Prototyp erstellt, der Teile davon implementiert. Auf Grund der Abgrenzung dieser Arbeit konnten jedoch nicht alle Ideen umgesetzt werden und könnten Teil weiterführender Arbeiten sein.

Der Software-Prototyp, der im Zuge dieser Arbeit umgesetzt wurde, erzeugt keine gültigen Asbru-Bedingungskonstrukte, wie sie z.B. in *filter*-, *setup*- oder *completion-condition* Elementen verwendet werden, sondern erzeugt nur einen Kommentar mit dem Bedingungsausdruck. Ergänzende Forschungen könnten sich damit beschäftigen, diesen Ausdruck in ein gültiges Asbru-Bedingungskonstrukt überzuführen, um so den aktuellen Prototypen zu erweitern.

Aufgrund Asbrus fehlender Interpreterunterstützung wurden die beiden alternativen Transformationsstrategien nicht umgesetzt. Diese Strategien, nämlich *Element-Preservation* und *Event-Condition-Action-Rule*, könnten in einer neuen eigenständigen Implementierung angewendet werden, wenn der Entwicklungsstand des Asbru-Interpreters eine Überprüfung zulässt.

Ein weiteres interessantes Thema für eine weiterführende Arbeit könnte die Anwendung des Verfahrens aus Abschnitt 3.3 in XSLT sein. Das Verfahren beschäftigt sich mit der Reorganisation von unstrukturierten Schleifen mit mehreren Ein- bzw. Ausstiegspunkten, die unter bestimmten Bedingungen in strukturierte Schleifen umgewandelt werden können. Dieses Verfahren beruht auf Fortführungsvariablen (continuation variables), die jeder Komponente mit mehr als einem Ein- oder

Ausstiegspunkt zugeordnet wird. Daraus werden Gleichungen hergeleitet und diese vereinfacht. Dieses Resultat repräsentiert die reorganisierte Schleife. Mit diesem Verfahren könnte der aktuelle Prototyp erweitert werden.

7 Referenzen

[1] N. Mulyar, W. M. Aalst und M. Peleg, „A pattern-based analysis of clinical computer- iterpretable guideline modeling languages," *BPM Center Report BPM-06-29, BPM Center, Technische Universiteit Eindhoven,* 2006.

[2] R. Kosara, S. Miksch und A. Seyfang, „Tools for acquiring clinical guidelines in Asbru," *In Proceedings of the Sixth World Conference on Integrate Design and Process Technology (IDPT'02),* 2002.

[3] Y. Shahar, S. Miksch und P. Johnson, „The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines," *Artificial Intelligence in Medicine,* Bd. 14, Nr. 1–2, pp. 29-51, 1998.

[4] C. Ouyandg, W. M. Aalst, M. Dumas und A. H. Hofstede, „Translating BPMN to BPEL," 2006-01.

[5] C. Ouyang, M. Dumas, A. H. M. ter Hofstede und W. M. P. van der Aalst, „From BPMN Process Models to BPEL Web Services," *International Conference on Web Services ICWS '06,* pp. 285-292, 2006.

[6] I. (. Object Management Group, „Business Process Model and Notation (BPMN)," 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/PDF/. [Zugriff am 11 4 2012].

[7] OMG, „OMG Unified Modeling Language Superstructure, Version 2.4..1," 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. [Zugriff am 15 6 2012].

[8] W. M. Aalst, „Verification of Workflow Nets," *ICATPN '97 Proceedings of the 18th International Conference on Application and Theory of Petri Nets,* pp. 407-426, 1997.

[9] G. Keller, M. Nüttgens und A. W. Scheer, „Semantische Prozessmodellierung auf der Grundlage„Ereignisgesteuerter Prozeßketten (EPK)"," *Inst. fuer Wirtschaftsinformatik, Saarbruecken, Germany,* Nr. 89, 1992.

[10] W. M. P. van der Aalst und A. H. M. ter Hofstede, „YAWL: yet another workflow language," *Information Systems,* Bd. 30, Nr. 4, pp. 245-275, 2005.

[11] OASIS, „Web Services Business Process Execution Language Version 2.0," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. [Zugriff am 15 6 2012].

[12] A. Seyfang, R. Kosara und S. Miksch, „Asbru's Reference Manual, Asbru Version 7.3," [Online]. Available: http://www.asgaard.tuwien.ac.at/asbru_7_3/asbru_7.3_reference.pdf. [Zugriff am 15 6 2012].

[13] J. Mendling, H. Reijers und A. H. Wil, „Seven Process Modeling Guidelines (7PMG).," *Information*

and Software Technology, Nr. 52, pp. 127-136, 2010.

[14] W. M. Aalst, A. H. Hofstede, N. Mulyar und N. Russell, „Workflow Control-Flow Patterns: A Revised View," 2006. [Online]. Available: http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf.

[15] P. Wohed, W. van der Aalst, M. Dumas, A. Hofstede und N. Russell, „Pattern-based Analysis of BPMN - an extensive evaluation of the Control-flow the Data and the Resource Perspectives," *BPM Center Report BPM-06-17, BPMcenter.org,* 2006.

[16] M. Murzek und G. Kramler, „BUSINESS PROCESS MODEL TRANSFORMATION ISSUES The top 7 adversaries encountered at defining model transformations," *Proceedings of the ninth international conference on enterprise information systems,* p. 144-151*,* 2007.

[17] P. Votruba, A. Seyfang, M. Paesoldl und S. Miksch, „Environment-Driven Skeletal Plan Execution for the Medical Domain," *European Conference on Artificial Intelligence (ECAI-2006),* pp. 847-848, 2006.

[18] S. Sendall und W. Kozaczynski, „Model transformation: the heart and soul of model-driven software development," *Software, IEEE,* Bd. 20, Nr. 5, pp. 42-45, 2003.

[19] OMG, „Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1," 2011. [Online]. Available: http://www.omg.org/spec/QVT/1.1/PDF/. [Zugriff am 9 7 2012].

[20] F. Jouault und I. Kurtev, „Transforming Models with ATL," in s *Satellite Events at the MoDELS 2005 Conference*, Bd. 3844, Springer Berlin / Heidelberg, 2006, pp. 128-138.

[21] M. Strommer, „Model Transformation By-Example," Vienna University of Technology, 01.05.2008.

[22] W3C, „XSL Transformations (XSLT) Version 2.0," 2007. [Online]. Available: http://www.w3.org/TR/xslt20/. [Zugriff am 28 8 2012].

[23] W3C, „Extensible Stylesheet Language (XSL) Version 1.1," 2006. [Online]. Available: http://www.w3.org/TR/xsl11/. [Zugriff am 28 8 2012].

[24] H. Elliotte Rusty und M. W. Scott, XML-in a Nutshell, 3. Hrsg., Köln: O'Reilly, 2005.

[25] W3C, „XML Path Language (XPath) 2.0 (Second Edition)," 2010. [Online]. Available: http://www.w3.org/TR/xpath20/. [Zugriff am 28 8 2012].

[26] w3schools, „XSLT 1.0 Turtorial," [Online]. Available: http://www.w3schools.com/xsl/. [Zugriff am 28 8 2012].

[27] Z. Jörg und M. Jan, „Epc-Based Modelling Of Bpel Processes".*In Proceedings of MITIP 2005, Italy.*

[28] Oliver Kopp, Daniel Martin, Daniel Wutke, Frank Leymann und Ulrich Frank, „The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages," *Enterprise*

Modelling and Information Systems, Bd. 4, Nr. 1, p. 3-13, 2009.

[29] J. Mendling, K. B. Lassen und U. Zdun, „On the transformation of control flow between block-oriented and graph-oriented process modelling languages," *International journal of business process integration and management,* Bd. 3, Nr. 2, pp. 96-108, Oct. 2008.

[30] Jan C. Recker und Jan Mendling, „On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages," Namur University Press, 2006, p. 521-532.

[31] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski und A. P. Barros, „Workflow Patterns," *Distributed and Parallel Databases,* Bd. 14, Nr. 1, pp. 5-51, 2003.

[32] C. Ouyang, M. Dumas, S. Breutel und A. ter Hofstede, „Translating Standard Process Models to BPEL," Bd. 4001, Springer Berlin / Heidelberg, 2006, pp. 417-432.

[33] M. Murzek, G. Kramler und E. Michlmayr, „Structural Patterns for the Transformation of Business Process Models," *Models for Enterprise Computing 2006 - International Workshop at EDOC 2006,* p. 43-52, 2006.

[34] M. Strommer, M. Murzek und M. Wimmer, „Applying Model Transformation By-Example on Business Process Modeling Languages," LNCS 4802, Springer, 2007, p. 116-125.

[35] K. Czarnecki und S. Helsen, „Feature-based survey of model transformation approaches," *IBM SYSTEMS JOURNAL,* Bd. 45, Nr. 3, pp. 621-645, 2006.

[36] OMG, „Object Constraint Language Version 2.2," 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2/PDF. [Zugriff am 9 7 2012].

[37] M. Strommer, M. Wimmer, H. Kargl und G. Kramler, „Towards Model Transformation Generation By-Example," 40, IEEE Computer Scociety, 2007, p. 285-286.

[38] L. García-Bañuelos, „Pattern Identification and Classification in the Translation from BPMN to BPEL," Bd. 5331, Springer Berlin / Heidelberg, 2008, pp. 436-444.

[39] J. Koehler und R. Hauser, „Untangling Unstructured Cyclic Flows – A Solution Based on Continuations," Bd. 3290, Springer Berlin / Heidelberg, 2004, pp. 121-138.

[40] J. Richard, P. David und P. Peshav, „The Program Structure Tree: Computing Control Regions in Linear Time," *ACM Sigplan'94 PLDI,* pp. 171-185, 1994.

[41] N. Mulyar, W. M. Aalst und M. Peleg, „A pattern-based analysis of clinical computer-iterpretable guideline modeling languages," *J Am Med Inform Assoc,* Bd. 14, Nr. 6, pp. 781-787, Nov.-Dec. 2007.

[42] M. L. James, A. J. Andrew, B. R. Robert, B. Barry, B. Erik und et.al., „NCCN Clinical Practice Guidelines in Oncology: Prostate Cancer," NCCN, 2012. [Online]. Available: http://www.tri-kobe.org/nccn/guideline/urological/english/prostate.pdf. [Zugriff am 26 6 2013].

[43] H. Rainer, F. Michael, K. M. Jochen und V. Jussi, „Combining analysis of unstructured workflows with transformation to structured workflows," *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International,* Bd. 10, pp. 129-140, 2006.

[44] I. (. Object Management Group, „BPMN 2.0 by Example. Version 1.0," 2010. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/examples/PDF. [Zugriff am 24 8 2012].

[45] V. Peter, S. Andreas, P. Michael und M. Silvia, „The Asbru Interpreter Project," 2005. [Online]. Available: http://ieg.ifs.tuwien.ac.at/projects/interpreter/index.html. [Zugriff am 20 11 2012].

Anhang A

1.1. Beispiel eines BPMN Prozesses

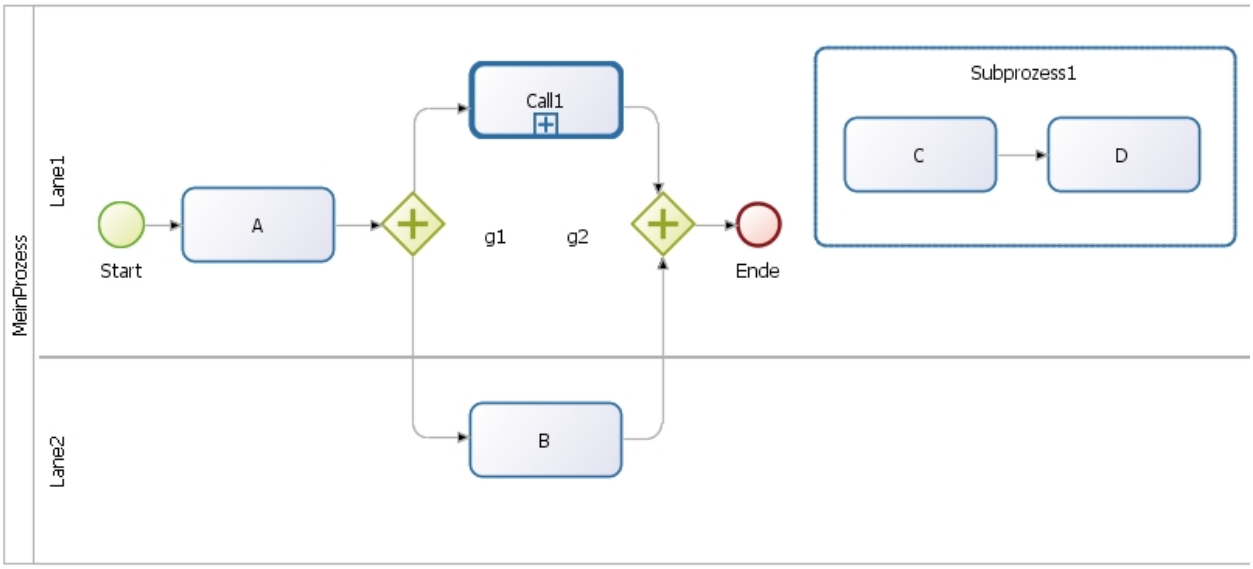


Abb. 116: BPMN Prozess mit einem Pool, zwei Lanes und einem Subprozess

Der Beispielprozess in BPMN-Notation aus Abb. 116 zeigt einen Ablauf, in dem Aktivität A ausgeführt wird und danach Aktivität Call1 und B parallel ausgeführt werden. Call1 aktiviert den Subprozess Subprozess1, der Aktivität C und D sequentiell hintereinander ausführt. Nach dem beide Aktivitäten Call1 und B beendet wurden, wird auch der Beispielprozess beendet.
Der nachfolgende Absatz zeigt die XML-Serialisierung des BPMN-Prozesses aus Abb. 116:

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <definitions
3    targetNamespace="http://www.omg.org/spec/BPMN/20100524/MODEL"
4    xmlns:di="http://www.omg.org/spec/BPMN/20100524/DI">
5
6    <!-- collaboration diagramm with single participating process -->
7    <collaboration id="MeinProzessDiagramm">
8      <participant id="_PXGrAM9IEeGzS7anfNrNyQ" name="MeinProzess"
9            processRef="MeinProzess"/>
10   </collaboration>
11
12   <process id="MeinProzess" name="MeinProzess">
13     <!-- lanes and their nodes -->
14     <laneSet id="MeinProzess_laneSet">
15       <lane id="Lane1" name="Lane1">
16         <flowNodeRef>Start</flowNodeRef>
17         <flowNodeRef>Ende</flowNodeRef>
18         <flowNodeRef>g1</flowNodeRef>
19         <flowNodeRef>A</flowNodeRef>
20         <flowNodeRef>g2</flowNodeRef>
21         <flowNodeRef>Subprozess1</flowNodeRef>
22         <flowNodeRef>C</flowNodeRef>
23         <flowNodeRef>D</flowNodeRef>
24         <flowNodeRef>Call1</flowNodeRef>
25       </lane>
26       <lane id="Lane2" name="Lane2">
```

```
27            <flowNodeRef>B</flowNodeRef>
28          </lane>
29        </laneSet>
30        <!-- nodes -->
31        <startEvent id="Start" name="Start"/>
32        <endEvent id="Ende" name="Ende"/>
33
34        <parallelGateway id="g1" name="g1"/>
35        <parallelGateway id="g2" name="g2"/>
36        <!-- subprocess -->
37        <subProcess id="Subprozess1" name="Subprozess1" triggeredByEvent="true">
38          <task id="C" name="C"/>
39          <task id="D" name="D"/>
40        </subProcess>
41
42        <callActivity id="Call1" name="Call1" calledElement="Subprozess1"/>
43        <task id="A" name="A"/>
44        <task id="B" name="B"/>
45        <!-- edges -->
46        <sequenceFlow id="_l4Tw4MdiEeGXR_0b8JWVBQ" sourceRef="Start" targetRef="A"/>
47        <sequenceFlow id="_h89t8MdoEeGQCui00nPQgg" sourceRef="A" targetRef="g1"/>
48        <sequenceFlow id="_tCkk4MdyEeGQCui00nPQgg" sourceRef="g2" targetRef="Ende"/>
49        <sequenceFlow id="_A4ZlwM9IEeGzS7anfNrNyQ" sourceRef="g1" targetRef="B"/>
50        <sequenceFlow id="_BkWs0M9IEeGzS7anfNrNyQ" sourceRef="g1" targetRef="C"/>
51        <sequenceFlow id="_DGQkcM9IEeGzS7anfNrNyQ" sourceRef="C" targetRef="g2"/>
52        <sequenceFlow id="_EP334M9IEeGzS7anfNrNyQ" sourceRef="B" targetRef="g2"/>
53      </process>
54
55      <di:BPMNDiagram name="MeinProzessDiagramm3">
56        <!-- diagramm layout information -->
57        ...
58      </di:BPMNDiagram>
59    </definitions>
```

Da in dem Beispiel aus Abb. 116 auch ein Pool vorkommt, handelt es sich hierbei um ein BPMN Collaboration Diagramm, an dem der Prozess (pool wird als *process* Element serialisiert) „MeinProzess“ als einziger Prozess teilnimmt (Zeile 6-10). „MeinProzess“ (Zeile 12-53) enthält auch zwei Lanes, die mit einem *laneSet* Element dargestellt werden, in dem für jede Lane die darin vorkommenden Knoten über deren *ID*s referenziert werden (Zeile 14-29). Außer den Lanes enthält der Prozess auch noch Knoten verschiedener Typen (Zeile 30-44) und Sequenzflusskanten (*sequenceFlow*, Zeile 45-52). Unter den Knoten befindet sich auch ein Subprozess (Zeile 36-40), der, wie ein Prozess, Knoten und Kanten enthalten kann, und eine Call-Aktivität (Zeile 42), die auf den Subprozess verweist.
Weiters werden Layout-Informationen zu den Elementen serialisiert, um Position und Größe zu speichern. Diese Information ist hier nicht angeführt, da sie für den Transformationsprozess unbedeutend ist, und wäre im *BPMNDiagramm* Element zu finden (Zeile 55-58).
Für genauere Informationen sei auf [6] und [44]verwiesen.

1.2. Beispiel einer Asbru Plan Library

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plan-library SYSTEM "Asbru_7.3h_DELTA.dtd">
<plan-library>
  <library-info title="NewPlanLibrary">
    <administrative-data original-author="AsbruViewNew"/>
  </library-info>
  <domain-defs>
    <domain name="mainDomain">
      <parameter-group>
        <parameter-def name="age" required="yes" type="Age">
          <raw-data-def mode="manual" use-as-context="no" />
        </parameter-def>
      </parameter-group>
    </domain>
  </domain-defs>

  <library-defs>
    <qualitative-scale-def name="Boolean">
      <qualitative-entry entry="false" />
      <qualitative-entry entry="true" />
    </qualitative-scale-def>

    <variable-def  name="variable1"  >
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="true"/>
        </initial-value>
      </scalar-def>
    </variable-def>
  </library-defs>
  <plans>
    <plan-group>
      <plan name="Sequence" title="Sequence">
        <plan-body>
          <subplans type="sequentially">
            <wait-for>
              <all/>
            </wait-for>

            <plan-activation>
              <plan-schema name="InclusiveChoice"/>
            </plan-activation>
          </subplans>
        </plan-body>
      </plan>

      <plan name="InclusiveChoice" title="InclusiveChoice">
        <plan-body>
          <subplans type="any-order" wait-for-optional-subplans="yes" >
            <wait-for>
              <none/>
            </wait-for>
            <plan-activation>
```

```
      <plan-schema name="Plan_A"/>
    </plan-activation>
    <plan-activation>
      <plan-schema name="Plan_B"/>
    </plan-activation>
  </subplans>
</plan-body>
</plan>

<plan name="Plan_A" title="Plan A">
  <conditions>
    <filter-precondition>
      <simple-condition>
        <comparison type="less-than">
          <left-hand-side>
            <parameter-ref name="age" />
          </left-hand-side>
          <right-hand-side>
            <numerical-constant value="30" />
          </right-hand-side>
        </comparison>
      </simple-condition>
    </filter-precondition>
  </conditions>
  <plan-body>
    <user-performed />
  </plan-body>
</plan>

<plan name="Plan_B" title="Plan B">
  <conditions>
    <filter-precondition>
      <constraint-combination type="and">
        <simple-condition>
          <comparison type="less-than">
            <left-hand-side>
              <parameter-ref name="age" />
            </left-hand-side>
            <right-hand-side>
              <numerical-constant value="10" />
            </right-hand-side>
          </comparison>
        </simple-condition>
        <plan-state-constraint state="completed">
          <plan-pointer >
            <static-plan-pointer plan-name="Plan_A" />
          </plan-pointer>
          <time-annotation>
            <now />
          </time-annotation>
        </plan-state-constraint>
      </constraint-combination>
```

```
106             </filter-precondition>
107           </conditions>
108           <plan-body>
109             <user-performed />
110           </plan-body>
111         </plan>
112       </plan-group>
113     </plans>
114   </plan-library>
```

Das o.a. Beispiel einer Asbru Plan Library zeigt wie ein Parameter „age“ in der Domain „mainDomain“ (Zeile 8-15), der Datentyp „Boolean“ (Zeile 17-21) und die Variable „variable1“ (Zeile 23-30) definiert wird. Plan „Sequence“ (Zeile 30-45) ruft seine Subpläne, also hier nur Plan „InclusiveChoice“ (Zeile 47-61), in sequentieller Reihenfolge auf.
Plan „InclusiveChoice“ aktiviert „Plan_A“ (Zeile 63-81) und „Plan_B“ (Zeile 83-113) und führt sie, falls deren Filtervorbedingungen erfüllt sind, aus. Dabei können die Pläne in beliebiger Abfolge nicht überlappend ausgeführt werden, und es wird auch auf die Beendigung optionaler Pläne gewartet, um „InclusiveChoice“ zu beenden (Zeile 49). Durch die Spezifizierung der Fortsetzungsbedingung mit dem *none* Element (Zeile 50-52) gibt es nur optionale Subpläne.
Die Filterbedingung von „Plan_A“ (Zeile 65-76) ist erfüllt, wenn der Parameter „age“ kleiner als 30 ist.
Die Filterbedingung von „Plan_B“ (Zeile 85-106) ist erfüllt, wenn der Parameter „age“ kleiner als 10 ist (Zeile 87-96), und „Plan_A“ erfolgreich beendet wurde (Zeile 97-105).
Zusammenfassend wird der Parameter „age“ und die Variable „variable1“ definiert. Plan_A wird ausgeführt, wenn das Alter eines Patienten unter 30 Jahren ist. Ist das Alter unter 10 Jahren, wird zusätzlich noch Plan_B ausgeführt, nachdem Plan_A beendet wurde.
Für die detaillierte Bedeutung der Elemente sei auf die Asbru-Spezifikation in [12] verwiesen.

Anhang B

1. Element-Preservation Strategie

Anbei finden sie das vollständige Asbru-Code Beispiel zu Beispiel 2a aus Abschnitt 0.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plan-library SYSTEM "Asbru_7.3h_DELTA.dtd">
3  <plan-library>
4    <library-info title="NewPlanLibrary">
5      <administrative-data original-author="AsbruViewNew"/>
6    </library-info>
7    <domain-defs>
8      <domain name="mainDomain">
9        <qualitative-scale-def name="Boolean">
10         <qualitative-entry entry="false" />
11         <qualitative-entry entry="true" />
12       </qualitative-scale-def>
13       <parameter-group>
14         <parameter-def name="cond1" required="no" type="Boolean">
15           <raw-data-def mode="manual" use-as-context="no" />
16         </parameter-def>
17         <parameter-def name="cond2" required="no" type="Boolean">
18           <raw-data-def mode="manual" use-as-context="no" />
19         </parameter-def>
20       </parameter-group>
21     </domain>
22   </domain-defs>
23   <plans>
24     <plan-group>
25
26       <!-- simulates an unstructured Block with AND-split and
27       OR-join behaviour-->
28       <plan name="Sequence" title="Sequence">
29         <plan-body>
30           <subplans type="sequentially">
31             <wait-for>
32               <all/>
33             </wait-for>
34             <plan-activation>
35               <plan-schema name="Plan_Block"/>
36             </plan-activation>
37
38           </subplans>
39         </plan-body>
40       </plan>
41
42       <plan name="Plan_Block" title="Plan_Block">
43         <plan-body>
44           <subplans type="unordered" wait-for-optional-subplans="yes">
45             <wait-for><one/></wait-for>
46             <plan-activation>
```

```
                <plan-schema name="Plan_A"/>
              </plan-activation>
              <plan-activation>
                <plan-schema name="Plan_B"/>
              </plan-activation>
              <plan-activation>
                <plan-schema name="Plan_C"/>
              </plan-activation>

            </subplans>
          </plan-body>
        </plan>

        <plan name="Plan_A" title="Plan A">
          <plan-body>
            <user-performed />
          </plan-body>
        </plan>

        <plan name="Plan_B" title="Plan B">
          <plan-body>
            <user-performed />
          </plan-body>
        </plan>

        <plan name="Plan_C" title="Plan C">
          <conditions>
            <filter-precondition>
              <constraint-combination type="or">
                <plan-state-constraint state="completed">
                  <plan-pointer >
                    <static-plan-pointer plan-name="Plan_A" />
                  </plan-pointer>
                  <time-annotation>
                    <now />
                  </time-annotation>
                </plan-state-constraint>
                <plan-state-constraint state="completed">
                  <plan-pointer >
                    <static-plan-pointer plan-name="Plan_B" />
                  </plan-pointer>
                  <time-annotation>
                    <now />
                  </time-annotation>
                </plan-state-constraint>
              </constraint-combination>
            </filter-precondition>
          </conditions>
          <plan-body>
            <user-performed />
          </plan-body>
        </plan>

```

```
100         </plan-group>
101       </plans>
102     </plan-library>
```

1.1. Event-Condition-Action-Rules Strategie

Anbei finden sie das vollständige Asbru-Code Beispiel zu Beispiel 2a aus Abschnitt 4.3.4

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plan-library SYSTEM "Asbru_7.3h_DELTA.dtd">
<plan-library>
  <library-info title="NewPlanLibrary">
    <administrative-data original-author="AsbruViewNew"/>
  </library-info>
  <domain-defs>
    <domain name="mainDomain">
      <qualitative-scale-def name="Boolean">
        <qualitative-entry entry="false" />
        <qualitative-entry entry="true" />
      </qualitative-scale-def>
      <parameter-group>
        <parameter-def name="dummy" required="no" type="Boolean">
          <raw-data-def mode="manual" use-as-context="no" />
        </parameter-def>
      </parameter-group>
    </domain>
  </domain-defs>

  <library-defs>
    <qualitative-scale-def name="Boolean">
      <qualitative-entry entry="false" />
      <qualitative-entry entry="true" />
    </qualitative-scale-def>

    <variable-def  name="start_comp"  >
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="true"/>
        </initial-value>
      </scalar-def>
    </variable-def>
    <variable-def  name="a_comp"  >
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="false"/>
        </initial-value>
      </scalar-def>
    </variable-def>
    <variable-def  name="b_comp"  >
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="false"/>
        </initial-value>
      </scalar-def>
    </variable-def>
    <variable-def  name="c_comp"  >
```

```
      <scalar-def type="Boolean">
        <initial-value>
          <qualitative-constant value="true"/>
        </initial-value>
      </scalar-def>
    </variable-def>

  </library-defs>
  <plans>
    <plan-group>
      <plan name="root" title="root">
        <plan-body>
          <iterative-plan>
            <do-repeatedly>
              <plan-activation>
                  <plan-schema name="block"/>
                </plan-activation>
            </do-repeatedly>
            <termination-condition>
              <comparison type="equal">
                <left-hand-side>
                  <variable-ref name="c_comp"/>
                </left-hand-side>
                <right-hand-side>
                  <qualitative-constant value="true"/>
                </right-hand-side>
              </comparison>
            </termination-condition>
          </iterative-plan>
        </plan-body>
      </plan>

      <plan name="block" title="block">
        <plan-body>
          <subplans type="unordered" wait-for-optional-subplans="yes" >
            <wait-for>
              <one/>
            </wait-for>
            <plan-activation>
              <plan-schema name="Plan_A"/>
            </plan-activation>
            <plan-activation>
              <plan-schema name="Plan_B"/>
            </plan-activation>
            <plan-activation>
              <plan-schema name="Plan_C"/>
            </plan-activation>
          </subplans>
        </plan-body>
      </plan>

      <plan name="Plan_A" title="Plan A">
        <conditions>
          <filter-precondition>
```

```
            <simple-condition>
              <comparison type="equal">
                <left-hand-side>
                  <variable-ref name="start_comp"/>
                </left-hand-side>
                <right-hand-side>
                  <qualitative-constant value="true"/>
                </right-hand-side>
              </comparison>
            </simple-condition>
          </filter-precondition>
        </conditions>
        <plan-body>
          <subplans type="sequentially">
            <wait-for>
              <all/>
            </wait-for>
            <user-performed />
            <variable-assignment variable="start_comp">
              <qualitative-constant value="false"/>
            </variable-assignment>
            <variable-assignment variable="a_comp">
              <qualitative-constant value="true"/>
            </variable-assignment>
          </subplans>
        </plan-body>
      </plan>

      <plan name="Plan_B" title="Plan B">
        <conditions>
          <filter-precondition>
            <simple-condition>
              <comparison type="equal">
                <left-hand-side>
                  <variable-ref name="start_comp"/>
                </left-hand-side>
                <right-hand-side>
                  <qualitative-constant value="true"/>
                </right-hand-side>
              </comparison>
            </simple-condition>
          </filter-precondition>
        </conditions>
        <plan-body>
          <subplans type="sequentially">
            <wait-for>
              <all/>
            </wait-for>
            <user-performed />
            <variable-assignment variable="start_comp">
              <qualitative-constant value="false"/>
            </variable-assignment>
            <variable-assignment variable="b_comp">
              <qualitative-constant value="true"/>
```

```
157                 </variable-assignment>
158               </subplans>
159             </plan-body>
160           </plan>
161
162
163     <plan name="Plan_C" title="Plan C">
164           <conditions>
165             <filter-precondition>
166               <constraint-combination type="or">
167                 <simple-condition>
168                   <comparison type="equal">
169                     <left-hand-side>
170                       <variable-ref name="a_comp"/>
171                     </left-hand-side>
172                     <right-hand-side>
173                       <qualitative-constant value="true"/>
174                     </right-hand-side>
175                   </comparison>
176                 </simple-condition>
177                 <simple-condition>
178                   <comparison type="equal">
179                     <left-hand-side>
180                       <variable-ref name="b_comp"/>
181                     </left-hand-side>
182                     <right-hand-side>
183                       <qualitative-constant value="true"/>
184                     </right-hand-side>
185                   </comparison>
186                 </simple-condition>
187               </constraint-combination>
188             </filter-precondition>
189           </conditions>
190           <plan-body>
191             <subplans type="sequentially">
192               <wait-for>
193                 <all/>
194               </wait-for>
195               <user-performed />
196               <variable-assignment variable="a_comp">
197                 <qualitative-constant value="false"/>
198               </variable-assignment>
199               <variable-assignment variable="b_comp">
200                 <qualitative-constant value="false"/>
201               </variable-assignment>
202               <variable-assignment variable="c_comp">
203                 <qualitative-constant value="true"/>
204               </variable-assignment>
205             </subplans>
206           </plan-body>
207         </plan>
208       </plan-group>
209     </plans>
210   </plan-library>
```