



Evaluierung der Konsistenz zwischen Business Process Modellen und Business Role-Object Spezifikation

Lars Westermann

Bachelorarbeit

Eingereicht am: 29.08.2019

Inhaltsverzeichnis

1. Einleitung	4
1.1. Motivation	4
1.2. Problemdefinition	4
1.3. Struktur der Arbeit	4
2. Hintergrund	6
2.1. Business Process Model and Notation	6
2.2. Compartment Role Object Model	8
2.3. Business Role-Object Specification	9
3. Verwandte Arbeiten	11
3.1. Klassifikation von Verfahren zur Konsistenzprüfung	11
3.2. Aktuelle Verfahren zur Konsistenzprüfung	12
3.3. Vergleich der bestehenden Verfahren	13
4. Konsistenz zwischen BPMN und BROS	15
4.1. Konsistenzproblem	15
4.2. Konsistenzregeln	16
4.3. Referenzarchitektur	19
5. Implementierung der automatischen Konsistenzprüfung	21
5.1. Implementierung der Referenzarchitektur	21
5.2. Matching der Modelemente	23
5.3. Implementierung der Konsistenzregeln	24
5.4. Benutzerinterface	26
6. Fallstudie	30
6.1. Anwendung am Beispiel einer Pizzabestellung	30
6.2. Erweiterbarkeit des Ansatzes	33
7. Schluss	35
7.1. Zusammenfassung	35
7.2. Wissenschaftlicher Beitrag	35
7.3. Zukünftige Arbeiten	35
A. Appendix	39

Abstract

TODO

Die heutigen Methoden zur Erstellung von Software hängen stark von geeigneten definierten Modellen ab, um die Struktur und das Verhalten der Software zu spezifizieren. Die strukturbasierten Modelle müssen an den verhaltensbasierten Modellen ausgerichtet sein, damit das anschließend entwickelte Softwaresystem auch die in den Vorgehensmodellen definierten Geschäftsprozesse umsetzt. Derzeit gibt es keine systematische Möglichkeit, prozessuale Geschäftsprozesse (z.B. in Form von BPMN-Prozessen) in strukturbasierte Modellen der Software zu spezifizieren, um eine solche Konsistenz sicherzustellen. Als erster Ansatz wird dieses Problem in der Sprache der Business Role-Object Specification (BROS) gelöst, indem zeitliche Elemente in eine statische strukturbasierte Modellspezifikation eingefügt werden. Es ist jedoch eine manuelle, komplexe und fehleranfällige Aufgabe, die Konsistenz von BROS mit einer bestimmten prozeduralen Geschäftsprozessen sicherzustellen und zu überprüfen.

In dieser Arbeit wird die Konsistenz zwischen BROS und der prozeduralen BPMN untersucht. Zu diesem Zweck werden die Modellelemente in einem BROS- und einem BPMN-Modell miteinander verglichen, um etwaige Abweichungen in Bezug auf mehrere Konsistenzkonzepte, sogenannte Konsistenzbeschränkungen, zu ermitteln. Basierend auf dieser Analyse werden dem Modellierer Warnungen gegeben, wenn Konsistenzbeschränkungen verletzt werden und wie die Probleme möglicherweise gelöst werden können. Diese Aufgabe wird automatisch über ein Tool ausgeführt werden. Die Proof-of-concept Implementierung nutzt dazu die Modelle des BROS-Editor FRaMED.io und des BPMN-Editor bpmn.io.

1. Einleitung

1.1. Motivation

Die heutigen Methoden zur Erstellung von Software hängen stark von geeigneten definierten Modellen ab, um die Struktur und das Verhalten der Software zu spezifizieren. Einerseits werden zur Strukturdefinition häufig UML-Strukturdiagramme verwendet, wie z.B. Klassendiagramme oder Komponentendiagramme. Andererseits werden prozedurale Modelle verwendet, um das Verhalten der Software darzustellen, z.B. BPMN-Diagramme, Sequenzdiagramme oder Petrinetze. Dennoch besteht eine Lücke zwischen diesen beiden Modellperspektiven: Während die Geschäftsprozesse in prozeduralen Modellen modelliert werden, kann die eigentliche Implementierung der Software nicht ohne die strukturbasierte, Modelle erfolgen. Die strukturbasierten Modelle müssen daher an den verhaltensbasierten Modellen ausgerichtet sein, damit das anschließend entwickelte Softwaresystem auch die in den Vorgehensmodellen definierten Geschäftsprozesse umsetzt.

Derzeit gibt es keine systematische Möglichkeit, prozessuale Geschäftsprozesse, in Form von BPMN-Prozessen, in strukturbasierte, Modellen der Software zu spezifizieren, um eine solche Konsistenz sicherzustellen. Als erster Ansatz wird dieses Problem in der Sprache der Business Role-Object Specification (BROS) gelöst, indem zeitliche Elemente in eine statische strukturbasierte Modellspezifikation eingefügt werden. Es ist jedoch eine manuelle, komplexe und fehleranfällige Aufgabe, die Konsistenz von BROS mit einer bestimmten prozeduralen Geschäftsprozessen sicherzustellen und zu überprüfen. Daher wird in dieser Arbeit die Konsistenz zwischen BROS und der prozeduralen BPMN untersucht.

1.2. Problemdefinition

- Welche Konsistenzregeln gelten zwischen BPMN- und BROS-Modellen?
- In wie weit ist eine automatische Konsistenzprüfung möglich?
- Wie sieht eine mögliche Implementierung des automatisierten Konsistenzvergleiches aus?

1.3. Struktur der Arbeit

Zu diesem Zweck werden in Kapitel 2 die Modellelemente von BPMN und BROS vorgestellt und mit Metamodellen beschrieben. Um einen Einblick in das Themengebiet des Konsistenzvergleiches zu bekommen werden im Kapitel 3 aktuelle Konsistenzprüfungsverfahren auf für UML-Modelle klassifiziert. Da BROS eine neue Modellierungssprache ist, existieren noch keine Konsistenzbeschränkungen zu anderen Modellen. Daher werden in Kapitel 4 Regeln aufgestellt um

die Konsistenz zwischen BPMN- und BROS-Modellen zu gewährleisten. Für die automatisierte Überprüfung der Konsistenzregeln wird eine Referenzarchitektur erstellt und in Kapitel 5 implementiert. Die Implementierung nutzt als Modellquellen die Editoren bpmn.io und FRaMED.io. Anschließend wird sie in Kapitel 6 mit einem Beispiel Use-Case evaluiert. Zusätzlich wird die Erweiterbarkeit der Implementierung um ein neue Regeln demonstriert. In Kapitel 7 wird schließlich der Erfolg der Arbeit erörtert.

2. Hintergrund

Um in der weiteren Arbeit den Konsistenzvergleich zwischen BPMN- und BROS-Modellen durchführen zu können, werden zunächst die Modelle und ihre unterstützten Elemente vorgestellt. Dafür wird für jedes Modell ein Metamodell eingeführt. Da sich einige Namen zwischen den Modellen doppeln werden die Elemente der Metamodelle mit dem Präfix "Bpmn" bzw. "Bros" benannt.

2.1. Business Process Model and Notation

Die *Business Process Model and Notation* (BPMN) ist der Standard für die grafische Beschreibung von Geschäftsprozessen. Dabei wird das Verhalten eines Systems mit einer an Flussdiagrammen angelehnten Form beschrieben. Einer der Hauptvorteile von BPMN ist die große Verbreitung. Dank ihr können Prozessbeschreibungen auf einfache Art und Weise zwischen verschiedenen Plattformen und Stakeholdern ausgetauscht werden. Neben der informellen Beschreibung von Prozessen unterstützt BPMN auch die direkte Ausführung von Geschäftsprozessen. Mittels diverser Interpreter können in BPMN modellierte Abläufe zur Automatisierung einfacher Aufgaben oder ganzer Fertigungsprozesse genutzt werden. Die Hauptelemente von einem BPMN-Modell sind:

- **Activity:** Eine Activity beschreibt eine Tätigkeit die innerhalb des Geschäftsprozesses ausgeführt wird. Da eine Aufgabe ausgeführt wird, kann die Ausführung einer Activity längere Zeit in Anspruch nehmen. Zur Darstellung wird ein Rechteck mit abgerundeten Ecken verwendet.
- **Gateway:** Ein Gateway wird für die Steuerung des Kontrollflusses verwendet. An einem Gateway können verschiedene Kontrollwege zusammenlaufen oder sich teilen. Dabei werden verschiedene Arten, wie zB. AND- und OR-Gateways unterstützt. Dargestellt wird ein Gateway mittels einem um 45 Grad gedrehtem Quadrat.
- **Event:** Ein Event beschreibt Ereignis das innerhalb des Geschäftsprozesses auftreten kann und werden mit einem Kreis dargestellt. Sie beeinflussen den Kontrollfluss, können ihn starten, pausieren oder auch beenden. Events werden in drei verschiedenen Dimensionen eingeteilt, nach ihrer Position, nach ihrer Wirkung und nach ihrer Art. Für die weitere Arbeit ist nur die Unterteilung nach ihrer Position innerhalb des Geschäftsprozesses wichtig. Dabei wird zwischen einem StartEvent (einfacher Rahmen), einem IntermediateEvent (doppelter Rahmen) und einem EndEvent (dicker Rahmen) unterschieden.
- **Flow:** Ein Flow ist eine gerichtete Verbindung zwischen anderen Modellelementen. Dabei wird zwischen SequenceFlows und MessageFlows unterschieden. Ein SequenceFlow stellt den Kontrollfluss dar und verbindet Elemente um eine Ausführungsreihenfolge festzulegen.

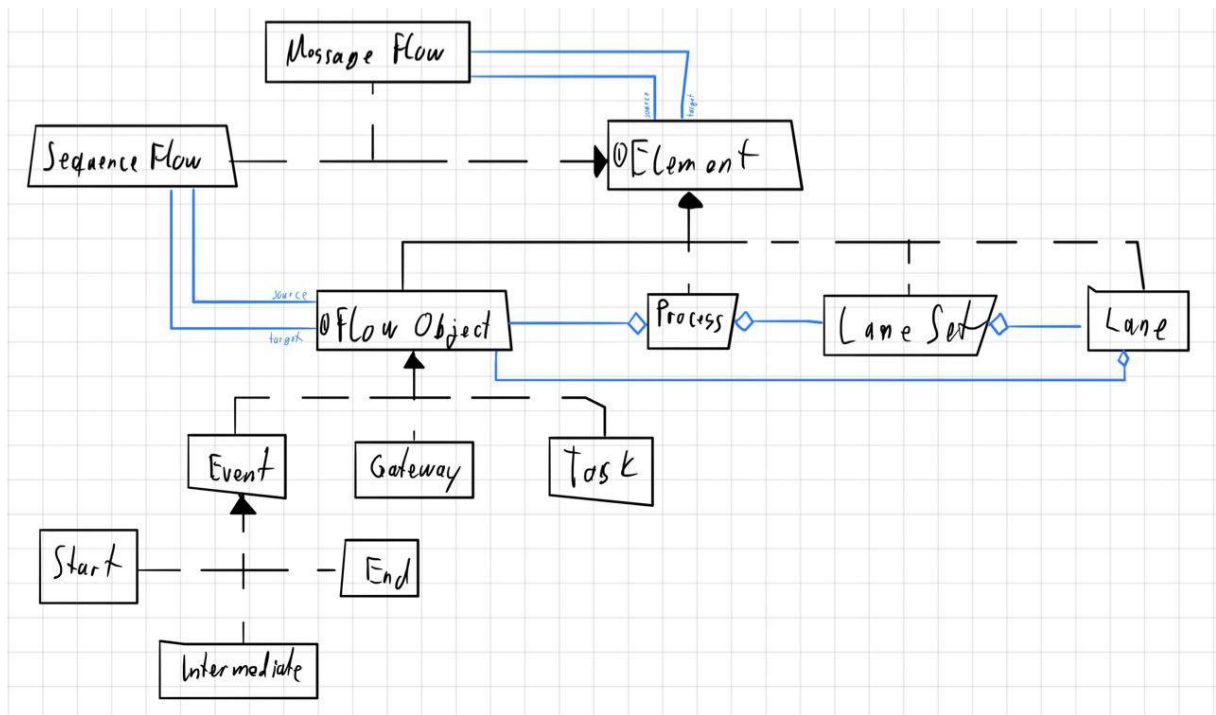


Abbildung 2.1.: BPMN Metamodell

Ein MessageFlow verbindet unterschiedliche Teilnehmer des Geschäftsprozesses und stellt die austausch von Mitteilungen dar.

- **Pool:** Ein Pool stellt eine Gruppe von zusammengehörenden Teilnehmern innerhalb eines Geschäftsprozesses dar. Dargestellt wird ein Pool mit einem Rechteck wobei der Name am linken Rand steht.
- **Swimlane:** Eine Swimlane ist ein einzelner Teilnehmer der zu einem Pool gehört und Aufgaben aus dem Geschäftsprozess erfüllt. Bei einem Pool der nur aus einer Swimlane besteht kann die Swimlane mit dem Pool vereinigt werden. Innerhalb eines Pools wird die Simelane als eine horizontale Bahn dargestellt.
- **Process:** Der Process bildet den vollständigen Geschäftsprozess ab und ist das Container-element für die anderen Modellelemente.

Da BPMN schon längere Zeit in Benutzung ist hat es eine gute Toolunterstützung für die Modellierung. Eines dieser Tools ist bpmn.io¹. Es ist ein webbasiertes Tool das einen großteil des BPMN-Standards unterstützt. Das Tool speichert und lädt die BPMN-Modelle in einem auf XML-basierenden Format mit der Dateiendung *.bpmn*. Die Datenstruktur basiert auf dem Metamodell das von der *Object Management Group*² spezifiziert wurde.

Für die weitere Arbeit wird nur ein Teil des vollständige BPMN-Standards der *OMG* genutzt. Ein deutlich gekürztes Metamodell für Business Prozesse wurde von Loja et al. 2010 erstellt. Das hier verwendete Metamodell basiert auf den Metamodellen der *OMG* und von Loja et al. 2010 (vgl. Abbildung 2.1). Es unterstützt dabei nur die oben genannten BPMN-Elemente. Weitere BPMN-Elemente werden in dieser Arbeit und dem später entwickeltem Tool nicht betrachtet.

Mit dem BPMN-Metamodell aus Abbildung 2.1 wird ein BPMN-Modell in einem Graphen, genauer einem Baum mit Querverweisen, dargestellt. Die Wurzel des Baumes ist das Modell

¹<https://demo.bpmn.io/>

²<https://www.omg.org/spec/BPMN/2.0/About-BPMN/>

ansich. Dieses kann mehrere Prozesse (BpmnProcess) enthalten. Jeder Prozess beinhalten verschiedene Flowobjekte, dazu zählen Events (BpmnEvent), Gateways (BpmnGateway) und Activities (BpmnTask). Die Events teilen sich in drei Unterklassen auf, die Start-, Intermediate- und EndEvents (BpmnStartEvent, BpmnIntermediateEvent, BpmnEndEvent). Außerdem kann ein Prozess mehrere Pools (BpmnLaneSet) und Swimlanes (BpmnLane) enthalten. Hierbei ist zu beachten, das obwohl eine Swimlane auch Flowobjekte enthalten kann, das jedes Flowobjekt nur einen Container haben darf. So kann ein Flowobjekt nicht gleichzeitig Kind eines Prozesses und einer Swimlane seien. Jedes Kind einer Swimlane ist transitiv auch Kind von dem Elternprozess der Swimlane. Die beiden Flow-Arten bilden die Querverweise innerhalb des Baumes. Ein Message-Flow (BpmnMessageFlow) kann zwischen allen BPMN-Elementen gezeichnet werden, wird aber hauptsächlich nur für die Kommunikation zwischen Prozessen verwendet. Der SequenceFlow (BpmnSequenceFlow) stellt den Ablauf innerhalb eines Prozesses dar und darf nur zwischen Flowobjekten existieren.

2.2. Compartment Role Object Model

Das *Compartment Role Object Model* (**CROM**) ist eine Modellierungssprache für Rollenbasierte Systeme. Eine Rolle beschreibt einen Aufgabenbereich der von verschiedenen Entitäten übernommen werden kann, man sagt die Entität spielt die Rolle. CROM führt zusätzlich noch das *Compartment* ein, das den Kontext der Rolle abbildet, in dem diese gespielt werden kann. Dazu wird das Modell in drei logische Aspekte unterteilt, den Verhaltensaspekt, den Relationenaspekt und den Kontextaspekt. Der Verhaltensaspekt beschreibt die Aktoren bzw Entitäten und die Rollen die von diesen gespielt werden können. Der Relationenaspekt fügt zu den Rollen weitere Constraints und Verbindungsbeschreibungen hinzu. Im dritten Aspekt, dem Kontextaspekt wird mittels der Compartments die Kontextabhängigkeit modelliert. Die dafür genutzten Elemente sind:

- **RoleType:** Eine RoleType ist die Darstellung einer Rolle. Sie wird mit einem abgerundeten Rechteck vergleichbar mit einer UML-Klasse dargestellt. Eine Rolle kann einer Entität zusätzliche Attribute und Methoden hinzufügen.
- **CompartmentType:** In einem CompartmentType bildet den Kontext von verschiedene RoleTypes ab. Das Compartment ist an sich auch eine Rolle und kann von Entitäten gespielt werden. Graphisch wird es wie eine UML-Klasse dargestellt, die zusätzlich noch ein weiteres Feld für die beinhalteten Rollen besitzt.
- **NaturalType:** Ein NaturalType oder auch DataType stellt eine Entität bzw Akteur des Modelles dar. Sie unterscheiden sich nur in der Semantik. Der NaturalType bildet eine natürliche Person ab, wohingegen der DataType einen künstlichen Teilnehmer beschreibt. Dabei werden beide Arten mit einer UML-Klasse dargestellt.
- **Fulfillment:** Ein Fulfillment ist eine Relation die eine Entität mit einer Rolle oder einem Compartment verbindet. Sie beschreibt welche Rolle von welchen Entitäten gespielt werden kann. Damit ist das Fulfillment immer von einer Entität auf eine Rolle gerichtet.
- **Relationship:** Eine Relationship ist eine Relation zwischen Entitäten oder Rollen die mit einer einfachen Linie ohne Pfeilenden dargestellt wird. Sie kann je nach Annotierung ein Constraint oder auch eine Verbindung zwischen diesen beschrieben.
- **Package:** Ein Package hilft bei der Strukturierung von großen Modellen. In ihnen kann ein Submodell abgebildet werden, in das nur Relationen hinein, aber nicht hinaus führen dürfen.

CROM besitzt mit dem Eclipse Plugin *FRaMED-2.0*³ einen eigenen graphischen Editorsupport. Der Name des Editors ist ein Akronym für *Full-fledged Role Modeling EDitor* und wurde speziell für CROM entwickelt.

2.3. Business Role-Object Specification

Der neu entwickelte Ansatz der *Business Role-Object Specification* (**BROS**) kombiniert die Vorteile der strukturbasierten Modellierung und der verhaltensbasierten Modellierung. Als Grundlage für BROS dient die strukturbasierte Modellierungssprache CROM. Diese wird mit Hilfe von Events um den Verhaltensaspekt erweitert. Zusätzlich zu den CROM-Elemente werden folgende Modellelemente unterstützt.

- **Scene:** Eine Scene stellt einen Teilnehmer innerhalb eines Prozesses dar. Sie besitzt Methoden und kann Rollen, Events und andere Szenen beinhalten. Dargestellt wird sie mit einem Rechteck mit doppeltem linkem Rahmen.
- **Event:** Ein Event beschreibt ein Ereignis innerhalb des Geschäftsprozesses. Mit einem Event kann eine Rolle erzeugt bzw. beendet werden. Dabei kann ein Event eine beliebige Abstraktion eines Prozesses sein.
- **Create-/DestroyRelation:** Eine Create- bzw. DestroyRelation verbindet ein Event mit einer Scene oder Rolle. Sie beschreibt welches Event für das erstellen oder das auflösen einer Rolle im Geschäftsprozess verantwortlich ist. Eine CreateRelation ist von einem Event zu einer Rolle/Scene gerichtet, eine DestroyRelation verläuft in die gegensätzliche Richtung.
- **ReturnEvent:** Ein ReturnEvent ist ein Event, welches die aktuelle Scene beendet. Es wird als ein Event mit doppeltem Rahmen auf dem Rand der Scene dargestellt. Das ReturnEvent ist nicht mit einer DestroyRelation verbunden, da es implizit im gesamten Verlauf der Scene auftreten kann.

Auch BROS besitzt mit *FRaMED.io*⁴ Editorsupport. *FRaMED.io* ist eine Neuentwicklung von *FRaMED-2.0* und wie *bpmn.io* webbasiert. Damit ist es Plattformübergreifend, ohne aufwendige Installation nutzbar.

TODO

Das Metamodell in Abbildung 2.2 basiert auf dem von Schön et al. o.D. und dem in *FRaMED.io* implementierten Metamodell. Die Elemente des Metamodelles teilen sich in zwei Klassen auf. Auf der einen Seite sind die Connections, zu denen die Create- und DestroyRelation (*BrosCreateRelation*, *BrosDestroyRelation*) gehören. Auf der anderen Seite sind die Objekte die sich in Endknoten und Objektgruppen teilen. Endknoten sind die *NaturalTypes* (*BrosClass*), die *RoleTypes* (*BrosRoleType*), *Events* (*BrosEvent*) und *ReturnEvents* (*BrosReturnEvent*). Die Objektgruppen sind Containerelemente die andere Objekte enthalten können. Zu ihnen zählen der *CompartmentType* (*BrosCompartment*), das *Package* (*BrosPackage*) und die *Scene* (*BrosScene*). Auch dieses Metamodell entspricht einem Graphen, genauer einem Baum mit Querverweisen. Jedes Objekt darf nur Kind einer Objektgruppen sein und die Eltern-Kind-Beziehung ist transitiv. Die Connections bilden die Querverweise innerhalb des Baumes.

³<https://github.com/Eden-06/FRaMED-2.0>

⁴<https://eden-06.github.io/FRaMED-io/>

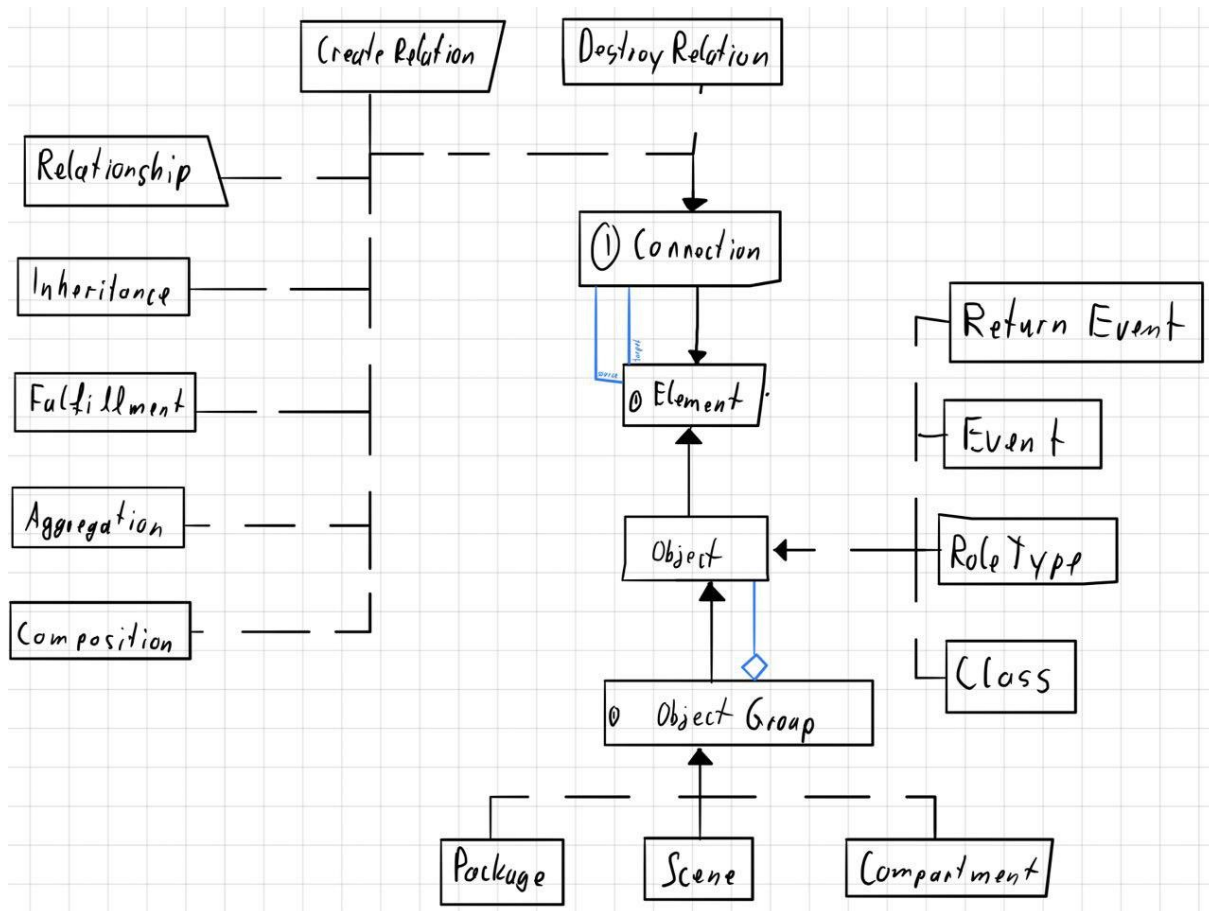


Abbildung 2.2.: BROS Metamodell

3. Verwandte Arbeiten

Der Bereich der Konsistenzprüfung zwischen strukturbasierten und verhaltensbasierten Modellen ist seit Jahren gut erforscht. Insbesondere gibt es ein großes Spektrum an Methoden zum Vergleichen von verschiedenen UML-Modellen. Viele dieser Methoden untersuchen die Konsistenz zwischen UML-Klassendiagrammen und UML-Verhaltensmodellen wie UML-Zustands- oder UML-Aktivitätsdiagramme. Da sich die Konzepte dieser UML-Modelle mit denen von BROS und BPMN ähneln, lassen sich diese Methoden auch auf BPMN- und BROS-Modelle anwenden. Dafür wird zunächst eine Klassifikation von Verfahren zur Konsistenzprüfung vorgestellt. Anhand dieses Schemas werden anschließend relevante Arbeiten erläutert und verglichen.

3.1. Klassifikation von Verfahren zur Konsistenzprüfung

Da es bereits etliche Arbeiten im Gebiet der Konsistenzprüfung von UML-Modellen gibt, wurde bereits eine Zusammenstellung der bestehenden Methoden unter anderem von Usman et al. 2008 und Lucas et al. 2009 durchgeführt. Um die unterschiedlichen Methoden besser klassifizieren zu können nutzen beide Arbeiten ein ähnliches Klassifikationsschema:

- **Nature (Usman et al. 2008):** Es beschreibt den Fokus der vergleichbaren Modelltypen. Dabei wird zwischen strukturbasierten und verhaltensbasierten Modellen unterschieden. Strukturbasierte Modelle beschreiben den Aufbau eines Systems. Dazu zählen unter anderem UML-Klassen-, UML-Komponentendiagramm und BROS-Modelle. Zu den verhaltensbasierten Modellen gehören beispielsweise UML-Zustands-, UML-Aktivitätsdiagramme und BPMN-Modelle. Diese Modelle verdeutlichen die Abläufe und Zustände innerhalb eines Systems. Mögliche Werte sind strukturbasiert, verhaltensbasiert und beides.
- **Diagrams (Usman et al. 2008):** Es beschreibt welche konkreten Modelle von der Methode unterstützt werden. Die Arbeiten von Usman et al. 2008 und Lucas et al. 2009 beziehen sich dabei ausschließlich auf UML Diagramme. Mögliche Werte sind die Modellnamen.
- **Consistency Type (Usman et al. 2008):** Es beschreibt welche Arten der Konsistenz von der Methode überprüft werden. Dabei wird unterschieden zwischen *Inter-Modell (vertikale) Konsistenz* (Konsistenz bei verschiedenen Abstraktionsstufen und gleichem Modelltyp), *Intra-Modell (horizontale) Konsistenz* (Konsistenz bei gleicher Abstraktionsstufe und verschiedenen Modelltypen) und *Evolutionkonsistenz* (Konsistenz der eines Modelles über verschiedene Entwicklungsstufen). Zusätzlich spezifiziert Usman et al. 2008 noch die *semantische-* und die *syntaktische Konsistenz*. Diese beziehen sich auf die Konsistenz eines Modelles zu seinem Metamodell. Dies wird für die nachfolgende Arbeit als Voraussetzung angesehen und nicht näher betrachtet.

- **Consistency Strategy (Usman et al. 2008):** Es beschreibt die benutzte Validierungsstrategie. Usman et al. 2008 nennt drei verschiedene Strategien die bei der Durchführung des Konsistenzvergleiches genutzt werden können. Diese sind: *Analysis* (Auf einem Algorithmus basierend), *Monitoring* (Auf einem Regelsatz basierend) und *Construction* (Auf der Generierung des zu vergleichenden Modelles basierend).
- **Intermediate Representation (Usman et al. 2008):** Es beschreibt ob die Methode eine temporäre Zwischendarstellung benötigt oder nicht. Mögliche Werte sind ja und nein.
- **Case Study (Usman et al. 2008):** Es beschreibt ob die Methode an einem Beispiel evaluiert wurde. Mögliche Werte sind ja und nein.
- **Automatable (Usman et al. 2008):** Es beschreibt ob die Methode manuell oder automatisiert von einem Programm durchgeführt werden kann. Mögliche Werte sind geringer Aufwand (H), mittel Aufwand (M) und hoher Aufwand (L).
- **Tool Support (Usman et al. 2008):** Es beschreibt ob die Methode von einem Tool unterstützt wird oder ein eigenes Tool entwickelt wurde. Mögliche Werte sind ja und nein.
- **Model Extensibility:** Es beschreibt wie gut die Methode um weitere Modelle für den Konsistenzvergleich erweiterbar ist. Mögliche Werte sind geringer Aufwand (H), mittel Aufwand (M) und hoher Aufwand (L).
- **Rule Extensibility:** Es beschreibt wie gut die Methode um weitere Konsistenzregeln erweiterbar ist. Mögliche Werte sind geringer Aufwand (H), mittel Aufwand (M) und hoher Aufwand (L).

Dieses Schema ist direkt auf die Konsistenzprüfung von BPMN- und BROS-Modelle anwendbar. Wichtig für die weitere Arbeit sind Verfahren deren *Nature* beide Modelltypen unterstützt und deren *Consistency Type* auf der *Intra-Modell (horizontale) Konsistenz* beruht. Der Klassifikationspunkt der Erweiterbarkeit wurde von Lucas et al. 2009 eingeführt. Für diese Arbeit wird er aber in die Aspekte der Modelerweiterbarkeit und der Regelerweiterbarkeit aufgeteilt um eine genauere Einteilung der Verfahren zu ermöglichen.

3.2. Aktuelle Verfahren zur Konsistenzprüfung

Anhand des aufgestellten Klassifikationsschemas werden in diesem Abschnitt verschiedene Verfahren der Konsistenzprüfung von UML-Modellen vorgestellt. Dabei werden nur Verfahren betrachtet die in ihrer *Nature* beide Modelltypen unterstützten. Der Klassifikationspunkt *Nature* wird daher nicht weiter betrachtet.

Transformation zu CSP-OZ: Rasch et al. 2003 transformiert Klassen- und Zustandsdiagramme nach *CSP-OZ* (Communicating Sequential Processes - Object-Z) als Zwischendarstellung. *CSP* ist eine Prozessalgebra für die Beschreibung der Zusammenarbeit verschiedener Systeme. *OZ* ist eine objektorientierte Erweiterung der Z-Notation. Diese wird zur formalen Beschreibung von Systemen genutzt, indem Variablen an Bedingungen geknüpft werden. Anhand eines Regelsatzes wird die Konsistenz der Zwischendarstellungen geprüft. Kim et al. 2004 nutzt ein ähnliches Verfahren, spezialisiert sich dabei aber nur auf Zustandsdiagramme.

Transformation zu Pertinetze: Shinkawa 2006 zeigt das verhaltensbasierte UML-Modelle in *CPN* (Coloured Petri Net) überführt werden können. Mittels mehrerer Beispiele werden verschiedene Transformationsregeln gezeigt. Die formale *Inter-Modell (vertikale) Konsistenz* der CPNs wird nur theoretisch behandelt. Bernardi et al. (Usman et al. 2008, 13) nutzt GSPN (Generalized stochastic Petri nets) anstelle von CPN. Diese eignen sich besonders für Event-basierte Systeme.

	Diagrams	Consistency Type	Consistency Strategy	Intermediate Representation	Case Study	Automatable	Tool Support	Model Extensibility	Rule Extensibility
Rasch 2003	CD, SM	Intra	Monitoring	CSP/OZ	1	H	0	H	M
Shinkawa 2006	UCD, CD, SD, AD, SC	Inter	Analysis	CPN	0	H	0	M	L
Mens 2005	CD, SD, SC	All	Monitoring	Extended UML	1	H	1	H	M
Egyed 2001	CD, OD, SD	Intra, Inter	Construction		0	H	part	M	M
Egyed 2006	CD, SD, SC	Intra	Monitoring		1	H	1	L	M

Tabelle 3.1.: Vergleich der bestehenden Verfahren

Anwendung von *Description logic*: Mens et al. 2005 nutzt *Description logic* für die formale Konsistenzprüfung. Dabei werden beide Modelltypen und alle Validierungsstrategien unterstützt. Für die *Evolutionskonsistenz* wird eine Erweiterung des UML-Metamodells erstellt. Zusätzlich führt Mens et al. 2005 eine Fallstudie durch und entwickelt eine Toolunterstützung. Ein ähnlicher Ansatz wird von Simmonds et al. 2004 genutzt. Noch weiter geht Satoh et al. (Usman et al. 2008, 15) indem UML-Klassendiagramme direkt in Logikprogramme übersetzt und ausgeführt werden.

Transformation in ein gemeinsames Modell: Egyed 2001 entwickelt die ViewIntegra Methode für die *Inter-Modell (vertikale)* und *Intra-Modell (horizontale)* Konsistenz. Der Fokus der ViewIntegra Methode liegt in der Skalierbarkeit und Effizienz der Überprüfung bei großen Modellen. Um dies zu erreichen wird eine iterative Umwandlung von verwandten UML-Modellen beschrieben. Dieses iterative Vorgehen reduziert den Entwicklungsaufwand erheblich, da nur noch ein Bruchteil aller benötigten Transformationsalgorithmen entwickelt werden muss. Dies vereinfacht auch die Erweiterbarkeit um neue Modellarten. Die konstruierten Modelle können anschließend direkt mittels *Inter-Modell (vertikale)* Konsistenz überprüft werden. Einige der beschriebenen Transformationsalgorithmen sind bereits implementiert wurden. Aufgrund der Umwandlung wird auch die *Intra-Modell (horizontale)* Konsistenz unterstützt.

Nutzung von UML-Constraints wie *OCL*: Egyed 2006 prüft die Konsistenz zwischen UML-Klassen-, Sequenz- und Zustandsdiagrammen mittels OCL-Regeln. Neben einer Fallstudie wurde auch ein Tool entwickelt um das Verfahren zu testen. Aufgrund von Einschränkungen von OCL, wie das Fehlen des transitiven Abschlusses, ist die Erweiterbarkeit dieses Ansatzes eingeschränkt. Briand et al. 2003 verwendet ebenfalls OCL-Regeln, konzentriert sich dabei aber auf die *Evolutionskonsistenz*.

3.3. Vergleich der bestehenden Verfahren

Ein wichtiger Punkt für die weitere Arbeit ist die Erweiterbarkeit. Wie bereits genannt, gibt es zwei verschiedene Aspekte der Erweiterbarkeit, die Modellerweiterbarkeit und die Regelerweiterbarkeit. Die Modellerweiterbarkeit beschreibt wie leicht sich ein Verfahren um neue Modelltypen erweitern lässt. Mit einer guten Modellerweiterbarkeit können viele verschiedene Modelltypen für die

Konsistenzprüfung unterstützt werden. Im Gegensatz dazu steht die Regelerweiterbarkeit. Sie beschreibt den Aufwand des Hinzufügens neuer Konsistenzaspekte zu den bereits unterstützten Modellarten. Verfahren die eine gute Regelerweiterbarkeit haben, können potentiell genauere Konsistenzprüfungen durchführen. Aus Tabelle 3.1 lässt sich ein Zusammenhang zwischen der Erweiterbarkeit und der Nutzung einer Zwischendarstellung ableiten. So haben die Verfahren die eine Zwischendarstellung nutzen eine höherer Affinität zu der Modellerweiterbarkeit. Dahingegen haben die Verfahren ohne Zwischendarstellung eine bessere Regelerweiterbarkeit.

Anhand der Zwischendarstellung lassen sich die beschreiben Verfahren in zwei Kategorien einteilen. Zum einen in die die eine Zwischendarstellung benötigen und die die keine Zwischendarstellung nutzen. Rasch et al. 2003, Shinkawa 2006 und Mens et al. 2005 beschreiben Verfahren die zunächst eine Zwischendarstellung aufbauen und anschließend die Konsistenz der Zwischendarstellung überprüfen. Andererseits könne die Verfahren von Egyed 2001 und Egyed 2006 direkt auf den Modellen arbeiten. Der Vorteil von einer Zwischendarstellung ist die leichte Erweiterbarkeit um neue Modellarten. Es muss nur eine neue Konvertierungsmethode des Modelles zur Zwischendarstellung entwickelt werden. Anschließend können die gleichen Algorithmen zur Konsistenzprüfung wie bei den bestehenden Modellen verwendet werden. Allerdings ist dies auch eine Beschränkung der Erweiterbarkeit. Eine Zwischendarstellung hat zumeist den Nachteil des Informationsverlustes. Dadurch lassen sich nur schwer neue Regeln für die Konsistenz der Zwischendarstellung hinzufügen. Es kann auch sein das eine Modellart nicht kompatibel zu der Zwischendarstellung ist. Dann ist eine Erweiterung um diese Modellart nicht möglich. Dagegen haben die Verfahren ohne Zwischendarstellung weniger Beschränkungen bei der Erweiterbarkeit von Modellarten. Hier ist das größte Hindernis der Aufwand eine neue Modellart hinzuzufügen. Im schlechtesten Fall müssen beim Hinzufügen einer neuen Modellart N neue Konvertierungsmethoden entwickelt werden. Dabei ist N die Anzahl der bereits unterstützten Modellarten. Egyed 2001 umgeht dieses Problem mittels einer iterativen Transformation. Dies bestätigt die Annahme der Abhängigkeit der Erweiterbarkeit von der Nutzung einer Zwischendarstellung.

Die am meisten benutzte Methode zur Konsistenzprüfung ist das *Monitoring* (vgl. Usman et al. 2008). Das *Monitoring* basiert auf der Konsistenzprüfung mittels eines Regelsatzes. Verfahren die *Monitoring* nutzen, haben eine gute Erweiterbarkeit im bezug auf neue Konsistenzregeln. Dazu zählen die Verfahren von Rasch et al. 2003, Mens et al. 2005 und Egyed 2006. *Monitoring* bietet allerdings keine formale Verifikation, sondern nur eine Überprüfung vergleichbar mit Unittests. Dies wird von dem Verfahren der *Analysis* gelöst. Mittels formaler Verifikation kann die Konsistenz bewiesen werden. Allerdings sind solcher Verfahren nicht sehr flexibel. Bei Shinkawa 2006 kann beispielsweise nur die *Inter-Modell (vertikale) Konsistenz* verifiziert werden. Die dritte Methode, die *Construction*, beschreibt Verfahren deren Hauptaufgabe die Konstruktion anderer Modelle ist. Hierbei ist die Abgrenzung zu Verfahren die auf einer Zwischendarstellung basieren zu beachten. Verfahren die mit einem Transformationsalgorithmus eine Zwischendarstellung konstruieren, die nicht Teil der unterstützten Modellarten ist, zählen nicht dazu. Egyed 2001 nutzt diese Methode und beschreibt die Transformation zwischen verschiedenen UML-Modellen. Die Konsistenzprüfung der Modelle erfolgt mittels *Inter-Modell (vertikale) Konsistenz* und ist unabhängig von dem Konstruktionsverfahren. Da die eigentliche Konsistenzprüfung nicht teil eines Konstruktionsverfahrens ist, kann diese Methode orthogonal zu den anderen beiden Methoden verwendet werden.

4. Konsistenz zwischen BPMN und BROS

Viele der bereits existierenden Verfahren lassen sich anpassen und können mit anderen Modellen genutzt werden. Anstelle eines UML Klassendiagramms kann ein BROS Modell auf der strukturbasierten Seite und ein BPMN Modell anstelle eines UML-Sequenzdiagramms auf der verhaltensbasierten Seite verwendet werden. Allerdings ist bei dem Vergleich von BPMN und BROS zu beachten das Inkonsistenzen keine strikten Fehler, sondern nur Warnungen an den Modellierer sind. Das liegt an der Möglichkeit ein BROS Modell beliebig anzureichern und das Events eine Abstraktion eines beliebigen Prozesses sein können. Um dieses Problem zu lösen wird zunächst eine genaue Problemdefinition gegeben. Anschließend werden verschiedene Regeln zur Konsistenzprüfung zwischen BPMN und BROS vorgestellt und ein Verfahren zur automatischen Prüfung dieser Regeln erläutert.

4.1. Konsistenzproblem

Wie bereits erwähnt, sind heutige Softwaresystem stark von den dazugehörigen Modellen anhängig. Damit ein solches System erfolgreich implementiert werden kann müssen die Modelle konsistent zueinander sein. Sollten schon zu Beginn eines Projektes unbemerkt Inkonsistenzen auftreten kann dies zum Scheitern des ganzen Projektes führen. Die verhaltensbasierte Modelle beschreiben die Interaktion innerhalb eines Systems. Dabei benötigen sie die Funktionalität die von den strukturbasierten Modellen bereitgestellt wird. Wenn sich die benötigte und die bereitgestellte Funktionalität unterscheidet kann das Softwaresystem nicht funktionieren.

Eine formale und allgemeingültige Definition des Konsistenzproblem in der Domain der Modellierung ist in der Literatur nur schwer zu finden. Bevor das Konsistenzproblem definiert werden kann muss zunächst beschrieben werden was eine Inkonsistenz ist. Laut Nuseibeh 1996 tritt eine Inkonsistenz genau dann auf, wenn eine Konsistenzregel verletzt wird. Eine Konsistenzregel ist eine Formalisierung von einem Aspekt der Konsistenz zwischen den betrachteten Modellen. Dies kann in natürlicher Sprach oder mit einem formalen Ansatz beschrieben sein, wobei nur letzteres automatisiert überprüft werden kann. Für die Definition nach Nuseibeh 1996 wird das Vorhandensein eines solchen Regelsatzes vorausgesetzt. Mit Hilfe dieser Definition kann das Konsistenzproblem wie folgt definiert werden:

Definition. *Das Konsistenzproblem beschreibt den Vorgang der Minimierung und Verhinderung von Inkonsistenzen mit Hilfe der Aufstellung und Prüfung von Konsistenzregeln.*

Diese Definition lässt sich leicht auf Konsistenzverfahren anwenden, die auf der Überprüfung von Regeln basieren. Aber auch formale Verfahren ohne einen eigenen Regelsatz lassen sich

mit dieser Definition beschreiben. So können z.B. die Verfahren, die auf der Transformation zu einem Petrinetz basieren mittels der Konsistenzregel "das konstruierte Petrinetz muss lebendig sein" überprüft werden.

4.2. Konsistenzregeln

Das hier vorgestellte Verfahren basiert auf der *Intra-Modell* bzw. horizontalen Konsistenzprüfung ohne Zwischendarstellung. Diese kann in zwei verschiedene Richtungen definiert werden. Eine Regel kann aufgrund eines Merkmals des ersten Modelles ein Merkmal im zweiten Modell fordern oder umgekehrt. Damit basieren diese Regeln auf der logischen Implikation. Um eine reflexive Regel zu erhalten (logische Äquivalenz) müssen zwei Regeln erstellt werden die sowohl die Hin- als auch die Rückrichtung abdecken. Im Bezug auf den Vergleich von BPMN und BROS bedeutet das, eine Regel hat immer ein Ursprungs- und ein Zielmodell. Da BROS beliebig erweitert werden kann, ist kein strikter Vergleich zwischen BPMN und BROS möglich. Aus diesem Grund sind die meisten Regel aus der Sicht des BPMN-Modelles aufgestellt.

Wie bereits in Kapitel 2 erläutert lassen sich die BPMN- und BROS-Modelle, aufgrund ihrer Metamodelle, in Form von Graphen darstellen. Mit Hilfe des Graph-Logik-Isomorphismus können dies Graphen auch in Form von Logikprogrammen ausgedrückt werden. Für die Spezifikation der Regeln wird im nachfolgenden eine auf Prolog basierende Notation genutzt. Dies ermöglicht eine einfache und leicht verständliche Darstellung der Regeln ohne eine eigene domänenspezifische Sprache einführen zu müssen. Als Grundlage für die Konsistenzregeln werden die in Quelltext 1 und Quelltext 2 aufgestellten Fakten und Prädikate genutzt.

Die Faktenbasis besteht aus vier Bestandteilen und bildet die gesamte Modellstruktur ab. Mit dem Fakt *bpmn/2* und *bros/2* werden die jeweiligen Modellelemente aufgelistet und gleichzeitig ihren Typ zugeordnet. Der Fakt *relation/3* bildet eine Relation zwischen zwei Modellelementen von Quelle nach Ziel ab und ordnet dieser Relation gleichzeitig einen Typ zu. Durch *parent/2* wird der hierarchische Aufbau der Modelle abgebildet. Dabei ist zu beachten das jedes Element nur einen Parent haben darf. Es gilt implizit *check_parent(X) :- parent(X, A), parent(X, B) -> A == B.* Damit wird sichergestellt das es sich bei den Graphen für die BPMN- und BROS-Modelle um Bäume handelt. Allgemein ist zu beachten das dies keine Zwischendarstellung ist und nur zur syntaktischen Definition der Regeln genutzt wird. Da der Graph des BPMN-Modelles und der Graph des BROS-Modelles disjunkt sind muss keine weitere Trennung der *relation/3* und *parent/2* Fakten durchgeführt werden.

```
% Definition aller BPMN Elemente mit ihrem zugehörigem Typ.
bpmn(Bpmn, Type).
```

```
% Definition aller BROS Elemente mit ihrem zugehörigem Typ.
bros(Bros, Type).
```

```
% Definition aller Relationen von Quelle nach Ziel mit Typ.
relation(Source, Target, Type).
```

```
% Definition der Modellstruktur per Eltern-Kind Beziehung.
parent(Child, Parent).
```

Quelltext 1: Definitionen der Faktenbasis

Zusätzlich werden noch zwei Hilfsprädikate definiert. Das Prädikat *transitive_parent/2* bildet den transitiven Abschluss der *parent/2* Relation. Jedem Kind werden all seine transitiven Eltern zugeordnet. Dies beinhaltet auch die Abbildung der Identität. Das wichtigste Prädikat ist *match/2*. Es bildet das Matching der BPMN-Elemente zu den BROS-Elementen ab. Dabei wird angenommen das ein Matching bereits existiert. Ein Matching ist die Menge der bidirektionalen Zuordnungen von BPMN-Elementen auf die dazugehörigen BROS-Elemente.

```

% Transitiver Abschluss der Modellstruktur.
transitive_parent(Child, Parent) :-
    Child = Parent.
transitive_parent(Child, Parent) :-
    parent(Child, Parent).
transitive_parent(Child, Parent) :-
    transitive_parent(Child, X), parent(X, Parent).

% Orakel für das Matching von Modellelementen.
match(Bpmn, Bros).

```

Quelltext 2: Definitionen der weiterführenden Regeln

Regel 1: Zu jedem *BPMN-Process* muss eine *BROS-Scene* oder ein *BROS-Event* existieren. Ein BPMN-Process stellt eine Umgebung für einen natürlichen Prozess dar. In BROS wird dies mit einer BROS-Scene repräsentiert. Allerdings kann ein BROS-Modell eine Abstraktion eines BPMN-Modelles sein. In diesen Fall kann der BPMN-Process auch durch ein BROS-Event modelliert werden. Durch diese Abstraktion muss der Inhalt des BPMN-Processes nicht weiter überprüft werden.

```

rule_1(Bpmn) :- bpmn(Bpmn, "Process") ->
(
    bros(Bros, "Scene"), match(Bpmn, Bros);
    bros(Bros, "Event"), match(Bpmn, Bros)
).

```

Quelltext 3: Regel 1

Regel 2: Zu jeder *BPMN-Swimlane* innerhalb eines *BPMN-Process* muss ein passender *BROS-RoleType* existieren. Eine Swimlane stellt einen Teilnehmer innerhalb des BPMN-Prozesses dar. Dieses Konzept wird in BROS mittels Rollen dargestellt. Daher muss jeder Teilnehmer des BPMN-Prozesses auch als Rolle im BROS-Modell existieren.

```

rule_2(Bpmn) :- bpmn(Bpmn, "Swimlane") ->
(
    bros(Bros, "RoleType"), match(Bpmn, Bros)
).

```

Quelltext 4: Regel 2

Regel 3: Zu jedem *BPMN-TerminationEvent* muss eine *BROS-ReturnEvent* existieren. Eine BPMN-TerminationEvent beendet den aktuellen BPMN-Prozess. Der BPMN-Prozess wird in BROS durch eine BROS-Scene dargestellt (vgl. **Regel 1**). Eine BROS-Scene wird durch ein BROS-ReturnEvent beendet. Somit muss ein BPMN-TerminationEvent mit einem BROS-ReturnEvent modelliert werden.

```

rule_3(Bpmn) :- bpmn(Bpmn, "TerminationEvent") ->
(
    bros(Bros, "ReturnEvent"),
    match(Bpmn, Bros),
    (
        parent(Bros, BrosParent),
        transitive_parent(Bpmn, BpmnParent),
        match(BpmnParent, BrosParent)
    )
).

```

Quelltext 5: Regel 3

Regel 4: Zu jedem *BPMN-EndEvent* muss ein *BROS-Event* oder *BROS-ReturnEvent* existieren. Ein BPMN-EndEvent beendet die aktuelle BPMN-Swimlane. Jede BPMN-Swimlane wird von einem BROS-RoleType repräsentiert (Vgl. **Regel 2**). Ein BROS-RoleType wird mit einem BROS-Event beendet, wenn zwischen diesen beiden eine BROS-DestroyRelation existiert. Alternativ

kann ein BPMN-EndEvent auch durch ein BROS-ReturnEvent modelliert werden, wenn das BPMN-EndEvent das abschließende Event des BPMN-Prozesses ist.

```

rule_4(Bpmn) :- bpmn(Bpmn, "EndEvent") ->
(
  (
    bros(Bros, "Event"),
    match(Bpmn, Bros),
    (
      relation(RoleType, Bros, "DestroyRelation"),
      transitive_parent(Bpmn, BpmnParent),
      match(BpmnParent, RoleType)
    )
  );
  (
    bros(Bros, "ReturnEvent"),
    match(Bpmn, Bros),
    (
      parent(Bros, BrosParent),
      transitive_parent(Bpmn, BpmnParent),
      match(BpmnParent, BrosParent)
    )
  )
).

```

Quelltext 6: Regel 4

Regel 5: Zu jedem *BPMN-StartEvent* muss ein *BROS-Event* existieren. Ein BPMN-StartEvent beginnt die aktuelle BPMN-Swimlane. Jede BPMN-Swimlane wird von einem BROS-RoleType repräsentiert (vgl. **Regel 2**). Ein BROS-RoleType wird mit einem BROS-Event begonnen, wenn zwischen diesen beiden eine BROS-CreateRelation existiert. Alternativ kann ein BPMN-StartEvent auch durch ein BROS-Event modelliert werden, wenn das BPMN-StartEvent das erste Event des BPMN-Prozesses ist und BROS-CreateRelation zwischen dem BROS-Event und der BROS-Szene existiert die den BROS-Prozess darstellt (vgl. **Regel 1**).

```

rule_5(Bpmn) :- bpmn(Bpmn, "StartEvent") ->
(
  bros(Bros, "Event"),
  match(Bpmn, Bros),
  (
    relation(Bros, X, "CreateRelation"),
    transitive_parent(Bpmn, BpmnParent),
    match(BpmnParent, X)
  )
).

```

Quelltext 7: Regel 5

Regel 6: Zu jedem *BROS-Event* bzw. *BROS-ReturnEvent* muss ein *BPMN-Element* existieren. Dies ist die einzige Regel die von dem BROS-Modell aus das BPMN-Modell überprüft. Jedes *BROS-Event* bzw. *BROS-ReturnEvent* muss eine Ursache im BPMN-Modell haben. Es darf kein Event im BROS-Modell geben das nicht im BPMN-Prozess existiert. Ein solches BROS-Event hätte keinen Auslöser im Verhaltensmodell und könnte somit nie eintreten. Passende BPMN-Elemente sind BPMN-Events unabhängig vom Typ, BPMN-Activities oder auch Ausgänge eines BPMN-Gateways.

```

rule_6(Bros) :- (bros(Bros, "Event"); bros(Bros, "ReturnEvent")) ->
(
  bpmn(Bpmn, "StartEvent"), match(Bpmn, Bros);
  bpmn(Bpmn, "EndEvent"), match(Bpmn, Bros);
  bpmn(Bpmn, "TerminationEvent"), match(Bpmn, Bros);
  bpmn(Bpmn, "Event"), match(Bpmn, Bros);
)

```

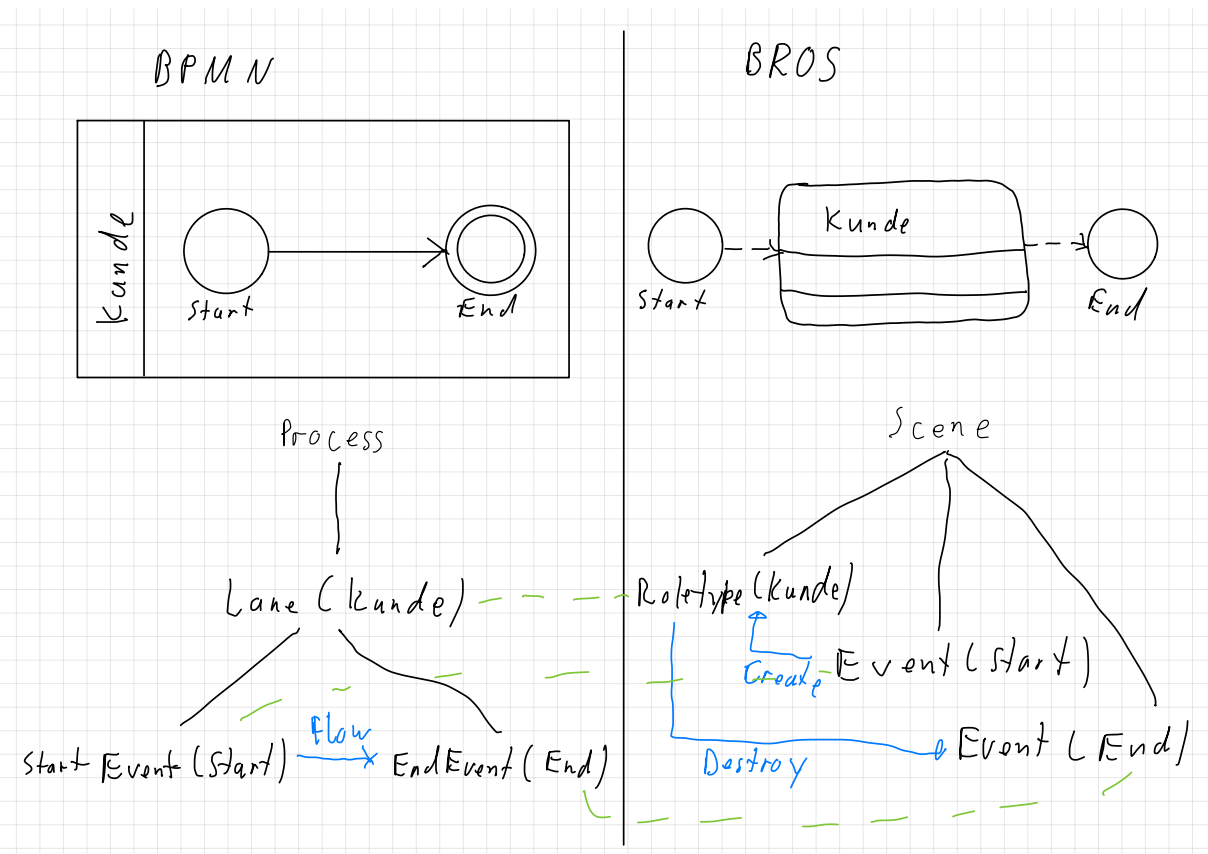


Abbildung 4.1.: Visualisierung des Graph Matching

```

bpmn(Bpmn, "Activity"), match(Bpmn, Bros);
bpmn(Bpmn, "Gateway"), match(Bpmn, Bros)
).

```

Quelltext 8: Regel 6

4.3. Referenzarchitektur

Im Gegensatz zu anderen Arbeiten wurde zur Überprüfung dieser Regeln kein formales Verfahren auf Basis von z.B. *Description Logic* oder *Petrinetzen* genutzt. Dies hat den Vorteil das die Regeln direkt auf den Modellen ausgeführt werden können und nicht erst eine Zwischendarstellung gebaut werden muss. Das hier genutzte Verfahren arbeitet in zwei Stufen auf den Modellen die als *Geschichteten Graph* dargestellt werden. Im ersten Schritt wird ein Matching von Modellelementen aufgebaut. Dies wird iterativ, in Form eines Fixpunkt-Algorithmus, durchgeführt um kaskadierendes Matching zu erlauben. Dieser Schritt wird im folgenden *Matching-Algorithmus* genannt. Im zweiten Schritt werden anhand des Matching die Regeln ausgeführt. Dabei werden die Regelergebnisse aggregiert. Dieser Vorgang wird *Verifikations-Algorithmus* bezeichnet.

Um das Verfahren anzuwenden müssen beiden Modelle in Form eines gerichteten geschichteten Graph vorliegen. Ein geschichteter Graph basiert auf einem Baum der die Grundstruktur des Modelles und die Eltern-Kind Beziehung abdeckt. Um die Relationen zwischen den Modellelementen darzustellen wird ein Schicht mit Querverweisen über den Baum gelegt. In Abbildung 4.1 wird die Transformation der Modelle in die dazugehörigen Graphen dargestellt. Auf der linken Seite ist ein BPMN-Modell mit dem dazugehörigen Graphen. Auf der rechten Seite ist ein vereinfachtes BROS-Modell mit seinem Graphen. Der schwarze Teil der Graphen bildet die Grundstruktur oder auch die erste Schicht. Dieser wird mit Hilfe der "containsRelation der Metamodelle direkt

aufgebaut. In blau ist die zweite Schicht, die Schicht der Relationen, visualisiert. Jede Relation ist mit ihrem Typ annotiert und hat eine feste Richtung. Die Graphen des BPMN-Modelles und des BROS-Modelles sind zueinander disjunkt.

Mit beiden Graphen kann nun der Matching-Algorithmus durchgeführt werden. Das Matching ist die dritte Schicht innerhalb der Graphen, die den Graphen des BPMN-Modelles mit dem Graphen des BROS-Modelles verbindet. In Abbildung 4.1 wird dies in grün dargestellt. Um diese Schicht zu konstruieren wird eine Orakelfunktion genutzt. Diese ermittelt ob zwei Elemente zueinander gehören. Dabei hat die Orakelfunktion zwei Parameter, den aktuellen Knoten des BPMN-Graphen und den des BROS-Graphen. Die übergebenen Knoten enthalten Referenzen auf ihre Relationen aus allen drei Schichten. In der ersten Schicht sind die Vorfahren und Nachkommen und in der zweiten Schicht die per Relation verbundenen Elemente. Die dritte Schicht ist zu Beginn leer. Wenn die Orakelfunktion für das BPMN-Element x und das BROS-Element y ein Matching feststellt, werden die Kanten (x, y) und (y, x) der dritten Schicht hinzugefügt. Eine einmal hinzugefügt Kante kann nicht wieder entfernt werden. Da die dritte Schicht auch mit an das Orakel übergeben wird, kann dieses aufgrund von einem bereits bestehen Matching eine Entscheidung treffen. Um dieses Verhalten zu nutzen wird der Vorgang des Matching aufbaues solange wiederholt bis sich die dritte Schicht nicht weiter ändert. Dieses Verhalten wird Fixpunkt-Algorithmus genannt. Innerhalb einer Iteration des Algorithmus wird die Orakelfunktion auf allen Paaren von Elementen des BPMN- und des BROS-Modelles angewendet. Ein Fixpunkt-Algorithmus kann unter Umständen nicht terminieren, wenn die Datenstruktur in einen oszillierenden Zustand übergeht. Dies wird verhindert indem Kanten nur zu dem Matching hinzugefügt werden und somit nie entfernt werden dürfen, wodurch das Matching monoton wachsend ist. Damit kann die dritte Schicht maximal zu einem Vollständigen Graphen anwachsen. Anschließend kann keine weitere Kante hinzugefügt werden und der Algorithmus terminiert automatisch.

Mit dem konstruierten Matching kann nun der Verifikations-Algorithmus ausgeführt werden. In diesem Schritt werden die unter Abschnitt 4.2 aufgestellten Konsistenzregeln überprüft. Diese Überprüfung kann mittels der genannten Prolog Regeln oder auch direkt auf den Graphen durchgeführt werden. Anders als der Matching-Algorithmus wird der Verifikations-Algorithmus nur einmal ausgeführt. Da alle Regeln die Konsistenzprüfung von einem Modell auf das andere Modell durchführen, können die Regeln die auf dem BPMN-Modell basieren unabhängig von den Regeln die auf dem BROS-Modell basieren ausgeführt werden. So werden die Regeln die die Überprüfung aus Sicht des BPMN-Modelles ausführen, für jedes Element des BPMN-Graphen angewendet. Um eine bessere Fehlermeldung für den Modellierer zu erstellen, wird im Falle einer Regelverletzung nicht nur das BPMN-Element, sondern auch das BROS-Element das zu dem Regelverstoß führt, als negative Konsistenzmeldung gespeichert. Mit Hilfe der beiden Elemente und der verletzen Regel kann eine genaue Fehlerbeschreibung gegeben werden. Dieses Verhalten ist analog für die Regel die aus Sicht des BROS-Modelles arbeiten. Nicht mit den Prologregeln darstellbar ist der Unterschied zwischen erfolgreichen und abgebrochenen Regelprüfungen. Jede Regel hat zu Beginn eine Implikation die die Gültigkeit der Regel auf bestimmte Elemente einschränkt. Für eine optimale Rückmeldung an den Modellierer muss zwischen einer falschen Vorbedingung und einer wahren Vorbedingung mit wahrer Konsequenz unterschieden werden. Im Fall das eine Regel eine wahre Vorbedingung und eine wahre Konsequenz aufweist wird eine positive Konsistenzmeldung gespeichert, andernfalls kann die Regelausführung ignoriert werden.

Das Ergebnis der Konsistenzprüfung ist eine Liste von Konsistenzmeldungen. Jede Konsistenzmeldung besteht aus ihrem Typ (positiv oder negativ), den betroffenen BPMN- und BROS-Elementen, der zugrundeliegenden Regel und einer textuellen Beschreibung. Die negativen Konsistenzmeldungen sind die gefundenen Inkonsistenzen zwischen dem BPMN- und BROS-Modell. Wenn beide Modelle konsistent zueinander sind, beinhaltet das Ergebnis nur positive Konsistenzmeldungen. Diese helfen dem Modellierer die automatische Konsistenzprüfung zu kontrollieren. Ohne die positiven Konsistenzmeldungen hätte der Modellierer keine Möglichkeit zu überprüfen ob überhaupt eine Konsistenzprüfung stattgefunden hat oder ob dabei Fehler aufgetreten sind.

5. Implementierung der automatischen Konsistenzprüfung

Nachdem das hier vorgestellte Verfahren theoretisch erläutert wurde, wird nun eine praktische Umsetzung vorgestellt. Dafür werden im ersten Abschnitt die technischen Rahmenbedingungen aufgeführt. Anschließend wird jeweils die Implementierung des *Matching-Algorithmus* und des *Verifikations-Algorithmus* beschrieben. Im vierten Abschnitt wird die Benutzerschnittstelle des implementierten Tools vorgestellt.

5.1. Implementierung der Referenzarchitektur

Im Abschnitt 4.2 werden die Konsistenzregeln mit Hilfe von einem Prolog-Dialekt eingeführt. Dies legt nahe für die Implementierung auch eine Sprache zu verwenden die auf Prolog basiert oder einfach mit Prolog interagieren kann. Allerdings hat die Implementierung in Prolog einen entscheidenden Nachteil. Für den *Verifikations-Algorithmus* wird ein dreiwertiger Rückgabewert pro Regel benötigt. Dies lässt sich mit Prolog nicht direkt abbilden. Zudem lassen sich die Regeln dank des *Graph-Logik-Isomorphismus* auch direkt auf dem Graphen anwenden. Eine weitere Anforderung an die Implementierung ist eine mögliche Integration in den bestehenden BROS-Editor *FRaMED.io*. Dieser basiert auf Web-Technologien und ist in der objektorientierten Programmiersprache Kotlin entwickelt. Aus diesem Grund wird auch Kotlin für die Entwicklung der Referenzimplementierung verwendet. Kotlin ist als JVM-Sprache entwickelt worden, kann aber auch zu Javascript und LLVM-Bytecode transpiliert werden. Dadurch besitzen Kotlin Programme eine hohe Erweiterbarkeit im Bezug auf andere Laufzeitumgebungen. Wie auch im BROS-Editor *FRaMED.io* wird die Transpilierung zu Javascript genutzt. Ziel der Implementierung ist es, bestehende BPMN- und BROS-Modelle, mit Hilfe des hier vorgestellten Verfahrens, auf Konsistenz zu überprüfen.

Als Dateiformat für BPMN-Modelle wird das auf XML basierende (**.bpmn*) Format von dem BPMN-Editor *bpmn.io* unterstützt. Der BPMN-Editor *bpmn.io* ist eine moderne Webanwendung zum editieren von BPMN-Modellen. Das Dateiformat enthält sowohl Struktur- als auch graphische Informationen. Diese sind in zwei XML-Namespace aufgeteilt. Die Struktur wird im Namespace *bpmn*, und die graphische Notation im Namespace *bpmndi* gespeichert. Für die Konsistenzprüfung ist nur der Namespace *bpmn* relevant. Der Namespace *bpmndi* und alle seine Kindelemente werden im folgenden ignoriert. Zum lesen der BPMN-Datei wird der XML-Parser verwendet der von Javascript nativ bereitgestellt wird. Von den Entwicklern von *bpmn.io* existiert bereits ein Parser und Validierer für BPMN-Dateien (*bpmn-moddle*). Dieser wird explizit nicht verwendet

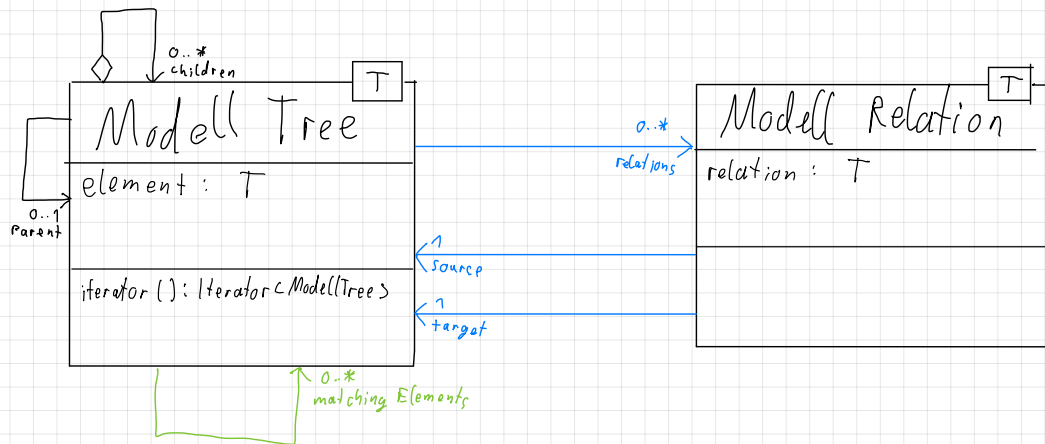


Abbildung 5.1.: Metamodell der Graphstruktur

da er in Javascript geschrieben wurde und somit eine zu starke Bindung an die Plattform Javascript erzeugt. Die integrierte Validierungsfunktion für BPMN-Modelle wird auch nicht benötigt da eine Voraussetzung für die Benutzung des Tools die syntaktische und semantische Korrektheit der Modelle ist. Mit Hilfe des XML-Parsers wird das BPMN-Modell als eine Instanz des BPMN-Metamodells geladen.

Die enge Kopplung zwischen dem BROS-Editor FRaMED.io und diesem Tool legt nah auch das auf JSON basierende Format des BROS-Editors zu nutzen. Analog zu dem BPMN-Editor, ist auch der BROS-Editor webbasiert und das Dateiformat enthält sowohl Struktur- als auch graphische Informationen. Die graphischen Informationen befinden sich im Root-Objekt unter dem Schlüsselwort *layer*. Auch diese graphischen Informationen werden im weiteren ignoriert. Zum lesen der BROS-Dateien wird der bereits existierende Parser aus dem BROS-Editor FRaMED.io genutzt. Dieser ist in Kotlin geschrieben und benutzt den nativen JSON-Parser von Javascript. Genau wie bei dem BPMN-Parser wird auch das BROS-Modell in eine Instanz des BROS-Metamodells geladen.

Bevor der *Matching-Algorithmus* und der *Verifikations-Algorithmus* beginnen können, müssen beide Instanzen des Metamodells noch in die Form eines Graphen konvertiert werden. Die Datenstruktur des Graphen basiert auf dem Metamodell aus Abbildung 5.1. In dieser Abbildung sind die drei Schichten des Graphen farbig hervorgehoben. In schwarz ist die erste Schicht dargestellt. Dabei besitzt jeder Knoten eine Liste von Kind-Knoten. Jeder Knoten hat zusätzlich noch einen Verweis auf seinen Eltern-Knoten. Zusätzlich besitzt jeder Knoten noch eine Referenz auf seine Metamodell-Instanz mit Typannotation. Die zweite Schicht ist in blau dargestellt und repräsentiert die Relation zwischen den Knoten. Dabei hat jeder Knoten ein Set von Verweisen auf seine Relationen. Die Relationen verweisen auf auf ihren Quell- und Zielknoten. Auch die Graphrelationen besitzen eine Referenz auf ihre Metamodell-Instanz mit Typannotation.

$$\text{'Aktion war erfolgreich'} , \text{'ErfolgreicheAktion'} \quad (5.1)$$

$$\{\text{'aktion'}, \text{'erfolgreich'}, \text{'war'}\} , \{\text{'aktion'}, \text{'erfolgreiche'}\} \quad (5.2)$$

$$\{\text{'aktion'}, \text{'erfolgreiche'}\} , \{\text{'aktion'}, \text{'erfolgreich'}, \text{'war'}\} \quad (5.3)$$

$$\{\text{'aktion'} \subseteq \text{'aktion'}\} , \{\text{'erfolgreiche'} \subseteq \text{'erfolgreich'}\} \quad (5.4)$$

Abbildung 5.2.: Anwendung des Name-Matching

Schließlich ist die dritte Schicht, die das Matching zwischen den Modellen darstellt mit grün markiert. Dabei handelt es sich um ein Set von Knoten. Jeder Knoten hat zusätzlich noch eine Hilfsfunktion die einen Iterator zurückgibt. Diese implementiert eine einfache Breitensuche um über den Graphen zu iterieren.

5.2. Matching der Modellelemente

Um die Verifizierung der Modelle zu ermöglichen muss zunächst ein dazugehöriges Matching aufgebaut werden. Dafür benötigt der *Matching-Algorithmus* eine Implementierung der Orakelfunktion. Dabei hat es sich als ausreichend gezeigt Typ-kompatible Elemente anhand ihres Namens zu vergleichen. In Quelltext 17 ist der für den Namensvergleich genutzte Algorithmus abgebildet. Dabei werden die Namen zunächst in einzelne Wörter gesplittet und anschließend unabhängig ihrer Endung und Reihenfolge verglichen. Um die Funktionsweise dieses Algorithmus zu verdeutlichen wird dieser beispielhaft in Abbildung 5.2 angewendet. Der Aufruf der Vergleichsfunktion erfolgt mit den beiden zu vergleichenden Zeichenketten (vgl. Schritt 5.1). Zunächst werden beide Namen in ihre Wörter zerlegt (vgl. Schritt 5.2) und nach der Länge sortiert (vgl. Schritt 5.3). Die Worttrennung erfolgt an Leerzeichen und vor jedem Großbuchstaben der nach einem Kleinbuchstaben steht. Diese Einschränkung verhindert die zeichenweise Trennung von Namen die vollständig großgeschrieben sind. Anschließend wird für jedes Element der ersten Menge ein passendes Element in der zweiten Menge gesucht. Die Zeichenketten müssen nicht komplett übereinstimmen. Es ist ausreichend wenn einer der beiden Strings ohne seine Endung ein Teil des anderen Strings ist. Um zu verhindern das ein sehr kurzer String in einem langen String erkannt wird, darf die Längenunterschied nicht größer als die Länge der Endung sein. Wenn zu jedem Element der ersten Menge ein passendes Element in der zweiten Menge gefunden wird, stimmen die beiden Namen überein (vgl. Schritt 5.4).

Zusätzlich zum Name-Matching müssen Regeln definiert werden auf welchen Elementen das Name-Matching angewendet werden muss. Jede Regel muss dafür das *Matching* Interface implementieren. Um diese Regeln gebündelt zu erstellen und sammeln gibt es die Hilfsklasse *Context*. Dieses bietet verschiedene Hilfsfunktionen um auf einfache Art und Weise Matching- und Verifikations-Regeln zu implementieren. Dabei benötigt die Hilfsfunktion zum erstellen einer Matching Regel drei Parameter. Die ersten beiden Parameter sind die Referenzen auf das BPMN- und BROS-Metamodell nach denen die Graphknoten gefiltert werden. Mit den Sprachfunktionen von Kotlin können diese Referenzen als generischer Parameter der Funktion übergeben werden. Der dritte Parameter ist ein Lambda das die beiden Knoten des BPMN-Graphen und des BROS-Graphen auf einen Wahrheitswert abbildet. Das Lambda wird nur ausgeführt wenn die Knoten zu den Filtern der ersten beiden Parameter passen. Wenn das Lambda wahr zurück gibt werden der dritten Schicht die beiden Kanten zwischen dem BPMN- und BROS-Knoten hinzugefügt, sofern diese nicht bereits existieren. Sonst wird das Ergebnis ignoriert, da nur Kanten zum Matching hinzugefügt werden können. In Quelltext 9 ist das Matching für einer BPMN-Swimlane und einem BROS-RoleType gegeben. Mittels der generischen Parameter werden die Graph-Knoten gefiltert und das Lambda wird nur auf den betreffenden Knoten ausgeführt. Im Lambda wird hier nur ein

einfaches Name-Matching durchgeführt. Die weitere Struktur wird nicht betrachtet.

```
Context.match<BpmnLane, BrosRoleType> { lane, role ->
    matchStrings(lane.element.name, role.element.name)
}
```

Quelltext 9: Matching Regel von einer BPMN-SwimLane und einem BROS-RoleType

Im zweiten Beispiel Quelltext 10 wird das Matching zwischen einem BPMN-Gateway und einem BROS-Event abgebildet. Hierbei ist zu beachten das die Namen der BPMN-Gateway Ausgänge nicht im Metamodell des Gateways sondern im Metamodell der weiterführenden BPMN-Flows gespeichert sind. Nachdem die Knoten nach den Metamodell-Typen gefiltert wurden, wird überprüft ob der Name eines BPMN-SequenceFlows der von dem BPMN-Gateway ausgeht mit dem BROS-Event übereinstimmt. Dafür bietet die Knoten-Klasse eine weitere Hilfsfunktion die die verbunden Relationen nach ihrem Typ gefiltert zurückgeben kann.

```
Context.match<BpmnGateway, BrosEvent> { bpmn, bros ->
    bpmn.relations<BpmnSequenceFlow>().any { flow ->
        flow.relation.name.isNotBlank() &&
            matchStrings(flow.relation.name, bros.element.desc)
    }
}
```

Quelltext 10: Matching Regel von einem BPMN-Gateway und einem BROS-Event

Das dritte Beispiel Quelltext 11 zeigt die Vorteile des Fixpunkt-Algorithmus. Diese Regel besagt das ein BPMN-Event ein äquivalentes Matching wie seine per BPMN-MessageFlow verbunden Nachbarn besitzt. Nach dem filtern der Graphknoten, wird über alle BPMN-MessageFlows iteriert. Ein Matching wird dann gefunden, wenn die Quelle oder das Ziel des BPMN-MessageFlows bereits ein Matching mit dem BROS-Event besitzt. Dies ermöglicht das schreiben einfacherer Regeln.

```
Context.match<BpmnEvent, BrosEvent> { bpmn, bros ->
    bpmn.relations<BpmnMessageFlow>().any { flow ->
        flow.source in bros.matchingElements ||
        flow.target in bros.matchingElements
    }
}
```

Quelltext 11: Fixpunkt Matching Regel von einem BPMN-BpmnEvent und einem BROS-Event

Allerdings kann es auch vorkommen das trotz dieser Regeln ein Matching zwischen zwei Elementen nicht gefunden wird oder auch fälschlicher Weise gefunden wird. Für diesen Fall hat der Modellierer die Möglichkeit ein sogenanntes *Predefined-Matching* hinzuzufügen. Damit lässt sich ein Matching zwischen Elementen hinzufügen oder auch entfernen. Das *Predefined-Matching* wird in jeder Iteration des Fixpunkt-Algorithmus, nachdem die Regeln auf den gesamten Graphen angewendet wurden ausgeführt. Durch das entfernen von Kanten aus dem Graphen ist es bei einem Fixpunkt-Algorithmus theoretisch möglich in einen nicht terminierenden Zustand überzugehen. Da aber in jeder Iteration, vor dem überprüfen auf Änderungen in der dritten Schicht, ein konstante Kantenmenge entfernt wird, reduziert dies nur den vollständigen Graphen auf einen Graphen ohne diese Kantenmenge. Dabei bleibt die dritte Schicht, am Ende jeder Iteration, monoton wachsend.

5.3. Implementierung der Konsistenzregeln

Nach dem Aufbau des Matching in der dritten Schicht, kann der *Verifikations-Algorithmus* arbeiten. Für das implementieren der Regeln stellt das *Context* Klasse zwei Hilfsfunktionen bereit. Dies sind die Funktionen *verifyBpmn* und *verifyBros* welche jeweils zwei Parameter benötigen. Der erste Parameter ist der Filter des Graph-Knoten. Dieser stellt die Vorbedingung aus der

Implikation in Abschnitt 4.2 dar. Der zweite Parameter ist ein Lambda das den gefilterten Graphknoten auf ein Konsistenzmeldung abbildet. Damit wird die Konsequenz der Implikation dargestellt. Neben einer positiven und einer negativen Konsistenzmeldung kann auch ein *null* Wert zurückgegeben werden. Ein *null* Wert bedeutet, das die Regel nicht anwendbar ist und das Ergebnis daher ignoriert werden soll. Damit wird die unter Abschnitt 4.2 beschriebende Dreiwertigkeit umgesetzt. Um dies zu veranschaulichen werden nachfolgend die Konsistenzregeln 2, 3 und 5 in ihre Kotlin-Implementierung überführt. Zeichenketten innerhalb der Implementierung werden dabei weggelassen und können in der Implementierung nachgelesen werden.

Regel 2 ist die einfachste zu implementierende Regel. Sie besagt das zu jeder BPMN-Swimlane ein passender BROS-RoleType existieren muss. Die zweite Regel ist auf dem BPMN-Graphen basiert und filtert in der Vorbedingung nach der BPMN-Swimlane. Dies wird mit dem Metamodell dargestellt und als erster generischer Parameter der Hilfsfunktion *verifyBpmn* übergeben. In der Konsequenz der Implikation wird nach einem BROS-RoleType gesucht das ein Matching mit der BPMN-Swimlane aufweist. Bei der Kotlin-Implementierung wird dies aus der anderen Richtung betrachtet. Da alle Elemente die ein Matching mit der BPMN-Swimlane habe bekannt sind, wird unter diesen Elementen ein BROS-RoleType gesucht. Sobald eines gefunden ist wird eine positive Konsistenzmeldung zurückgegeben. Um die Rückmeldung an den Modellierer zu verbessern wird der Konsistenzmeldung noch eine textuelle Beschreibung und eine Referenz auf den Knoten des BROS-Graphen, auf den die Regel erfolgreich angewendet wurde, hinzugefügt. Die Referenz zu dem Knoten des BPMN-Graphen wird automatisch von der Hilfsfunktion hinzugefügt. Sollte kein Element gefunden werden, wird eine negative Konsistenzmeldung zurückgeben. Auch diese Meldung beinhaltet eine textuelle Beschreibung der Fehlerursache. Die Referenzangabe zu einem weiteren Knoten ist optional und kann, wenn die Information nicht verfügbar oder für die Auswertung des Modellierers nicht notwendig ist, weggelassen werden.

```
Context.verifyBpmn<BpmnLane> { bpmn ->
    for (bros in bpmn.matchingElements) {
        if (bros.checkType<BrosRoleType>()) {
            return@verifyBpmn Result.match("...", bros = bros)
        }
    }
    Result.error("...")
}
```

Quelltext 12: Implementierung von Regel 2

Die **Regel 3** ist ähnlich aufgebaut wie Regel 2. In der Regel 3 wird überprüft das zu jedem BPMN-TerminationEvent ein passendes BROS-ReturnEvent existiert. Genau wie die Regel 2 basiert sie auf dem BPMN-Graphen. Allerdings ist ein BPMN-TerminationEvent im BPMN-Metamodell ein normales BPMN-Event das zusätzlich ein Attribut TerminationEvent besitzt. Hierfür ist die Dreiwertigkeit des Rückgabewertes wichtig. Zunächst wird in der Vorbedingung nach einem BPMN-Event gefiltert. Anschließend wird jedes gefundene BPMN-Element überprüft, ob es auch ein BPMN-TerminationEvent ist. Sollte dies nicht der Fall sein wird das Lambda mit dem Rückgabewert null beendet. Damit wird die Ausführung der Regel komplett ignoriert, wie es auch bei dem generischen Filterargument geschieht. Wenn nun ein BPMN-TerminationEvent gefunden ist, kann, analog zu Regel 2, in der dritten Schicht nach einem BROS-ReturnEvent gesucht werden. Der Rückgabewert null ermöglicht es auch komplexere Vorbedingungen leicht zu überprüfen.

```
Context.verifyBpmn<BpmnEndEvent> { bpmn ->
    if (!bpmn.element.terminationEvent) return@verifyBpmn null

    for (bros in bpmn.matchingElements) {
        if (bros.checkType<BrosReturnEvent>()) {
            return@verifyBpmn Result.match("...", bros = bros)
        }
    }
}
```

```

    Result.error("...")
}

```

Quelltext 13: Implementierung von Regel 3

Regel 5 gehört zu den komplexeren Regeln. Sie überprüft das zu jedem BPMN-StartEvent ein passendes BROS-Event existiert. Zusätzlich müssen beide Events auch zueinander passende Elemente erstellen bzw. starten. In der Vorbedingung wird überprüft, das das BPMN-Element eine BPMN-Event ist, keine BPMN-TerminationEvent ist und der Typ des BPMN-Events ein BPMN-StartEvent ist. Dabei wird erneut der null Rückgabewert genutzt. Anschließend wird in der dritten Schicht nach einem BROS-Event gesucht. Bei dem BROS-Event wird in der zweiten Schicht nach einer BROS-CreateRelationship gesucht. Diese verbindet ein BROS-Event mit dem von ihm erzeugten Element. Nun wird überprüft ob das Ziel der BROS-CreateRelationship sich mit in der dritten Schicht des Elternelementes des BPMN-StartEvents befindet. Ein BPMN-StartEvent beginnt seine BPMN-Swimlane, die auch sein Elternelement ist. Wenn dieses Überprüfung erfolgreich ist wird eine positive Konsistenzmeldung zurückgegeben, sonst eine negative. Beide Konsistenzmeldung haben neben der textuellen Beschreibung auch eine Referenz auf das BPMN-Event. Sollte kein passendes BPMN-Event gefunden werden, wird auch eine negative Konsistenzmeldung zurückgegeben, allerdings ohne eine Referenz auf ein BROS-Event.

```

Context.verifyBpmn<BpmnStartEvent> { bpmn ->
    for (bros in bpmn.matchingElements) {
        if (bros.checkType<BrosEvent>()) {
            val bpmnParents = bpmn.transitiveParent<BpmnElement>()
            val broCreate = bros.relations<BrosCreateRelation>()
                                .firstOrNull()
                                ?.source as? ModelTree<BrosElement>

            for (bpmnParent in bpmnParents) {
                if (broCreate != null &&
                    broCreate in bpmnParent.matchingElements) {
                    return@verifyBpmn Result.match("...", bros = bros)
                }
            }
            return@verifyBpmn Result.error("...", bros = bros)
        }
    }
    Result.error("...")
}

```

Quelltext 14: Implementierung von Regel 5

5.4. Benutzerinterface

Da das entwickelte Tool eine Webanwendung ist kann es in allen moderneren Browsern benutzt werden die Javascript aktiviert haben. Die Benutzung des Tools ist in zwei Schritte unterteilt. Im erster Abschnitt müssen die Quelldateien der zu überprüfenden Modelle geladen werden. Dies ist in Abbildung 5.3 zu sehen. Im mittleren UI Bereich können die Modelldateien geladen werden. Dazu können die Dateien einfach per Drag'n'Drop in das Tool gezogen werden. Das Tool erkennt den Inhalt unabhängig des Namens und importiert die entsprechende Datei. Alternativ kann eine manuelle Dateiauswahl genutzt werden oder der Inhalt der Datei in das entsprechende Textfeld kopiert werden. Unterhalb der Textfelder wird eine kurze Information zu dem geladenen Modell oder eine Fehlermeldung im Parsing-Vorgang angezeigt. *Im oberen rechten Bereich des UIs befindet sich ein Button mit dem sich ein Demo-Projekt laden lässt um das UI zu testen. (Button entfernen?)* Sobald jeweils ein valides BPMN- und BROS-Modell geladen wurden,

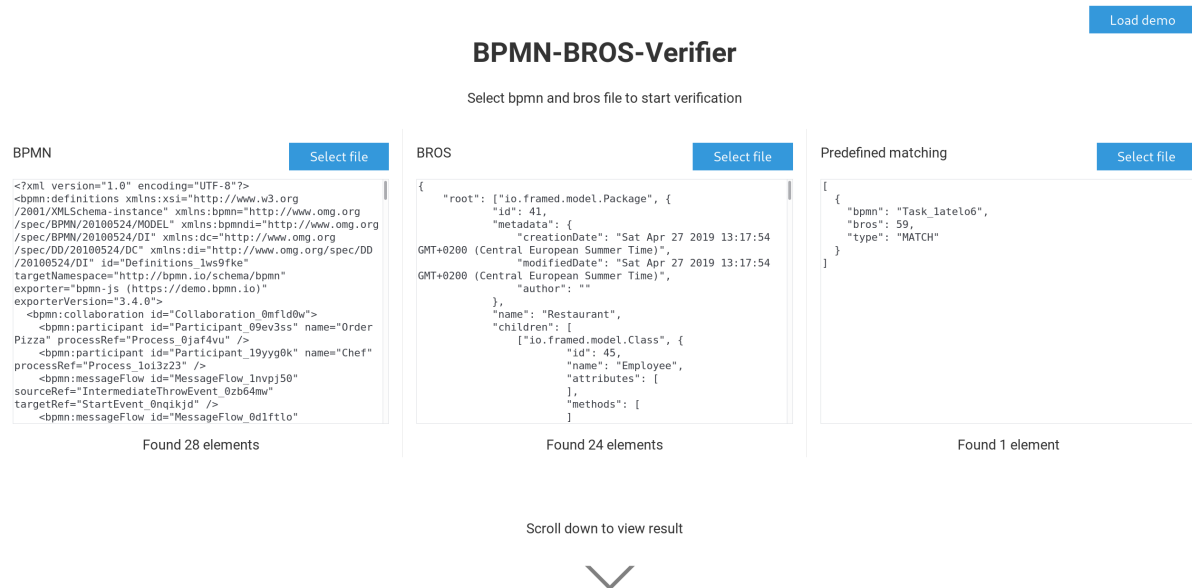


Abbildung 5.3.: Benutzerinterface des Dateiuploads

wird die Konsistenzprüfung automatisch gestartet. Der Ergebnisfeil im unteren Bildschirmbereich signalisiert das ein Verifizierungsergebnis verfügbar ist.

Mit einem Klick auf den Ergebnisfeil oder durch das manuelle scrollen gelangt man in den zweiten UI Abschnitt. Dieser ist in Abbildung 5.4 dargestellt. Hier sind vier verschiedene Bereiche zu sehen. Im oberen Bereich werden statistische Informationen angezeigt. Direkt unterhalb des Informationsbereiches befindet sich die Navigationsleiste mit der sich der Inhalt des Hauptbereiches auswählen lässt. Der Hauptbereich zeigt die Validierungsergebnisse und Informationen zu dem Matching an. Im unteren Bildschirmbereich befindet sich eine optional einblendbare Eingabemaske, mit der ein PredefinedMatching erstellt werden kann.

Der Informationsbereich teilt sich in vier weitere Sektionen. Rechts befindet sich die Eigenschaftsauswahl. Mit ihr können verschieden Elemente im Hauptbereich nach ihrem Typ ein- und ausgeblendet werden. Zusätzlich kann das PredefinedMatching deaktiviert werden. Dies kann der Modellierer nutzen um die Ergebnisse übersichtlicher darzustellen. Die drei linken Sektionen beinhalten statistische Informationen zu der Durchführung und dem Ergebnis der Konsistenzprüfung. Dabei werden auch Informationen zu dem Matching angezeigt.

- **Verification stats:** Informationen zu der aktuellen Verifikation.
 - *Successful checks (x of y):* Anzahl der x positiven Konsistenzmeldung von allen y Konsistenzmeldung.
 - *Errors (x of y):* Anzahl der x negativen Konsistenzmeldung von allen y Konsistenzmeldung.
 - *Coverage ($x\%$):* Prozentuale Anzeige x der *Successful checks*.
 - *Fixed point matching rounds (x):* Anzahl der durchgeführten Matching Runden x aufgrund des Fixpunkt-Algorithmus. Mögliche Werte von x sind:
 - * 1: Es wurde kein Matching gefunden.
 - * 2: Es wurde ein Matching gefunden. Es gibt kein Matching was auf ein anderes Matching aufbaut.

Verification stats Successful checks 14 of 18 Errors 4 of 18 Coverage 77% Fixed point matching rounds 3	BPMN matching stats Matched elements 17 of 28 Unmatched elements 11 of 28 Multiple matches 1 Coverage 60%	BROS matching stats Matched elements 12 of 24 Unmatched elements 12 of 24 Multiple matches 5 Coverage 50%	Features <input checked="" type="checkbox"/> Use predefined matching <input checked="" type="checkbox"/> Show errors <input checked="" type="checkbox"/> Show warnings <input checked="" type="checkbox"/> Show infos <input checked="" type="checkbox"/> Show successful
Verify result	BPMN matching	BROS matching	Predefined matching
<div> <div> <div>BPMN ID: Process_0ja4vu</div> <div>BpmnProcess(Order Pizza)</div> <div>Message</div> <div>BpmnProcess(Order Pizza) matches BrosScene(OrderPizza)</div> </div> <div> <div>BROS ID: 47</div> <div>BrosScene(OrderPizza)</div> </div> <div>Module</div> <div>Rule 1 - BpmnProcess</div> </div>			
<div> <div> <div>BPMN ID: Process_1oi3z23</div> <div>BpmnProcess(Chef)</div> <div>Message</div> <div>BpmnProcess(Chef) matches BrosRoleType(Chef)</div> </div> <div> <div>BROS ID: 63</div> <div>BrosRoleType(Chef)</div> </div> <div>Module</div> <div>Rule 1 - BpmnProcess</div> </div>			
<div> <div> <div>BPMN ID: Lane_0ie3gyp</div> <div>BpmnLane(Customer)</div> <div>Message</div> <div>BpmnLane(Customer) matches BrosRoleType(Customer)</div> </div> <div> <div>BROS ID: 48</div> <div>BrosRoleType(Customer)</div> </div> <div>Module</div> <div>Rule 2 - BpmnLane</div> </div>			
<div> <div> <div>BPMN ID: Lane_0y4qyff</div> <div>BpmnLane(Waiter)</div> <div>Message</div> <div>BpmnLane(Waiter) matches BrosRoleType(Waiter)</div> </div> <div> <div>BROS ID: 60</div> <div>BrosRoleType(Waiter)</div> </div> <div>Module</div> <div>Rule 2 - BpmnLane</div> </div>			
<div> <div> <div>BPMN ID: EndEvent_0azq1qn</div> <div>BpmnEndEvent(order cancelled)</div> <div>Message</div> </div> <div> <div>×</div> <div> <div>BPMN ID: Lane_0ie3gyp</div> <div>BpmnLane(Customer)</div> </div> <div> <div>BROS ID: 48</div> <div>BrosRoleType(Customer)</div> </div> <div>Nomatch</div> <div>Create</div> <div>minationEvent</div> </div> </div>			

Abbildung 5.4.: Benutzerinterface der Validierungsergebnisse

- * 3-5: Es wurde ein Matching gefunden. Es gibt ein oder mehrere Matchings die auf andere Matching aufbauen.
- * ≥ 6 : Es wurde ein Matching gefunden. Es ist möglich das ein kaskadieren Effekt eingetreten ist und ein zu großes Matching aufgebaut wurde. Das Matching sollte von dem Modellierer überprüft werden.

- **BPMN/BROS matching stats** Informationen zu dem Matching aus Sicht des BPMN bzw. BROS Modelles.
 - *Matched elements (x of y)*: Anzahl der x Elemente mit gefundenem Matching von allen y Elemente.
 - *Unmatched elements (x of y)*: Anzahl der x Elemente ohne gefundenem Matching von allen y Elemente.
 - *Multiple matches (x)*: Anzahl der x Elemente die mit mehreren Elementen ein Matching haben.
 - *Coverage (x%)*: Prozentuale Anzeige x der *Matched elements*.

Im Hauptbereich befindet sich die Liste der Validierungsergebnisse. Mit Hilfe der Navigationsleiste kann zusätzlich das BPMN- oder BROS-Matching oder auch das geladene Predefined-Matching angezeigt werden. Jeder Eintrag der Liste hat den gleichen Aufbau. Auf der linken Seite des Eintrages ist farblich der Typ markiert. Je nach Tab gibt es unterschiedliche Farbkodierungen. Im Hauptteil des Eintrages sind drei Textfelder. Diese sind das BPMN-Element, das BROS-Element und die textuelle Beschreibung des Eintrages. Zusätzlich ist noch ein viertes Textfeld vorhanden das den Namen der Regel angibt, die diesen Eintrag erstellt hat. Mit einem Klick auf das Textfeld des BPMN- oder BROS-Elementes kann direkt ein PredefinedMatching erstellt oder gelöscht werden. Die Verifizierungsergebnisse haben die Farbkodierungen grün (positive Konsistenzmeldung) und rot (negative Konsistenzmeldung). Die Matching Ergebnisse sind blau (Matching für dieses Element gefunden) und gelb (keine Matching für dieses Element gefunden) markiert. Da sich nicht alle Elemente eines Modelles auf das andere Abbilden lassen, ist ein Fehlendes

Matching meißt kein Fehler. Genauso ist das vorhanden sein eines Matchings nicht immer korrekt, da das Matching auf die Namen angewiesen ist. Aus diesem Grund ist die Farbkodierung des Matchings schwächer als die der Verifizierung. Ein bestehendes PredefinedMatching kann auch im Tab PredefinedMatching mit dem Löschen Button oben rechts in jedem Eintrag entfernt werden.

6. Fallstudie

Nachdem die verschiedenen Konsistenzregeln aufgestellt und implementiert wurden, soll nun die Benutzbarkeit und die Erweiterbarkeit des Tools gezeigt werden. Dafür wird in diesem Kapitel eine beispielhafte Anwendung des Tools aus Sicht des Benutzers durchgeführt (Abschnitt 1) und anschließend eine neue Regeln aus Sicht des Entwicklers implementiert (Abschnitt 2).

6.1. Anwendung am Beispiel einer Pizzabestellung

In seiner Arbeit hat Schön et al. o.D. für die Einführung in BROS das Beispiel einer Pizzabestellung erstellt. Das dort gegebene BPMN-Modell wird als Grundlage genutzt um ein neues BROS-Modell zu erstellen und die Konsistenz mit Hilfe des neu entwickelten Tools zu überprüfen. Die Modelle werden zu diesem Zweck neu in bpmn.io und FRaMED.io modelliert um sie automatisiert überprüfen zu können. Nachdem ein konsistentes BROS-Modell entwickelt wurde, wird die Benutzbarkeit des Tools diskutiert.

In dem Geschäftsprozess der Pizzabestellung gibt es drei Teilnehmer, die in Abbildung 6.1 zu sehen sind. Der erste Teilnehmer ist der *Kunde*. Dieser startet den Prozess indem er eine Bestellung aufgibt. Die Bestellung wird von dem *Kellner* aufgenommen und an den *Koch* weiter gegeben. Wenn die Bestellung von dem Koch bearbeitet wird und die Bestellung länger benötigt als gewöhnlich kann sich der Kunde bei dem Kellner beschweren, dessen Aufgabe dann das beruhigen des Kunden ist. Sollte dies nicht erfolgreich sein kann der Kunde sich beschweren und seine Bestellung zurücknehmen, was den Prozess vorzeitig beenden. Nachdem der Koch die Pizza fertig

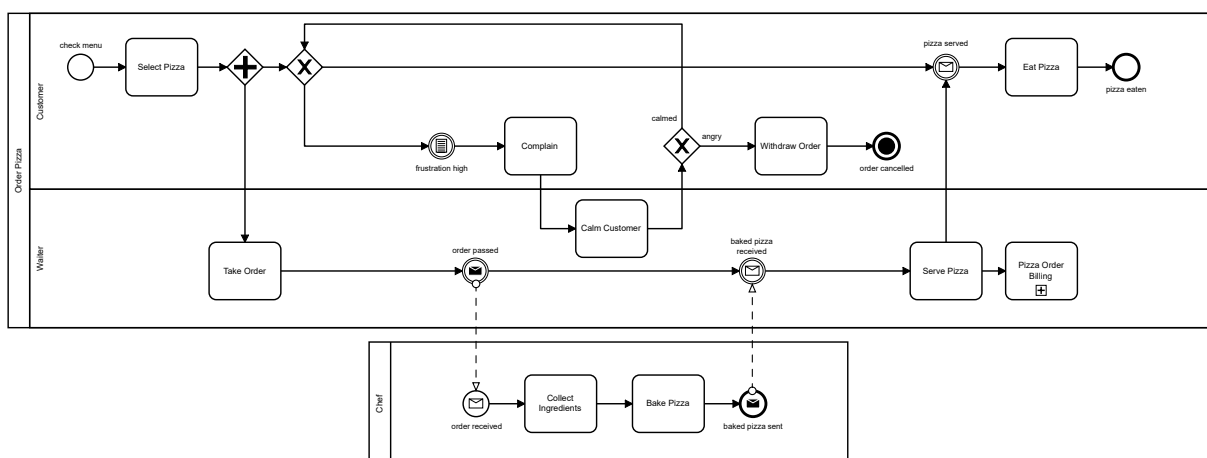


Abbildung 6.1.: BPMN-Modell der Pizzabestellung

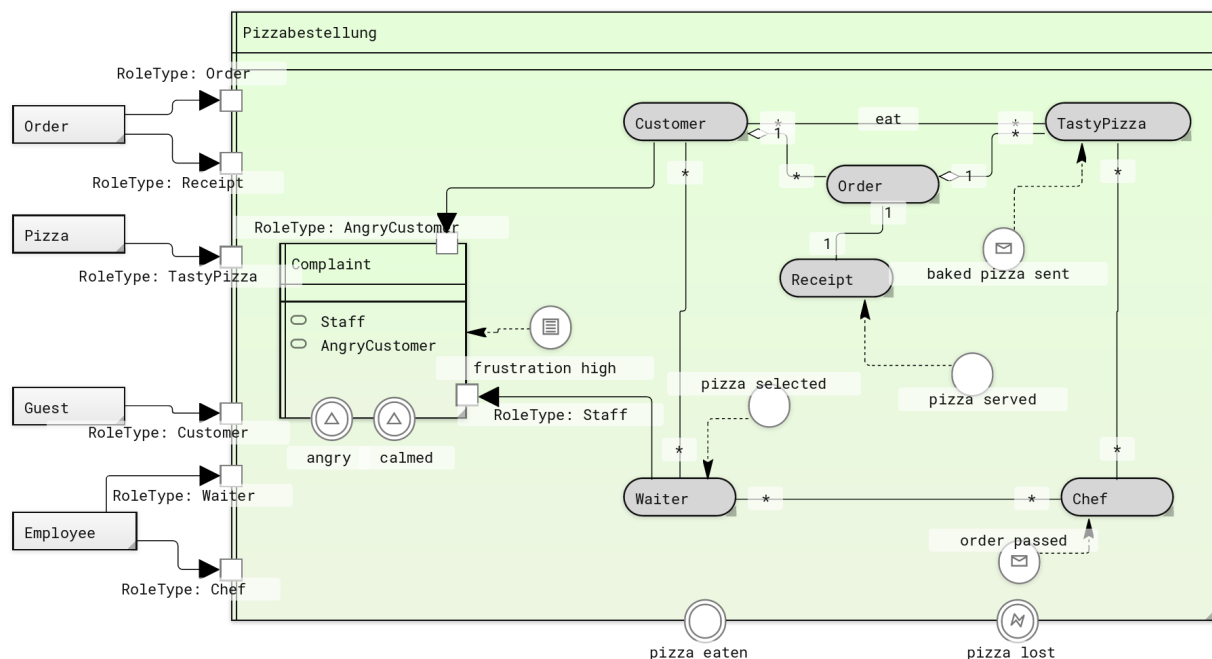


Abbildung 6.2.: Entwurf für das BROS-Modell der Pizzabestellung

zubereitet hat übergibt er sie wieder dem Kellner der die Bestellung dem Kunden übereicht. Anschließend erstellt der Kellner noch die Rechnung für den Kunden. Sobald der Kunde die Pizza gegessen hat ist der Prozess beendet.

Für dieses BPMN-Modell kann nun ein dazu passendes BROS-Modell entworfen werden (vgl. Abbildung 6.2). Aus dem Prozess lassen sich vier verschiedene Entitäten entnehmen. Dies sind *Bestellung*, *Pizza*, *Gast* und *Mitarbeiter*, welche als *ClassType* modelliert werden. Der gesamte Prozess wird mit der Scene *Pizzabestellung* abgebildet. Die drei Teilnehmer aus dem BPMN-Modell werden mit Rollen innerhalb der Scene übersetzt, die mit Events verbunden sind. Der Ablauf der Beschwerde von Kunden wird mit einer eigenständigen Scene modelliert, die zwei verschiedene *ReturnEvents* besitzt.

Nachdem beide Modelle erstellt wurden, kann die Konsistenzprüfung mittels des Tools erfolgen. Dafür werden die beiden gespeicherten Dateien, mittels Drag'n'Drop in das Benutzerinterface gezogen, wodurch die Prüfung automatisch startet. Bei der Überprüfung konnten die Regeln 18 mal angewendet werden und es wurden 12 mal positive Konsistenzmeldungen zurückgegeben. Damit sind in dem BROS-Modell 6 Konsistenzprobleme aufgetreten. Die davon betroffenen Regeln sind die Regeln 1, 3, 4, 5 und 6.

Der erste Regelverstoß basiert auf der Regel 1 und besagt das für den Bpmn-Prozess "Örder Pizza" kein entsprechendes BROS-Element gefunden werden konnte. Das liegt an dem Namensunterschied gegenüber der BROS-Scene, die "Pizzabestellung" heißt. Dieses Konsistenzproblem lässt sich auf zwei unterschiedlichen Wegen lösen. Zum einen kann ein ein Predefined Matching zwischen dem BPMN-Prozess und der BROS-Scene erstellt werden. Dazu kann der BPMN-Prozess mit einem klick auf die ID markiert werden. Die BROS-Scene ist unter dem Reiter *BROS matching* auffindbar und kann mit einem klick ebenfalls markiert werden. In dem Dialog am unterem Bildschirmrand kann nun das Matching hinzugefügt werden, wodurch die Überprüfung erneut durchgeführt wird. Um die Konsistenz auch für zukünftige Modelle für diesen Geschäftsprozess zu erhalten kann die BROS-Scene auch umbenannt werden. Nachdem die BROS-Scene zu "Örder-Pizza" umbenannt (vgl. Abbildung A.1a) wurde kann die Konsistenzprüfung mit dem laden der neuen Datei wiederholt werden.

Durch das umbenennen der BROS-Scene hat sich auch ein weiteres Konsistenzproblem gelöst, so das nun nur noch 4 Probleme vorliegen. Das zweite Problem kommt von der Regel 3 und be-

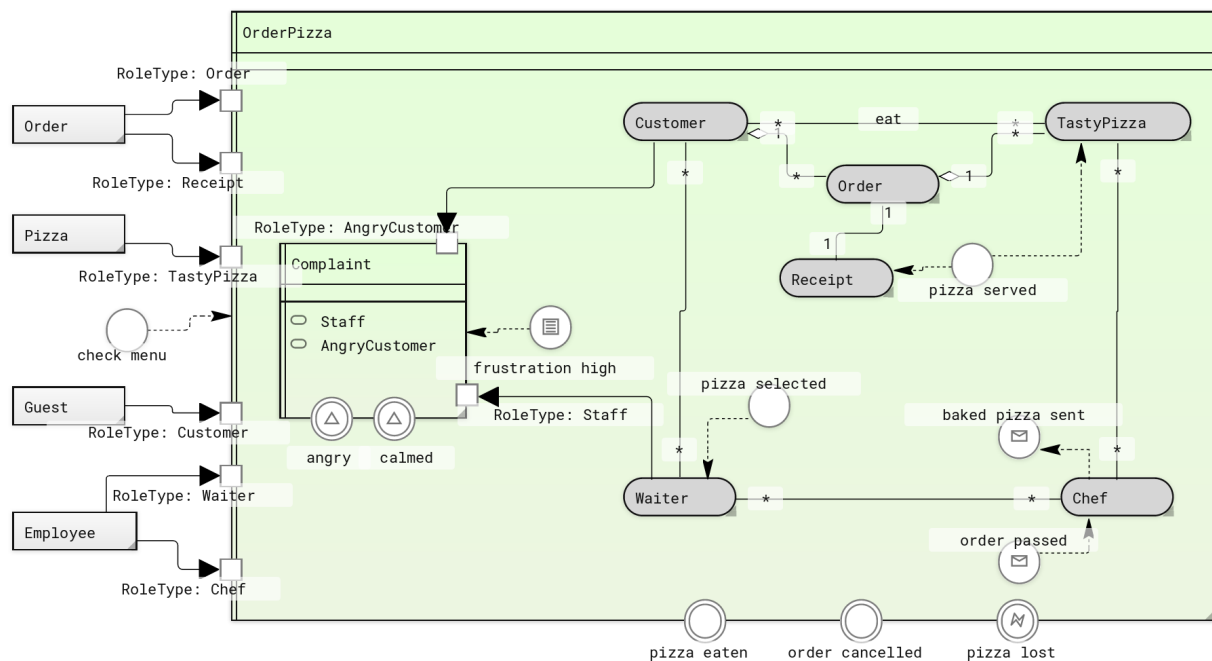


Abbildung 6.3.: Vollständiges BROS-Modell der Pizzabestellung

schreibt das zu dem BPMN-TerminationEvent "order cancelled" kein passendes BROS-ReturnEvent gefunden wurde. Dieses Problem stammt von dem vergessenen BROS-ReturnEvent. Es kann leicht behoben werden indem zu der BROS-Szene noch ein weiteres BROS-ReturnEvent mit dem Namen "order cancelled" hinzugefügt wird (vgl. Abbildung A.1b). Dabei erhöht sich auch die Anzahl der zutreffenden Regeln um eins auf 19, da ein weiteres BROS-Event existiert, das mit Regel 6 überprüft werden muss.

Für das dritte Konsistenzproblem ist Regel 4 verantwortlich. Es besagt das für BPMN-EndEvent "baked pizza send"war ein BROS-Event im Matching gefunden werden konnte, aber die Elemente die von den Events beendet werden unterschiedlich sind. Das BPMN-EndEvent beendet die BPMN-Swimlane für den Koch, das BROS-Event hingegen beendet nichts. Dies liegt an einer fehlerhaften Verknüpfung der BROS-Event. Momentan wird die BROS-Rolle "TastyPizza" von dem BROS-Event "baked pizza send" erzeugt, was allerdings falsch ist. Für das erzeugen dieser BROS-Rolle ist das BROS-Event "pizza served" verantwortlich. Das BROS-Event "baked pizza send" wird hingegen mit einer BROS-DestroyRelation mit der BROS-Rolle des Koches verbunden, wie in Abbildung A.1c zu sehen ist.

Der vorletzte Regelverstoß basiert auf der Regel 5 und ist trivial zu lösen. Er beschreibt, das zu dem BPMN-StartEvent "check menu" kein passendes BROS-Event mit BROS-CreateRelation zu der BROS-Szene "OrderPizza" existiert. Dieses BROS-Event wurde im bisherigen BROS-Modell vergessen. Zum lösen des Konsistenzproblems muss ein BROS-Event mit dem Namen "check menu" und der BROS-CreateRelation hinzugefügt werden (vgl. Abbildung 6.3).

Anschließend bleibt nur noch ein Konsistenzproblem übrig. Dieses besagt das ein Verstoß gegen Regel 6 vorliegt, indem zu dem BROS-Event "pizza lost" kein passendes BPMN-Element gefunden wurde. Das ist ein Problem, welches sich nicht einfach lösen lässt, da kein passendes BPMN-Element existiert. Mit dem BROS-Event wurde das Modell um eine neue Möglichkeit erweitert, die die Konsistenz verletzt. Um dieses Problem zu lösen muss entweder das BROS-Event gelöscht oder das BPMN-Modell angepasst werden. Wenn das BROS-Event aber erhalten bleiben soll und es keine Möglichkeit gibt das BPMN-Modell zu ändern, muss dieser Regelverstoß ignoriert werden.

Damit zeigt fig:pizzaBros6 das vollständige BROS-Modell das so weit wie möglich mit dem BPMN-Modell konsistent ist. Dank des Tools konnten viele Konsistenzprobleme gefunden und

leicht behoben werden. Bei der letzten negativen Konsistenzmeldungen des Tools wurde gezeigt, dass dieser Regelverstoß absichtlich herbeigeführt wurde. Das Tool hat zurzeit keine Möglichkeit mit bewussten Konsistenzproblemen umzugehen. Hierfür wäre eine Funktion zur manuellen Abstufung der Regelverletzung von einem Fehler zu einer Warnung oder Information hilfreich.

6.2. Erweiterbarkeit des Ansatzes

Bisher wurden nur Regeln vorgestellt, die die Konsistenz von BPMN-Modellen zu BROS-Modellen prüfen. Da das BROS-Modell gegenüber dem BPMN-Modell angereichert werden kann ist dies auch die häufigste Anwendung. Allerdings können auch einige Regeln in die andere Richtung überprüft werden. Um gleichzeitig die Erweiterbarkeit des Ansatzes und der Implementierung zu zeigen wird **Regel 6** aus Abschnitt 4.2 hinzugefügt. Die Regel 6 überprüft das zu jedem BROS-Event und BROS-ReturnEvent ein dazugehöriges BPMN-Element existiert. Dabei können die BPMN Elemente von den Typen BPMN-Activity, BPMN-Gateway und BPMN-Event sein. Der genaue Typ des BPMN-Events ist dabei nicht relevant, es kann sich dabei unter anderem um ein BPMN-StartEvent oder auch ein BPMN-TerminationEvent handeln.

Jede Regel sollte zur besseren Übersicht in eine eigene Datei ausgelagert werden. Die bereits bestehenden Regeln befinden sich im Package *io.framed.modules*. Für die Regel 6 wird dafür die Datei *Rule6BrosEvent.kt* erstellt. Jede Datei enthält eine Funktion die einmalig zum Setup der Regeln ausgeführt wird. Dies hat die Signatur *fun Context.setupRule6BrosEvent()*. Damit wird ein Parameter definiert, der den Typ *Context* hat und als Receiver (*this*) genutzt werden kann. Innerhalb dieser Funktion können nun die Matching- und Verifikations-Funktionen definiert werden. Die explizite Angabe des *Context* Receiver Parameters ist nun nicht mehr nötig, da dies durch die Setup-Funktion an innere Funktionen propagiert wird.

Die Matching Regeln zwischen den verschiedenen BPMN-Events und dem BPMN-Gateway wurden im Abschnitt 5.2 bereits implementiert. Für die Erweiterung muss damit nur die Matching Regel für BPMN-Activities mittels Name-Matching hinzugefügt werden. Dabei wird in den generischen Parametern nach BPMN-Activities und BROS-Events gefiltert. Diese werden mittels ihrer Namen verglichen und bei erfolgreichen Vergleich zum Matching hinzugefügt. Da BROS-Events und BROS-ReturnEvents unterschiedliche Metamodell Typen haben, muss diese Matching-Regel ein weiteres mal hinzugefügt werden, wobei der generische Parameter für das BROS-Modell auf ein BROS-ReturnEvent gesetzt wird. Die Implementierung des Lambdas bleibt unverändert.

```
match<BpmnTask, Event> { bpmn, bros ->
    matchStrings(bpmn.element.name, bros.element.desc)
}
```

Quelltext 15: Matching Regel zwischen BPMN-Activities und BROS-Events

Nachdem das Matching auf die unterstützten Modellelemente erweitert wurde, kann nun die Verifikationsregel implementiert werden. Anders als die in Abschnitt 5.3 implementierten Regeln ist Regel 6 von dem BROS-Modell aus gerichtet. Darum wird nicht die Funktion *verifyBpmn* sondern äquivalente Funktion *verifyBros* genutzt. Im generischen Parameter der Funktion wird nach einem BROS-Event gefiltert. Innerhalb des Lambdas wird überprüft ob es ein Element im Matching des BROS-Events gibt das ein BPMN-Element ist. Hier wird zur Vereinfachung des Quellcodes ausgenutzt, dass mit den Matching Regeln keine unzulässigen BROS-Elemente im Matching aufgenommen werden können. Sollte das Matching mit anderen BROS-Event zu BPMN-Element Regeln erweitert werden, muss im Schleifenkörper eine genauere Überprüfung des BPMN-Element Typs erfolgen. Analog zu der neuen Matching-Regel muss auch die Verifikationsregel ein weiteres mal mit dem Filter nach BROS-ReturnEvents eingefügt werden.

```
verifyBros<Event> { bros ->
    for (element in bros.matchingElements) {
        val match = element.element as? BpmnElement ?: continue
    }
}
```

```

        return@verifyBros Result.match("...", bpmn = element)
    }
    Result.error("...")
}

```

Quelltext 16: Implementierung von Regel 6

Ein Nachteil der Kotlin/JS Plattform ist das Fehlen von Reflection und Annotationprocessing wie in Java bzw. Kotlin/JVM. Aus diesem Grund wird die neu hinzugefügte Setup-Funktion nicht automatisch erkannt und muss manuell registriert werden. In der Datei *io.framed.modules.Main.kt* existiert eine Liste von Setup-Funktionen die ausgeführt werden. Um nun die neue Regel hinzuzufügen wird der Liste ein neuer Eintrag (*Context::setupRule6BrosEvent*) hinzugefügt. Der Aufruf *::* auf eine Funktion gibt eine Funktionsreferenz zurück und erlaubt eine spätere Ausführung. Die vollständige Implementierung der Regel 6 ist unter Quelltext 18 zu finden.

7. Schluss

In diesem Kapitel soll evaluiert werden in wie weit diese Arbeit die im Kapitel 1 genannten Ziele erfüllt hat.

7.1. Zusammenfassung

Ziel der Arbeit war die Konsistenz zwischen BPMN- und BROS-Modellen zu überprüfen. Dafür wurden bestehende Konsistenzverfahren für UML-Modelle analysiert und anhand ihrer Eigenschaften klassifiziert. Darauf aufbauend wurden sechs Regeln für die Konsistenzprüfung aufgestellt und formalisiert. Für die Automatisierung der Konsistenzprüfung wurde eine Referenzarchitektur erstellt, die die formalen Regeln ausführt. Um die Funktionalität des Verfahren zu zeigen, wurde ein Tool in Kotlin implementiert um zusätzlich eine mögliche Anbindung an den BROS-Editor FRaMED.io zu erleichtern. Diese Implementierung wurde anschließend anhand des Beispiels einer Pizzabestellung von Schön et al. o.D. getestet indem ein zu dem BPMN-Modell konsistentes BROS-Modell entwickelt wurde. Außerdem wurde die Erweiterbarkeit dieses Verfahrens Regel evaluiert, indem eine weitere Regel zu der Implementierung hinzugefügt wurde.

7.2. Wissenschaftlicher Beitrag

Das in dieser Arbeit vorgestellte Verfahren gehört zu der Klasse der Verfahren ohne Zwischendarstellung (vgl. Tabelle 7.1). Da es auf dem Auswerten verschiedener Konsistenzregeln basiert, nutzt es die Strategie des Monitorings. Damit ist der ähnlichste Ansatz im Bezug auf UML-Modelle das Verfahren nach Egyed 2006. Beide Verfahren testen die Intra-Modell (horizontale) Konsistenz, sind automatisierbar und wurden bereits evaluiert. Sie unterscheiden sich neben den unterstützten Modellarten hauptsächlich im Grad der Erweiterbarkeit. Das Verfahren von Egyed 2006 benutzt zum Konsistenzvergleich Regeln auf basis von OCL. Damit ist die Erweiterbarkeit um neue Modellarten nicht möglich, wenn diese nicht zum UML-Standard gehören. Das hier genutzte Verfahren hat auch einen hohen Aufwand, wenn eine neue Modellart hinzugefügt wird. Allerdings ist die Erweiterbarkeit um neue Regeln mit weniger Aufwand verbunden. Das das Verfahren direkt auf dem Graphen der Metamodell-Instanz arbeitet können beliebige Regeln leicht erstellt werden. Egyed 2006 Verfahren ist hingegen auf die Möglichkeiten der OCL beschränkt, was die Erstellung einer neuen Regeln erschweren könnte.

7.3. Zukünftige Arbeiten

Eine Abgrenzung dieser Arbeit war, das keine Handlungsempfehlungen oder Lösungsmöglichkeiten bei gefundenen Inkonsistenzen gegeben werden. In diesem Feld sollte evaluiert werden welche

	Diagrams	Consistency Type	Consistency Strategy	Intermediate Representation	Case Study	Automatable	Tool Support	Model Extensibility	Rule Extensibility
Rasch 2003	CD, SM	Intra	Monitoring	CSP/OZ	1	H	0	H	M
Shinkawa 2006	UCD, CD, SD, AD, SC	Inter	Analysis	CPN	0	H	0	M	L
Mens 2005	CD, SD, SC	All	Monitoring	Extended UML	1	H	1	H	M
Egyed 2001	CD, OD, SD	Intra, Inter	Construction		0	H	part	M	M
Egyed 2006	CD, SD, SC	Intra	Monitoring		1	H	1	L	M
BROS	BPMN, BROS	Intra	Monitoring		1	H	1	L	H

Tabelle 7.1.: Einordnung des neuen Verfahrens

Inkonsistenzmuster häufig auftreten und ob diese eine Vorhersehbare Lösungsmöglichkeit aufweisen. Auch könnte die Fehlertoleranz des Tools erhöht werden, indem man eine syntaktische und semantische Konsistenzprüfung der Einzelmodelle integriert. Diese ist momentan noch von externen Programmen abhängig oder aufgabe des Modellierers. Da externe Programme meist auf diese Aufgabe spezialisiert sind müsste zunächst evaluiert werden ob eine integrierte Lösung auch eine gleichbleibende Erkennungsrate besitzt. Um die Benutzbarkeit des Tools weiter zu verbessern kann eine Integration in FRaMED.io erfolgen. Die technische Grundlage ist mit der Referenzimplementierung bereits gegeben. Hierfür müsste noch evaluiert werden wie das Ergebnis der Analyse in die graphische Darstellung integriert werden kann.

Literatur

- Briand, Lionel C, Yvan Labiche und Leeshawn O'Sullivan (2003). „Impact analysis and change management of UML models“. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*. IEEE, S. 256–265.
- Egyed, Alexander (2001). „Scalable consistency checking between diagrams-The VIEWINTEGRA approach“. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, S. 387–390.
- (2006). „Instant consistency checking for the UML“. In: *Proceedings of the 28th international conference on Software engineering*. ACM, S. 381–390. DOI: 10.1145/1134285.1134339.
- Kim, S-K und David Carrington (2004). „A formal object-oriented approach to defining consistency constraints for UML models“. In: *2004 Australian Software Engineering Conference. Proceedings*. IEEE, S. 87–94. DOI: 10.1109/aswec.2004.1290461.
- Loja, Luiz Fernando Batista et al. (2010). „A business process metamodel for enterprise information systems automatic generation“. In: *Anais do I Congresso Brasileiro de Software: Teoria e Prática-I Workshop Brasileiro de Desenvolvimento de Software Dirigido por Modelos*. Bd. 8, S. 37–44.
- Lucas, Francisco J, Fernando Molina und Ambrosio Toval (2009). „A systematic review of UML model consistency management“. In: *Information and Software Technology* 51.12, S. 1631–1645. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.009.
- Mens, Tom, Ragnhild Van Der Straeten und Jocelyn Simmonds (2005). „A framework for managing consistency of evolving UML models“. In: *Software Evolution with UML and XML*. IGI Global, S. 1–30.
- Nuseibeh, Bashar (1996). „To be and not to be: On managing inconsistency in software development“. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE, S. 164–169.
- Rasch, Holger und Heike Wehrheim (2003). „Checking Consistency in UML Diagrams: Classes and State Machines“. In: *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, S. 229–243. DOI: 10.1007/978-3-540-39958-2_16.
- Schön, Hendrik et al. (o.D.). „Business Role-Object Specification: A Language for Behavior-aware Structural Modeling of Business Objects“. In: ().
- Shinkawa, Yoshiyuki (2006). „Inter-Model Consistency in UML Based on CPN Formalism“. In: *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. IEEE, S. 411–418. DOI: 10.1109/apsec.2006.41.
- Simmonds, Jocelyn et al. (2004). „Maintaining Consistency between UML Models Using Description Logic“. In: *L'OBJET* 10.2-3, S. 231–244. ISSN: 1262-1137. DOI: 10.3166/objet.10.2-3.231-244.

Usman, Muhammad et al. (2008). „A Survey of Consistency Checking Techniques for UML Models“. In: *2008 Advanced Software Engineering and Its Applications*. IEEE, S. 57–62. DOI: 10.1109/asea.2008.40.

A. Appendix

```

fun matchStrings(string1: String, string2: String): Boolean {

    val a = stringToSet(string1)
    val b = stringToSet(string2)

    val longer: Set<String>
    val shorter: Set<String>
    if (a.size >= b.size) {
        longer = a
        shorter = b
    } else {
        longer = b
        shorter = a
    }

    if (
        shorter.isEmpty() ||
        shorter.size.toDouble() / longer.size.toDouble() <
        WORD_LENGTH_THRESHOLD
    ) {
        return false
    }

    return shorter.all { short ->
        val s = short.take(
            max(
                MIN_WORD_LENGTH,
                short.length - WORD_ENDING_LENGTH
            )
        )

        longer.any { long ->
            val l = long.take(
                max(
                    MIN_WORD_LENGTH,
                    long.length - WORD_ENDING_LENGTH
                )
            )

            long.startsWith(s) ||
            short.startsWith(l) &&
            abs(l.length - s.length) <= WORD_ENDING_LENGTH
        }
    }
}

private fun stringToSet(str: String): Set<String> = str
    .replace(SPLIT_CAMEL_CASE_REGEX, "$1_$2")
    .split("_")
    .map { it.toLowerCase().trim() }
    .filter { it.length >= MIN_WORD_LENGTH }
    .toSet()

private val SPLIT_CAMEL_CASE_REGEX = "([a-z])([A-Z])".toRegex()
private const val WORD_LENGTH_THRESHOLD = 0.6
private const val MIN_WORD_LENGTH = 3
private const val WORD_ENDING_LENGTH = 2

```

Quelltext 17: Algorithmus zum Name-Matching

```

fun Context.setupRule6() {

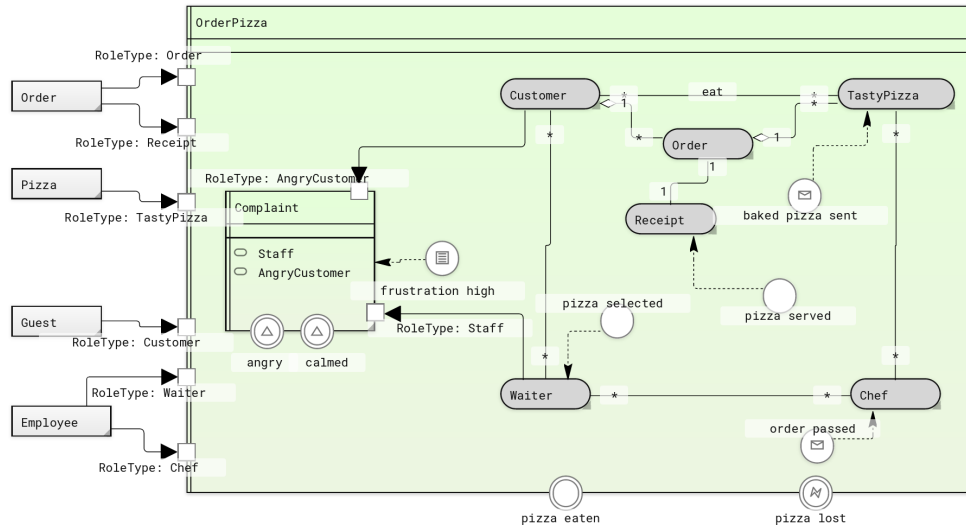
    match<BpmnTask, Event> { bpmn, bros ->
        matchStrings(bpmn.element.name, bros.element.desc)
    }
    match<BpmnTask, ReturnEvent> { bpmn, bros ->
        matchStrings(bpmn.element.name, bros.element.desc)
    }

    verifyBros<Event>("Rule_6_-_BrosEvent") { bros ->
        for (element in bros.matchingElements) {
            val match = element.element as? BpmnElement ?: continue
            return@verifyBros Result.match("...", bpmn = element)
        }
        Result.error("...")
    }

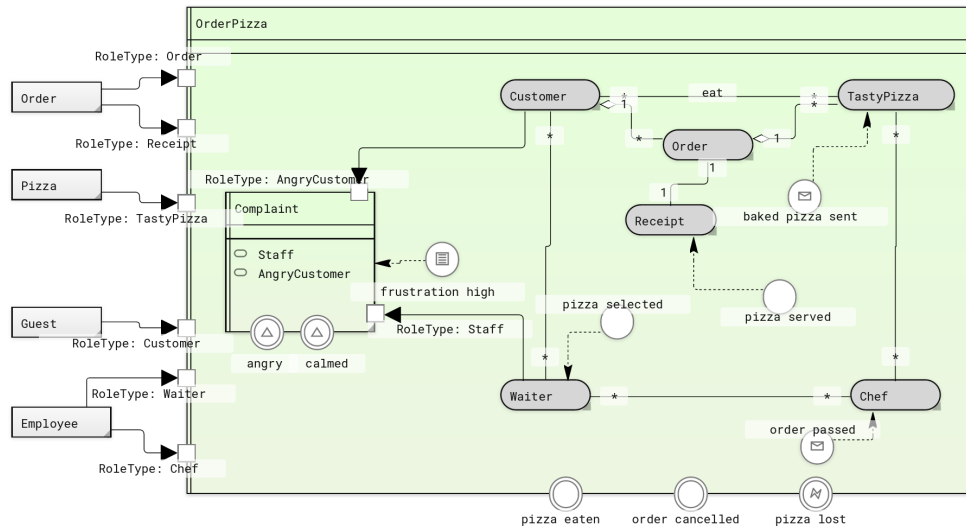
    verifyBros<ReturnEvent>("Rule_6_-_BrosReturnEvent") { bros ->
        for (element in bros.matchingElements) {
            val match = element.element as? BpmnElement ?: continue
            return@verifyBros Result.match("...", bpmn = element)
        }
        Result.error("...")
    }
}

```

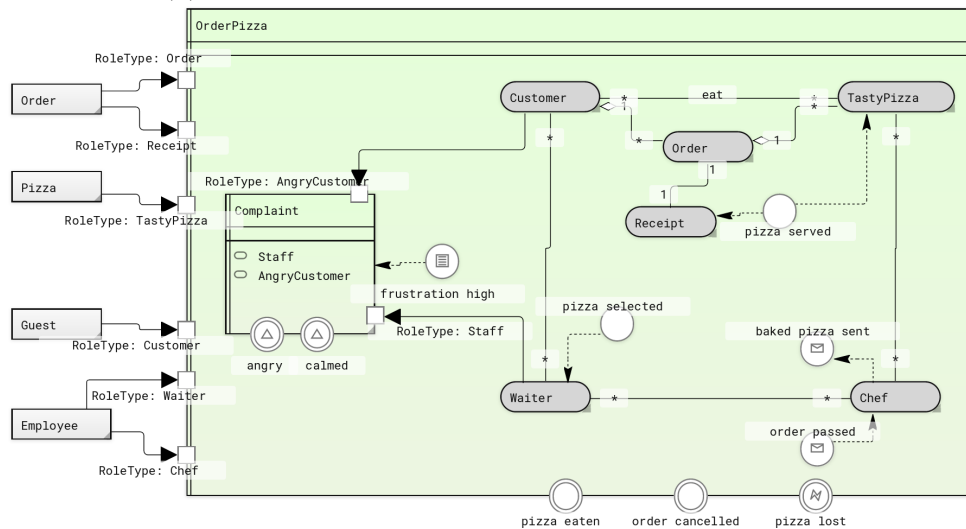
Quelltext 18: Implementierung der Datei Rule6BrosEvent.kt



(a) Entwurf 2 für das BROS-Modell der Pizzabestellung



(b) Entwurf 3 für das BROS-Modell der Pizzabestellung



(c) Entwurf 4 für das BROS-Modell der Pizzabestellung

Abbildung A.1.: Entwurfsschritte für das BROS-Modell der Pizzabestellung