TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur für Softwaretechnologie

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

Lars Westermann

Bachelorarbeit

Submitted on: 29.08.2019

Inhaltsverzeichnis

Abstract

Die heutigen Methoden zur Erstellung von Software hängen stark von geeignet 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, Geschäftsprozesse (zum Beispiel 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 einem bestimmten Geschäftsprozess 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 Konsistenzprobleme zu ermitteln. Konsistenzprobleme treten auf, wenn bestimmte Aspekte der Konsistenz zwischen den Modellarten verletzt werden. Basierend auf dieser Analyse werden dem Modellierer Warnungen gegeben, wenn Konsistenzprobleme auftreten und was die Ursache dieser Probleme ist. 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

Eine der wichtigesten Voraussetzungen für das Gelingen heutiger Softwaresysteme sind geeignet definierte Modelle. Die verschiedenen genutzten Modellarten müssen zwei verschiedene Aspekte der Software abbilden. Auf der einen Seite sind die strukturbasierten Modelle. Diese spezifizieren unter anderem den internen Aufbau eines Systems. Dafür werden häufig UML-Strukturdiagramme verwendet, wie zum Beispiel Klassendiagramme oder Komponentendiagramme. Auf der anderen Seite werden verhaltensbasierte Modelle benötigt. Mit ihnen werden Abläufe und andere Verhaltensaspekte dargestellt. Oftmals werden beispielsweise BPMN-Diagramme, Petrinetze und UML-basierte Diagramme wie Sequenzdiagramme genutzt. Allerdings reichen die einzelnen Modelltypen nicht aus um das System ausreichend zu beschreiben. Die Abläufe innerhalb der Geschäftsprozesse werden mit den verhaltensbasierten Modellen modelliert. Die eigentliche Implementierung des Systems benötigt aber zusätzlich die strukturbasierten Modelle. Für den Erfolg eines Softwareprojektes ist es daher essenziell, dass die Strukturmodelle an die Verhaltensmodelle ausgerichtet sind, damit das System die im Geschäftsprozess definierten Abläufe umsetzen kann.

Um dies zu ermöglichen, wurde die Modellierungssprache *Business Role-Object Specification* (BROS) erstellt. Sie schließt die Lücke zwischen den Geschäftsprozessen, in Form von BPMN-Prozessen, und den strukturbasierten Modellen. BROS erreicht dies, indem zeitliche Elemente in die statische strukturbasierte Modellspezifikation eingefügt werden. Die Überprüfung der Konsistenz zwischen dem Geschäftsprozess und der BROS, ist eine komplexe und fehleranfällige Aufgabe, die der Modellierer manuell ausführen muss. Daher wird in dieser Arbeit die Konsistenz zwischen BROS und der prozeduralen BPMN untersucht. Eine möglichst automatisierte Konsistenzprüfung spart den Modellierern Arbeitsaufwand Auch die Korrektheit der Überprüfung kann sich erhöhen.

1.2. Problemdefinition

Die Überprüfung der Konsistenz zwischen Modellen wird auf dem Gebiet von UML aktiv erforscht und es existieren viele verschiedene Ansätze. So kann nicht nur die Konsistenz eines Modelles zu seinen vertiefenden Modellen, sondern auch die Konsistenz zwischen mehreren Modellarten überprüft werden. Dies ermöglicht es, die Konsistenz der Modelle fortlaufend, während der gesamten Konzeptions- und Entwicklungsphase eines Projektes, automatisiert zu überprüfen. Allerdings gilt dies nicht für die Konsistenz zwischen der neuen Modellierungssprache BROS und der BPMN. Für diese Modelle gibt es Verfahren mit Toolunterstützung, um die interne Konsistenz der Modelle zu überprüfen. Jedoch es noch kein Verfahren oder Ansatz, um

die Konsistenz zwischen den beiden Modellen zu überprüfen. Um diese fehlende Konsistenzprüfung zu entwickeln, muss zunächst geklärt werden, welche Konsistenzregeln zwischen BPMN- und BROS-Modellen gelten. Da solche Konsistenzregeln auf unterschiedliche Weisen erstellt und formalisiert werden können, soll dabei besonders auf eine mögliche Automatisierung geachtet werden. Wichtig für diese Arbeit sind die Inkonsistenzen, die potentielle Widersprüche zwischen den Modellen darstellen. Die Automatisierung eines Konsistenzvergleiches ist ein wichtiger Aspekt in der Konsistenzprüfung, da eine manuelle Überprüfung nicht nur fehleranfälliger ist, sondern auch besonders bei großen Modellen sehr zeitaufwendig und damit auch kostspielig ist. Sollte eine automatische Überprüfung der Konsistenzregeln möglich sein, soll untersucht werden, wie eine mögliche Implementierung des automatisierten Konsistenzvergleiches realisiert werden kann. Dazu soll ein Tool entwickelt werden, das mit dem bestehenden BROS-Editor *FRaMED.io* kompatibel ist. Für das Tool ist es ausreichend, die Inkonsistenzen zu finden. Es ist nicht notwendig, eine Handlungsempfehlung für den Modellierer zu geben oder eine automatische Korrektur durchzuführen. Da in dieser Arbeit erstmals die Konsistenz zwischen BPMN und BROS untersucht wird, soll bei der Implementierung besonders auf die Modularität und Erweiterbarkeit auf neue Konsistenzaspekte geachtet werden. Zusammenfassend werden folgende Punkte analysiert:

1. Welche Konsistenzbeziehungen bestehen zwischen BPMN- und BROS-Modellen?
2. Wie lassen sich die Konsistenzbedingungen automatisiert überprüfen?
3. Mit welchem Aufwand ist dieses Verfahren erweiterbar?

1.3. Struktur der Arbeit

Um die Konsistenzbedingungen zwischen BPMN- und BROS-Modellen zu erstellen, werden im Kapitel 2 zunächst die Modellelemente, die in dieser Arbeit von BPMN und BROS genutzt werden, vorgestellt und jeweils mit einem Metamodell detaillierter beschrieben. Für den Einstieg in das Themengebiet des Konsistenzvergleiches, werden im Kapitel 3 aktuelle Konsistenzprüfungsverfahren für UML-Modelle analysiert. Diese werden anhand eines Schemas klassifiziert und im Hinblick auf ihre Anwendbarkeit zu weiteren Modellarten untersucht. Da BROS eine neue Modellierungssprache ist, existieren noch keine Konsistenzregeln zu anderen Modellen und damit auch nicht zu BPMN. Daher werden im Kapitel 4 Regeln aufgestellt, um die Konsistenz zwischen BPMN- und BROS-Modellen zu gewährleisten. Zur Erläuterung der Regeln wird jeweils ein Beispiel und eine formale Definition gegeben. Für die automatisierte Überprüfung der Konsistenzregeln wird eine Referenzarchitektur erstellt und im Kapitel 5 implementiert. Die Implementierung nutzt als Modellquellen die Editoren *bpmn.io* und *FRaMED.io*. Anschließend wird sie im Kapitel 6 anhand eines Beispiel-Use-Case evaluiert. Zusätzlich wird die Erweiterbarkeit der Implementierung um eine neue Regel demonstriert. Abschließend wird im Kapitel 7 der Erfolg der Arbeit diskutiert und ein Ausblick auf zukünftige Arbeiten auf diesem Gebiet gegeben.

2. Hintergrund

Um in der weiteren Arbeit den Konsistenzvergleich zwischen BPMN- und BROS-Modellen durchführen zu können, werden zunächst die Modelle vorgestellt. Für jedes Modell werden die Modellelemente eingeführt, die in der Arbeit unterstützt werden. Diese werden in ein Metamodell eingeordnet, das als Grundlage für den späteren Konsistenzvergleich dient. Da BROS auf der Modellierungssprache CROM basiert, wird zunächst CROM vorgestellt.

Es existieren Modellelemente, die sowohl in BPMN als auch in BROS den gleichen Namen haben (wie zum Beispiel "Event"). Um diese leichter unterscheiden zu können, werden die Modellelemente mit dem Präfix "BPMN" bzw. "BROS" benannt.

2.1. Business Process Model and Notation

Die *Business Process Model and Notation* (**BPMN**) ist ein weit verbreiteter Standard für die grafische Beschreibung von Geschäftsprozessen. Dabei wird das Verhalten eines Systems in einer, an Flussdiagramme angelehnten, Form beschrieben. Einer der Hauptvorteile von BPMN ist die große Verbreitung. Dadurch können Prozessbeschreibungen auf einfache Art und Weise zwischen verschieden Plattformen und Stakeholdern ausgetauscht werden. Neben der informellen Beschreibung von Prozessen unterstützt BPMN auch die direkte Ausführung von Geschäftsprozessen. Dafür existieren Interpretern, die in BPMN-Modellierte Abläufe zur Automatisierung einfacher Aufgaben oder ganzer Fertigungsprozesse genutzt werden können. Die Hauptelemente der BPMN, die auch in dieser Arbeit unterstützt werden, sind:

- **Activity:** Eine Activity beschreibt eine Tätigkeit, die innerhalb des Geschäftsprozesses ausgeführt wird. Da das Ausführen einer Aktion Zeit in Anspruch nehmen kann, kann auch die Abarbeitung einer Activity Zeit benötigen. Zur Darstellung einer Activity 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 zum Beispiel AND- und OR-Gateways, unterstützt. Je nach verwendetem Gateway, verläuft der weitere Kontrollfluss zum Beispiel parallel, oder nur einer der möglichen Wege wird genutzt. Dargestellt wird ein Gateway mittels eines um 45 Grad gedrehten Quadrates.

- **Event:** Ein Event beschreibt ein Ereignis, das innerhalb des Geschäftsprozesses auftreten kann. Es wird mit einem Kreis dargestellt. Events beeinflussen den Kontrollfluss, können ihn starten, beenden oder andere unabhängige Aktionen auslösen. Sie werden in drei verschiedene 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. Zusätzlich existiert noch das *TerminationEvent*, welches den laufenden Prozess vollständig abbricht und mit einem fast vollständig ausgefülltem EndEvent repräsentiert wird.

- **Flow:** Ein Flow ist eine gerichtete Verbindung zwischen Modellelementen. Dabei wird zwischen den *Sequence-* und den *MessageFlows* unterschieden. Ein *SequenceFlow* stellt den Kontrollfluss dar und verbindet Elemente, um eine Ausführungsreihenfolge festzulegen. Ein *MessageFlow* verbindet unterschiedliche Teilnehmer des Geschäftsprozesses und symbolisiert den Austausch von Mitteilungen. Beide Flows werden mit einem Pfeil dargestellt, wobei der *MessageFlow* gestrichelt ist.

- **Pool:** Ein Pool stellt eine Gruppe von zusammengehörenden Teilnehmern innerhalb eines Geschäftsprozesses dar. Wenn ein Prozess nur aus einem Pool besteht, kann der Pool und der Prozess als ein Element dargestellt werden. 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. Sollte der Pool dabei auch gleichzeitig den Prozess darstellen, so kann die Swimlane als eigenständiger Prozess betrachtet werden. Innerhalb eines Pools wird die Simelane als eine horizontale Bahn dargestellt.

- **Process:** Ein Process bildet einen Teil des Geschäftsprozesses ab und ist das Containerelement für die anderen Modellelemente. An dem vollständigen Geschäftsprozess können mehrere Prozesse beteiligt sein, die mittels *MessageFlows* untereinader kommunizieren. Ein Teilprozess kann, wie bereits beschieben, als Pool bzw. Swimlane oder auch als Activity dargestellt werden.

Da BPMN schon über einem Jahrzehnt in Benutzung ist, existiert eine gute Toolunterstützung für die Modellierung. Eines dieser Tools ist *bpmn.io*[1]. Hierbei handelt es sich um 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*. Diese Datenstruktur basiert auf dem Metamodell, das von der *Object Management Group*[2] spezifiziert wurde.

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

Mit dem BPMN-Metamodell aus Abbildung 2.1 wird ein BPMN-Modell in einem Graphen dargestellt. Die Grundstruktur bildet dabei ein Baum, der den hierarchische Aufbau des Modelles abbildet. Zusätzlich wird der Baum mit Querverweisen angereichert, welche die Relationen darstellen. Die Wurzel des Baumes ist das Modell an sich. Dieses kann mehrere Prozesse (BpmnProcess) enthalten. Jeder Prozess beinhaltet verschiedene Flowobjekte. Dazu zählen Events (BpmnEvent), Gateways (BpmnGateway) und Activities (BpmnTask). Die Events

[1] https://demo.bpmn.io/
[2] https://www.omg.org/spec/BPMN/2.0/About-BPMN/

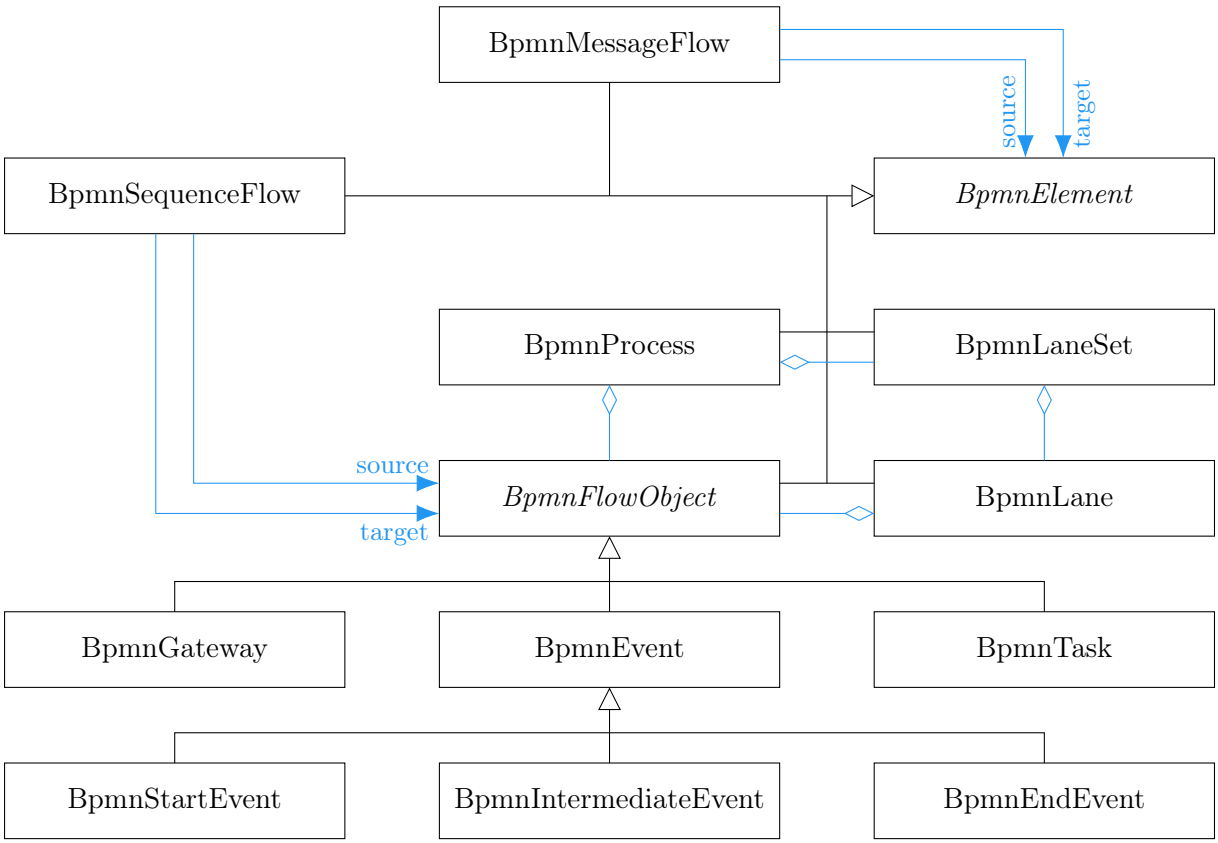


Abbildung 2.1.: BPMN Metamodell

teilen sich in die drei Unterklassen auf: Start-, Intermediate- und EndEvents (BpmnStartEvent, BpmnIntermediateEvent, BpmnEndEvent). Außerdem kann ein Prozess mehrere Pools (BpmnLaneSet) und Swimlanes (BpmnLane) enthalten. Hierbei ist zu beachten, dass jedes Flowobjekt nur einen Container haben darf, obwohl Swimlanes und Processes Flowobjekte enthalten können. So kann ein Flowobjekt nicht gleichzeitig Kind eines Prozesses und einer Swimlane sein. Jedes Kind einer Swimlane ist transitiv auch Kind von dem Elternprozess der Swimlane. Die beiden Flow-Arten bilden die Querverweise innerhalb des Baumes. Ein MessageFlow (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 das *Compartment* ein, welches den Kontext der Rolle abbildet, in dem diese gespielt werden kann. Dadurch 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 den Compartments die Kontextabhängigkeit modelliert. Die dafür genutzten Elemente sind:

- **RoleType:** Eine RoleType ist die Darstellung einer Rolle. Sie wird mit einem abgerundeten Rechteck dargestellt, das, vergleichbar mit einer UML-Klasse, in drei Bereiche unterteilt ist. In den Bereichen werden der Name, die Attribute und die Methoden der Rolle abgebildet. Einer Entität, die diese Rolle spielt, werden eben jene Attribute und Methoden hinzufügt.

- **CompartmentType:** Ein CompartmentType bildet den Kontext von verschiedenen 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 des Modelles dar. Sie unterscheiden sich nur in der Semantik. Der NaturalType bildet häufig eine natürliche Person ab, wohingegen der DataType oft 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 Annotation ein Constraint oder auch eine Verbindung zwischen diesen beschreiben.

- **Package:** Packages helfen 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 Editor. Der Name des Editors ist ein Akronym für *Full-fledged Role Modeling EDitor*. Dieser wurde speziell für CROM entwickelt. Er unterstützt den vollständigen Standard von CROM. Unter zu Hilfename eines *Feature Editor* lassen sich bestimmte Modellfunktionen ein- und ausschalten.

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 u.a. mit Hilfe von Events um den Verhaltensaspekt erweitert. Ziel ist es, einen bereits existierenden Geschäftsprozess wiederzuverwenden, um auf dessen Basis, die Businessobjekte zu modellieren. Das so entstandene Modell kann anschließend als Vorlage für die eigentliche Implementierung genutzt werden (vgl Schön et al. 2019). Dies hat den Vorteil, dass obwohl es sich um ein Strukturmodell handelt, weiterhin der Ablauf des Prozesses erkennbar ist. Damit ist leichter erkennbar, wie Objekte miteinander interagieren. Um dies zu ermöglichen, wird der CROM-Standard um folgende Modellelemente erweitert:

- **Scene:** Eine Scene stellt einen Teilnehmer innerhalb eines Prozesses dar. Sie besitzt Methoden und kann Rollen, Events und andere Scenen beinhalten. Dargestellt wird sie mit einem Rechteck mit doppeltem linken 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.

[3] https://github.com/Eden-06/FRaMED-2.0

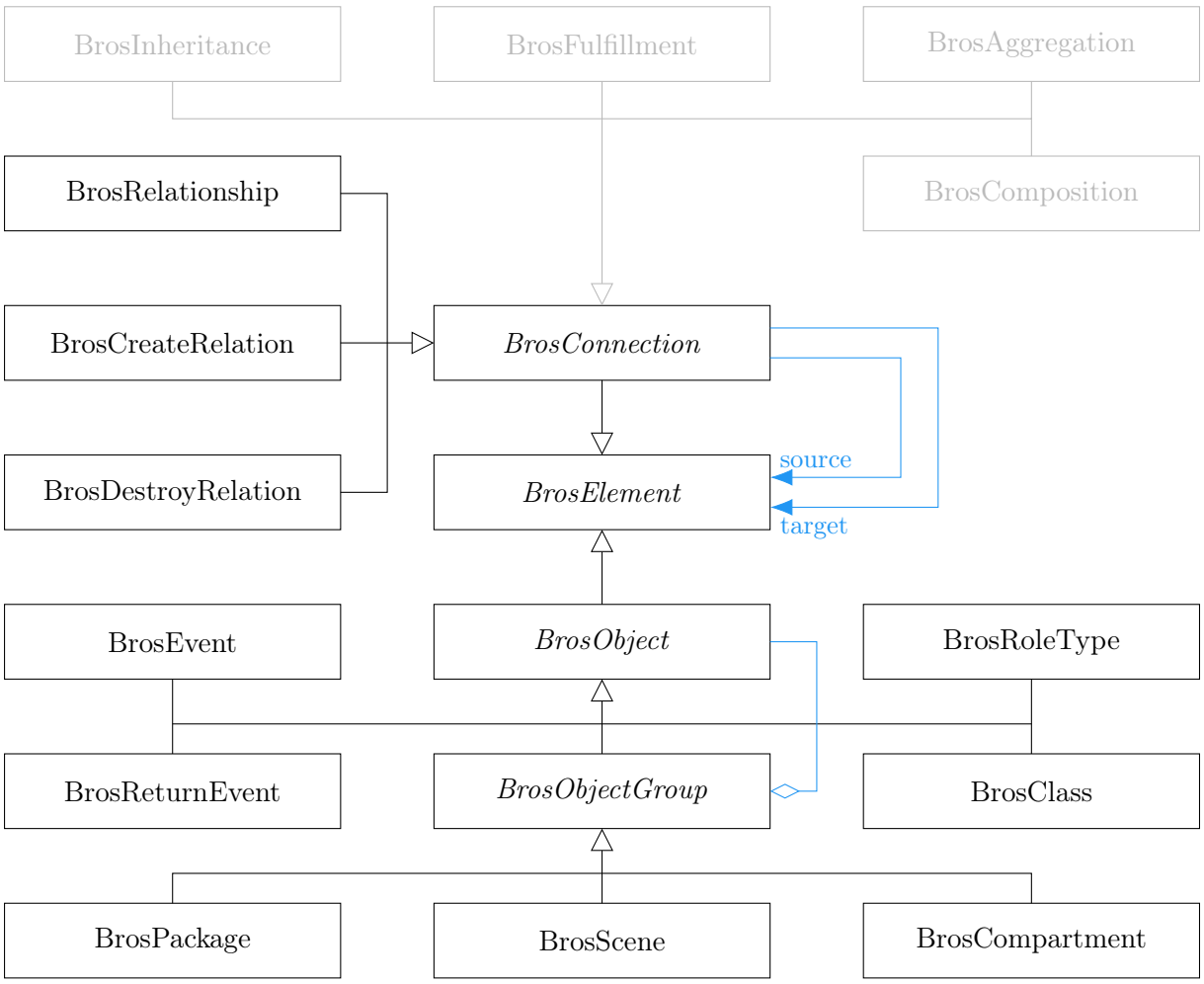


Abbildung 2.2.: BROS Metamodell

- **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, innerhalb des Geschäftsprozesses, verantwortlich ist. Eine CreateRelation ist von einem Event zu einer Rolle bzw. 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] einen graphischen Editor. *FRaMED.io* ist eine Neuentwicklung von *FRaMED-2.0* und wie *bpmn.io* webbasiert. Damit ist es plattformübergreifend und ohne aufwendige Installation nutzbar. Es unterstützt das Erstellen von CROM- und BROS-Modellen, ist aber nicht mit existierenden *FRaMED-2.0*-Projekten kompatibel.

Das Metamodell in Abbildung 2.2 basiert auf dem von Schön et al. 2019 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-, DestroyRelation und die Relationship (BrosCreateRelation, BrosDestroyRelation, BrosRelationship) gehören. Dazu zählen auch die vier Connections im oberen Bereich des Metamodelles (BrosInheritance,

[4]https://eden-06.github.io/FRaMED-io/

BrosFulfillment, BrosAggregation, BrosComposition). Diese sind für die weitere Arbeit nicht relevant und werden deshalb ausgegraut dargestellt. Da diese aber auch häufig für die Modellierung verwendet werden können, sind sie im Metamodell mit aufgeführt. Auf der anderen Seite sind die Objekte, die sich in Endknoten und Objektgruppen teilen. Endknoten sind die Natural- und DataTypes (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 Objektgruppe sein und die Eltern-Kind-Beziehung ist transitiv. Die Connections bilden die Querverweise innerhalb des Baumes.

Zusätzlich dazu existieren noch weitere Beschränkungen innerhalb des Metamodelles, die nicht mit diesen dargestellt werden können. Beispielsweise kann ein BrosCompartment nicht Bestandteil einer BrosScene sein. Für die Einhaltung dieser Beschränkungen ist der Modellierer verantwortlich. Allerdings kann die Einhaltung der Beschränkungen auch automatisiert überprüft werden. So unterstützt der BROS-Editor *FRaMED-io* diese Bedingungen und erlaubt nur das Erstellen von validen Modellen.

3. Verwandte Arbeiten

Der Bereich der Konsistenzprüfung zwischen strukturbasierten und verhaltensbasierten Modellen wird seit Jahren intensiv erforscht. Insbesondere gibt es ein großes Spektrum an Methoden zum Vergleichen von verschiedenen UML-Modellarten. 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. Um die bestehenden Verfahren zur Konsistenzprüfung zu analysieren, wird zunächst eine Klassifikation für diese Verfahren vorgestellt. Anhand dieses Schemas werden anschließend relevante Arbeiten und deren Verfahren erläutert und verglichen.

3.1. Klassifikation von Verfahren zur Konsistenzprüfung

Da es bereits etliche Arbeiten im Gebiet der Konsistenzprüfung von UML-Modellen gibt, wurden bereits mehrere Zusammenstellungen der bestehenden Verfahren durchgeführt. In dieser Arbeit werden das Survey von Usman et al. 2008 und die Review von Lucas et al. 2009 genauer betrachtet. Beide Autoren haben für die Klassifikation der Arbeiten je ein Schema erstellt, die sich in vielen Punkten ähneln. Für die weitere Arbeit wird eine erweiterte Kombination der beiden Klassifikationsschemata genutzt. Jeder Punkt, der direkt aus einer der beiden bestehenden Arbeiten entnommen wurde, ist mit dem Autor, der ihn zuerst genannt hat, annotiert.

- **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-Komponentendiagramme 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 oder beides.
- **Diagrams (Usman et al. 2008):** Es beschreibt, welche konkreten Modellarten von der Methode unterstützt werden. Die Arbeiten von Usman et al. 2008 und Lucas et al. 2009 beziehen sich dabei ausschließlich auf UML Modelle. Die Verfahren müssen dabei aber nicht auf UML beschränkt sein. Mögliche Werte sind die Namen der unterstützten Modelltypen.
- **Consistency Type (Usman et al. 2008):** Es beschreibt, welche Arten der Konsistenz von der Methode überprüft werden. Dabei wird unterschieden zwischen der

Inter-Modell (vertikale) Konsistenz, der *Intra-Modell (horizontale) Konsistenz* und der *Evolutionskonsistenz*. Zusätzlich spezifiziert Usman et al. 2008 noch die *syntaktische* und die *semantische Konsistenz*. Diese beiden Konsistenzarten werden für den späteren Konsistenzvergleich als Voraussetzung angesehen und nicht weiter untersucht. Mögliche Werte sind damit die drei oberen Konsistenzarten.

- **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 formellen Algorithmus basierend), *Monitoring* (auf der Überprüfung eines 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 von ihrem Autor 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 mit geringem Aufwand (H), mit mittlerem Aufwand (M) und mit hohem 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), mittlerer Aufwand (M) und hoher Aufwand (L).

- **Rule Extensibility:** Es beschreibt, wie gut die Methode um weitere Konsistenzregeln innerhalb der unterstützten Modellarten erweiterbar ist. Mögliche Werte sind geringer Aufwand (H), mittlerer Aufwand (M) und hoher Aufwand (L).

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

Die *Intra-Modell Konsistenz* beschreibt die Konsistenz zwischen verschiedenen Modellarten, die sich auf der gleichen Abstraktions- und Evolutionsstufe befinden. Dahingegen befasst sich die *Inter-Modell Konsistenz* mit der Konsistenz eines Modelltyps auf unterschiedlichen Abstraktionsstufen. Die Konsistenz zwischen verschiedenen Evolutionsstufen wird von der *Evolutionskonsistenz* untersucht. Diese Überprüfung erfolgt innerhalb einer Modellart auf der gleichen Abstraktionsstufe. Mit diesen drei Konsistenzarten kann die Konsistenz der gesamten Modellierung überprüft werden. Die Voraussetzung dafür ist, dass die zu untersuchenden Modelle korrekt sind. Für diese Überprüfung sind die *syntaktische* und die *semantische Konsistenz* verantwortlich. Dabei untersucht die *syntaktische Konsistenz* die Korrektheit eines Modelles zu seinem Metamodell, wohingegen die *semantische Konsistenz* die Korrektheit von zusätzlichen Beschränkungen überprüft. Wie bereits erwähnt, ist diese Korrektheit der Modelle für die weitere Arbeit eine Voraussetzung. Methoden, die diese Konsistenz überprüfen, werden nicht weiter untersucht. Zudem existieren bereits Tools die diese Überprüfung sowohl für

BPMN (zum Beispiel *bpmn.io*) als auch für BROS (*FRaMED-io*) automatisiert durchführen können.

3.2. Aktuelle Verfahren zur Konsistenzprüfung

Anhand des aufgestellten Klassifikationsschemas werden in diesem Abschnitt verschiedene Verfahren der Konsistenzprüfung von UML-Modellen vorgestellt. Da die Konsistenzprüfung zwischen BPMN und BROS beide Modelltypen erfordert, werden dabei 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, dass 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.

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 verfolgt. Noch weiter gehen Satoh et al. (Usman et al. 2008, 15), indem UML-Klassendiagramme direkt in Logikprogramme überführt und auf Erfüllbarkeit überprüft 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-Modellarten beschrieben. Dieses iterative Vorgehen reduziert den Entwicklungsaufwand erheblich, da nur noch ein Bruchteil aller benötigten Transformationsalgorithmen entwickelt werden müssen. Dies vereinfacht auch die Erweiterbarkeit um neue Modellarten. Die konstruierten Modelle können anschließend direkt mittels *Inter-Modell (vertikale) Konsistenz* überprüft werden. Einige der beschrieben Transformationsalgorithmen sind bereits implementiert worden. 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 etabliert, existieren zwei verschiedene Aspekte der Erweiterbarkeit, die Modellerweiterbarkeit und

	Diagrams	Consistency Type	Consistency Strategy	Intermediate Representation	Case Study	Automatable	Tool Support	Model Extensibility	Rule Extensibility
Rasch 2003	CD, SM	Intra	Monitoring	CSP/OZ	✓	●	✗	●	◐
Shinkawa 2006	UCD, CD, SD, AD, SC	Inter	Analysis	CPN	✗	●	✗	◐	○
Mens 2005	CD, SD, SC	All	Monitoring	Extended UML	✓	●	✓	●	◐
Egyed 2001	CD, OD, SD	Intra, Inter	Construction		✗	●	∼	◐	◐
Egyed 2006	CD, SD, SC	Intra	Monitoring		✓	●	✓	○	◐

●=mit geringem Aufwand; ◐=mit mittlerem Aufwand; ○=mit hohem Aufwand;
✓=ja; ✗=nein; ∼=teilweise

Tabelle 3.1.: Vergleich der bestehenden Verfahren

die Regelerweiterbarkeit. Die Modellerweiterbarkeit beschreibt mit welchem Aufwand 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. Hingegen haben die Verfahren ohne Zwischendarstellung eine bessere Regelerweiterbarkeit.

Anhand der Zwischendarstellung lassen sich die beschriebenen Verfahren in zwei Kategorien einteilen. Zum einen, in die, die eine Zwischendarstellung benötigen und zum anderen, 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 können die Verfahren von Egyed 2001 und Egyed 2006 direkt auf den Modellen arbeiten. Der Vorteil einer Zwischendarstellung ist, die leichte Erweiterbarkeit um neue Modellarten. Es muss nur eine neue Konvertierungsmethode des neuen Modelles zur Zwischendarstellung entwickelt werden. Anschließend können die gleichen Algorithmen zur Konsistenzprüfung verwendet werden, die bereits bei den bestehenden Modellen Anwendung finden. 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, dass eine Modellart nicht kompatibel zu der genutzten Zwischendarstellung ist. Dann ist eine Erweiterung um diesen Modelltyp nicht möglich. Dahingegen haben die Verfahren ohne Zwischendarstellung weniger Beschränkungen bei der Erweiterbarkeit um weitere 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, wobei n die Anzahl, der bereits unterstützten Modellarten ist. 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 zu der Methode der *Construction*. 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 zwingend 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 Modellarten 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. Anders als bei UML, ist bei dem Vergleich von BPMN und BROS zu beachten, dass Inkonsistenzen keine strikten Fehler, sondern nur Warnungen an den Modellierer sind. Das liegt an dem Funktionsumfang von BROS Modellen. Diese können gegenüber den BPMN-Modellen mit zusätzlich Elementen angereichert werden. So kann das BROS Modell zusätzliche Elemente beinhalten oder Teile des BPMN-Modells auf einer höheren Ebene der Abstraktion darstellen. Beispielsweise kann ein BROS-Event die Abstraktion eines gesamten Prozesses innerhalb des BPMN-Modelles abbilden. Um diese Konsistenzbeziehung genauer untersuchen zu können, muss zunächst geklärt werden, wie die Konsistenz zwischen Methoden definiert ist. Dafür wird eine Definition des Konsistenzproblems gegeben. Anhand dieser, 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 Softwaresysteme stark von den dazugehörigen Modellen abhängig. 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 gesamten Projektes führen. Die verhaltensbasierten 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 ordnungsgemäß funktionieren. Solche Fehler stellen daher in der Industrie ein großes Risiko dar und können die Entwicklungskosten erheblich erhöhen. Um dieses Risiko zu verringern, müssen Verfahren entworfen werden, die eine möglichst automatisierte Konsistenzprüfung durchführen können.

Die methodische Entwicklung eines solchen Verfahrens setzt eine genaue Definition des Konsistenzproblems voraus. Allerdings ist eine formale und allgemeingültige Definition des Konsistenzproblems, in der Domain der Modellierung in der Literatur, nur schwer zu finden. Bevor das Konsistenzproblem definiert werden kann, muss zunächst beschieben 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 Sprache oder

in einer formalen Notation beschieben werden, wobei nur letzteres automatisiert überprüft werden kann. Für die Definition nach Nuseibeh 1996 wird das Vorhandensein eines solchen Regelsatzes vorrausgesetzt. 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.*

Somit muss für die Entwicklung eines Konsistenzverfahrens zunächst definiert werden, welche Inkonsistenzen zwischen den Modellen existieren können und damit auf welchem Regelsatz das Verfahren basiert. Anhand des aufgestellten Regelsatzes kann ein Algorithmus zum Auffinden der Inkonsistenzen erstellt werden. Dafür müssen die Regeln, wie bereits erwähnt, in einer formalen Form und nicht in natürlicher Sprache vorliegen. Die Aufgabe der Minimierung und Verhinderung der Inkonsistenz, ist auch mit einem automatisierten Tool, dem Modellierer überlassen. Die Verfahren können dem Modellierer Handlungsempfehlungen geben, eine automatische Korrektur ist allerdings nur schwer möglich. So könnte ein Problem zum Beispiel durch das Entfernen eines Elementes behoben werden, wodurch kaskadierend nochmehr Konsistenzfehler entstehen könnten. Da diese Definition auf der Überprüfung von Regeln aufbaut, ist sie leicht auf Verfahren anwendbar, die für die Konsistenzprüfung, ebenfalls auf einem Regelsatz basieren. Aber auch formale Verfahren ohne einen eigenen Regelsatz, lassen sich mit dieser Definition beschreiben. So können zum Beispiel 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. Regeln zum direkten Vergleich der Modelle können in zwei verschiedene Richtungen definiert werden. Eine Regel kann aufgrund eines Merkmales 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 Regeln 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 die 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 jeweiligen Typen zugeordnet. Der Fakt *relation/3* bildet eine Relation zwischen zwei Modellelementen von Quelle nach Ziel ab und ordnet dieser Relation gleichzeitig einem 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, dass es sich bei den Graphen der BPMN- und BROS-Modelle um Bäume handelt. Allgemein ist zu beachten, dass 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

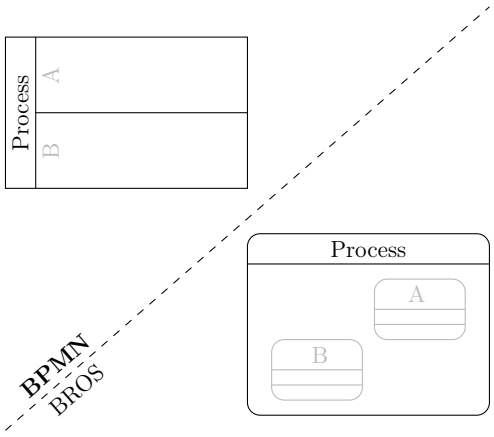
Zusätzlich werden noch zwei Hilfsprädikate definiert. Das Prädikat *transitive_parent/2* bildet den transitiven Abschluss der *parent/2* Relation. Jedem Kind werden all seine transitiven Eltern zugeordnet. Dies beinhaltet auch die Abbildung der Identität. Das wichtigste Prädikat ist *match/2*. Es bildet eine Zuordnungen von BPMN-Elementen zu den BROS-Elementen und umgekehrt ab. Zwei Elemente haben ein Matching zueinander, wenn sie auf dem semantischen Ebene die gleiche Bedeutung haben. Der genaue Aufbau des Matching ist dabei implementierungsabhängig. Es kann zum Beispiel auf den Namen der Elemente oder auch deren Struktur basieren. Im folgenden wird angenommen, dass ein Matching bereits existiert, wodurch es im weiteren auch als Orakelfunktion bezeichnet wird. Das 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).
```

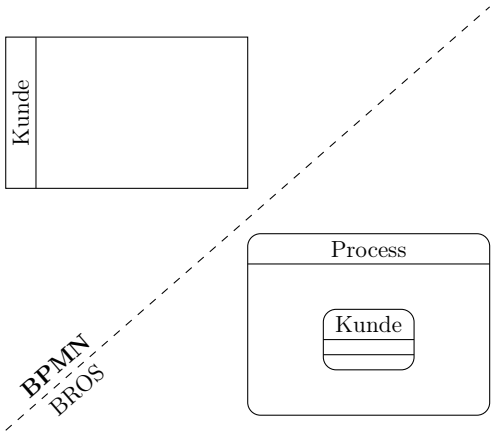
Quelltext 2: Definitionen der weiterführenden Regeln

Mit der Hilfe dieser Prädikate können nun Regeln für die Konsistenz zwischen BPMN- und BROS-Modellen erstellt werden. Dafür wird jede Regel erläutert und an einem Minimalbeispiel visualisiert. Die darauffolgende Formalisierung in Prolog dient als Referenz für die weitere Arbeit.

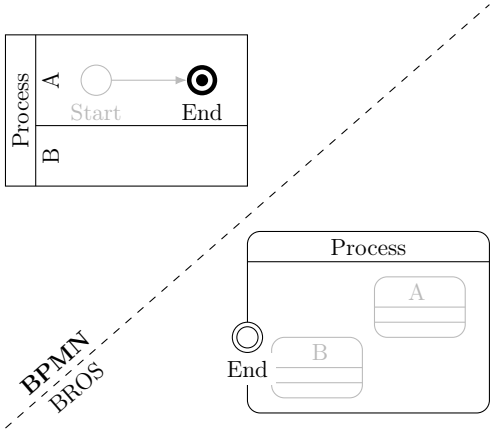
Regel 1: Zu jedem *BPMN-Process* muss eine *BROS-Scene* oder ein *BROS-Event* existieren. Ein BPMN-Process stellt eine Umgebung für einen natürlichen Prozess dar. In BROS wird dies mit einer BROS-Scene repräsentiert. Allerdings kann ein BROS-Modell eine Abstraktion eines BPMN-Modelles sein. In diesen 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. Dies ist exemplarisch in Abbildung 4.1a für eine BROS-Scene dargestellt.



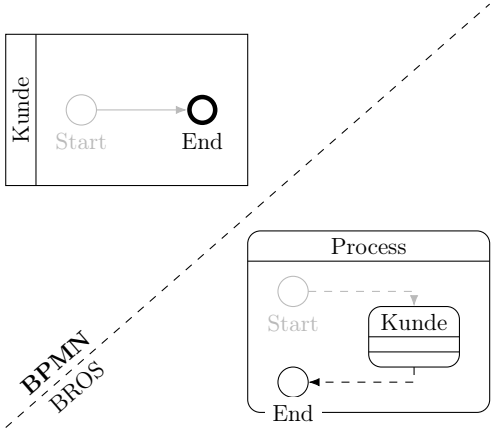
(a) Darstellungen der Regel 1



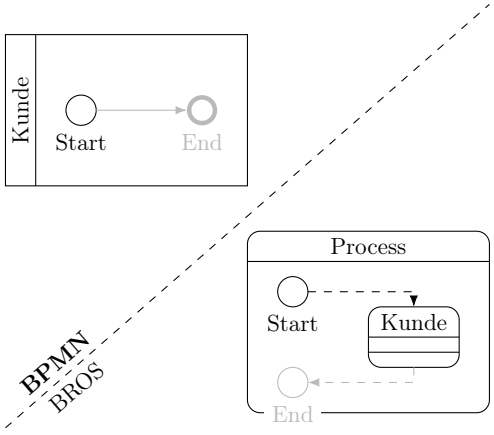
(b) Darstellungen der Regel 2



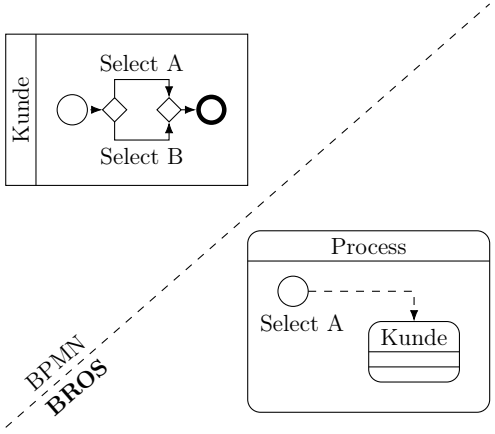
(c) Darstellungen der Regel 3



(d) Darstellungen der Regel 4



(e) Darstellungen der Regel 5



(f) Darstellungen der Regel 6

Abbildung 4.1.: Exemplarische Darstellungen der Konsistenzregeln 1-6

In beiden Modellen muss der Prozess enthalten sein, der Inhalt und die Relationen werden nicht betrachtet.

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

Quelltext 3: Formalisierung der 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. In Abbildung 4.1b existiert in dem BPMN-Modell eine BPMN-Swimlane "Kunde", welche im BROS-Modell als BROS-RoleType "Kunde" dargestellt wird.

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

Quelltext 4: Formalisierung der 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. Diese Regel wird in Abbildung 4.1c visualisiert.

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

Quelltext 5: Formalisierung der 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. Der erste Fall ist mit Hilfe des Prozessteilnehmers "Kunde" in Abbildung 4.1d beispielhaft dargestellt.

```
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: Formalisierung der 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 eine BROS-CreateRelation zwischen dem BROS-Event und der BROS-Scene existiert, die den BROS-Prozess darstellt (vgl. **Regel 1**). Diese Regel entspricht, im Sinne des Ablaufes des Geschäftsprozesses, dem Gegenstück zu **Regel 4**. Die Visualisierung nutzt das gleiche Beispiel, nur ist in Abbildung 4.1e der Fokus auf das BPMN-StartEvent und die BROS-CreateRelation gelegt.

```
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: Formalisierung der 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. Beispielhaft ist diese Regel in Abbildung 4.1f dargestellt. Hier wird der Fall abgebildet, das ein BROS-Event mit einem BPMN-Gateway übereinstimmt.

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

Quelltext 8: Formalisierung der Regel 6

4.3. Referenzarchitektur

Im Gegensatz zu anderen Arbeiten wurde zur Überprüfung dieser Regeln kein formales Verfahren auf Basis von zum Beispiel *Description Logic* oder *Petrinetzen* genutzt. Dies hat den Vorteil, dass die Regeln direkt auf den Modellen ausgeführt werden können und nicht erst eine Zwischendarstellung konstruiert werden muss. Das hier genutzte Verfahren arbeitet in zwei Stufen auf den Modellen, die als geschichteter Graph dargestellt werden und ist in Abbildung 4.2 dargestellt. Im ersten Schritt wird ein Matching von Modellelementen aufgebaut. Dies wird iterativ in Form eines Fixpunkt-Algorithmus durchgeführt, um ein kaskadierendes Matching zu erlauben. Dieser Schritt wird im folgenden *Matching-Algorithmus* genannt. Im zweiten Schritt werden anhand dem Matching die erstellten Regeln ausgeführt. Dabei werden die Regelergebnisse aggregiert. Dieser Vorgang wird als *Verifikationsalgorithmus* bezeichnet.

Um das Verfahren anzuwenden, müssen beiden Modelle in Form eines gerichteten Graph vorliegen, der aus mehreren Ebenen besteht. 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 eine Schicht mit Querverweisen über den Baum gelegt. In Abbildung 4.3 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 *contains*-Relation der Metamodelle direkt aufgebaut. In Blau ist die zweite Schicht visualisiert, welche die Relationen beinhaltet. Jede Relation ist mit ihrem Typ annotiert und hat eine feste Richtung. Die Graphen des BPMN-Modelles und des BROS-Modelles sind zueinander disjunkt.

Matching-Algorithmus

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

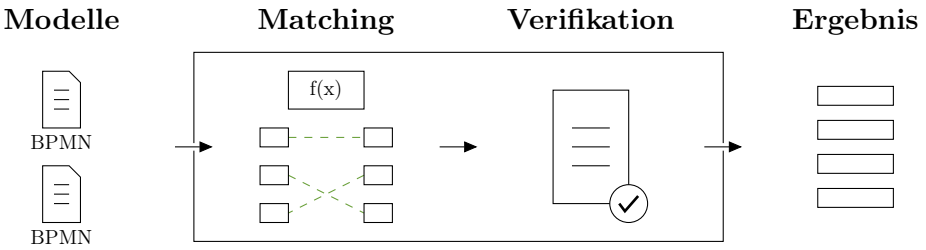


Abbildung 4.2.: Referenzarchitektur

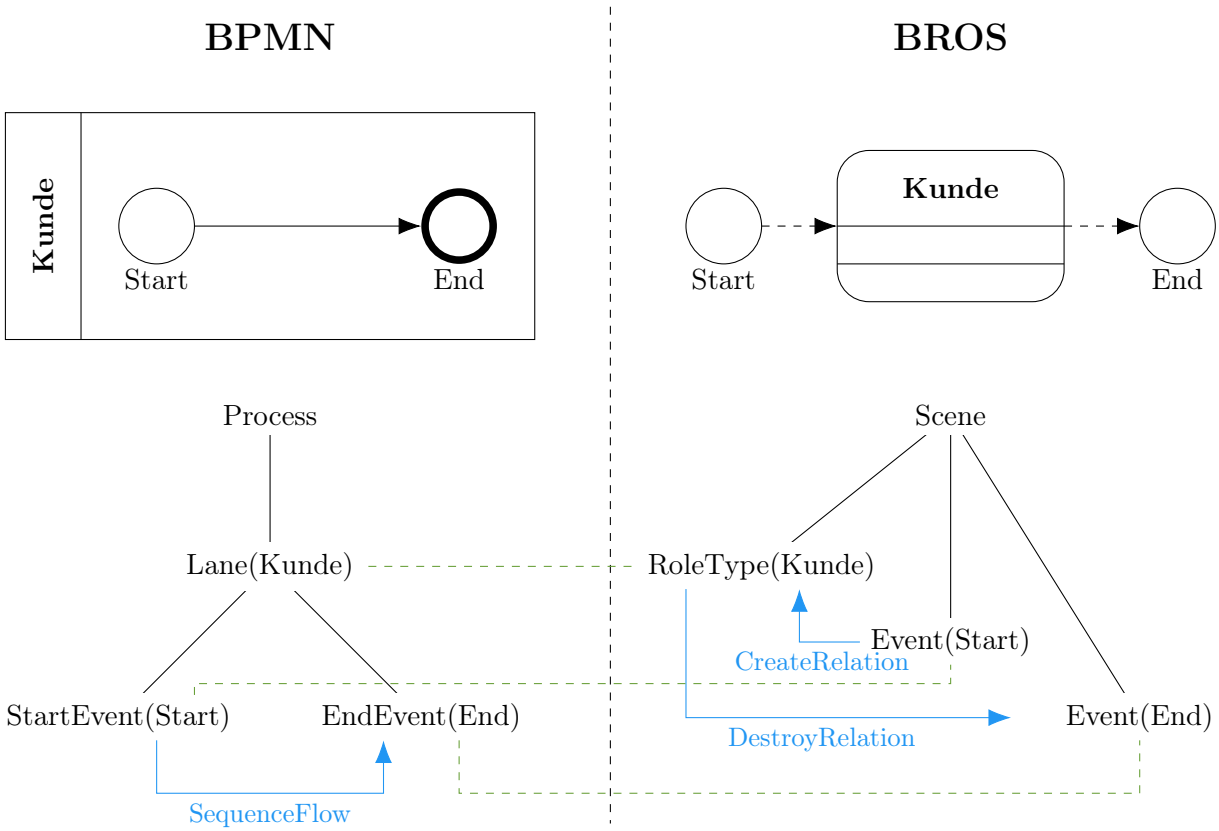


Abbildung 4.3.: Visualisierung des Graph Matching

dem Graphen des BROS-Modelles verbindet. In Abbildung 4.3 wird dies in grün dargestellt. Um diese Schicht zu konstruieren, wird eine Orakelfunktion genutzt. Diese ermittelt ob zwei Elemente zueinander passend sind. Dafür hat die Orakelfunktion zwei Parameter, den aktuellen Konten 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 mittels Relationen 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 hinzugefügte Kante, kann nicht wieder entfernt werden. Da die dritte Schicht auch mit an das Orakel übergeben wird, kann dieses aufgrund von einem bereits bestehenden Matching eine Entscheidung treffen. Um dieses Verhalten nutzen zu können, wird der Vorgang des Matchingaufbaues 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 beginnt zwischen Zuständen zu oszillieren. Dies wird verhindert, indem Kanten nur zu dem Matching hinzugefügt 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.

Verifikationsalgorithmus

Mit dem konstruierten Matching kann nun der Verifikationsalgorithmus 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 Hauptteil des Matching-Algorithmus wird der Verifikationsalgorithmus nur einmal durchlaufen. Da alle Regeln die Konsistenzprüfung von einem Quellmodell zu einem Zielmodell 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 verletzten Regel, kann eine genaue Fehlerbeschreibung gegeben werden. Dieses Verhalten ist analog für die Regeln, die aus Sicht des BROS-Modelles arbeiten.

Nicht mit den Prolog-Regeln 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 eine erfolgreiche Regelprüfung in zwei Fälle unterschieden werden. Eine Regel wird nicht nur als erfolgreich ausgeführt markiert, wenn sie tatsächlich erfolgreich war, sondern auch, wenn sie bei einer falschen Vorbedingung vorzeitig abgebrochen wurde. Dieses Verhalten ergibt sich aus den Prolog-Definitionen. Die Implikation gibt wahr zurück, wenn entweder die Vorbedingung falsch ist oder die Konsequenz wahr ist. Interessant für den Modellierer ist nur der Fall, dass die Konsequenz wahr ist. Dabei wird eine positive Konsistenzmeldung gespeichert. Bei einer falschen Vorbedingung kann die Ausführung 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 gefunden 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 auszuwerten. 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 *Verifikationsalgorithmus* beschrieben. Im vierten Abschnitt wird die Benutzerschnittstelle des implementierten Tools vorgestellt. Das Tool ist auf *GitHub* veröffentlicht[1] und kann direkt ausgeführt werden.

5.1. Implementierung der Referenzarchitektur

Im Abschnitt 4.2 wurden die Konsistenzregeln mit Hilfe von einem Prolog-Dialekt eingeführt. Dies legt nah, 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 *Verifikationsalgorithmus* 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 oder LLVM-Bytecode transpiliert werden. Dadurch besitzen Kotlin Programme eine hohe Erweiterbarkeit im Bezug auf weitere Laufzeitumgebungen. Wie auch im BROS-Editor FRaMED.io wird die Transpilierung zu Javascript genutzt, wodurch das Tool als eine Webanwendung entwickelt wird. 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-Namespaces aufgeteilt. Die Struktur wird im Namespace *bpmn*, und die graphische Darstellung des Modelles im Namespace *bpmndi* gespeichert. Für die Konsistenzprüfung ist nur der Namespace *bpmn* relevant. Der Namespace

[1] https://pixix4.github.io/bpmn-bros-verifier/

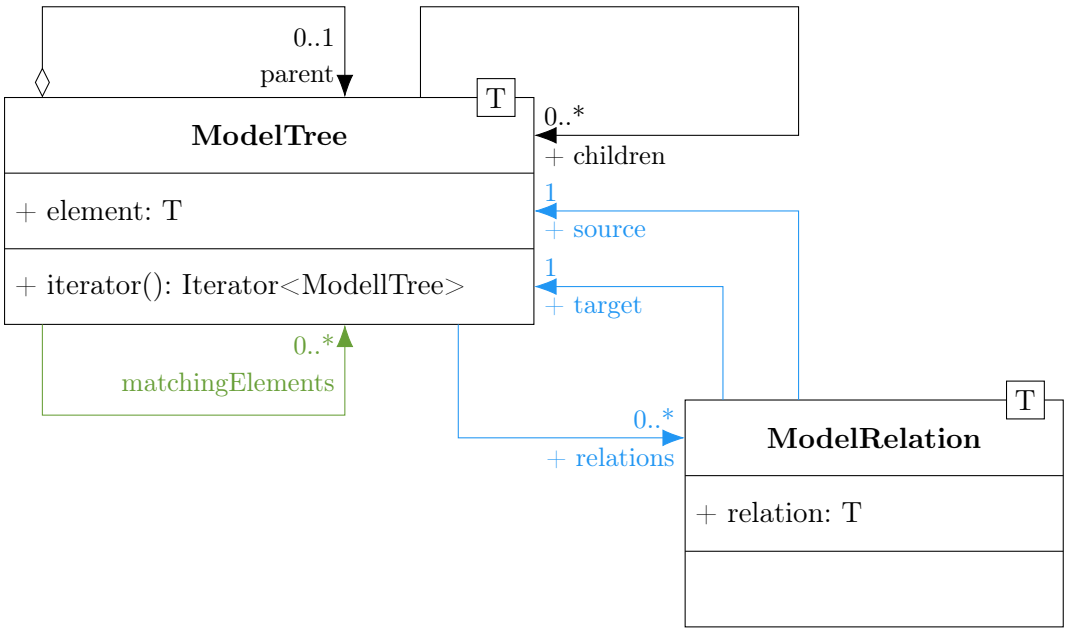


Abbildung 5.1.: Metamodell der Graphstruktur

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, da er in Javascript geschrieben ist und somit eine starke Bindung an die Plattform Javascript erzwingt. Die integrierte Validierungsfunktion für BPMN-Modelle wird auch nicht benötigt da eine Vorraussetzung 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 *Verifikationsalgorithmus* 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 besitzt außerdem einen Verweis auf seinen Eltern-Knoten. Zusätzlich hat 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. In dieser Schicht referenziert jeder Knoten die Relationen, die eine Verbindung mit ihm haben. Jede Relation besitzt einen Verweis auf seine Quell- und Zielknoten. Auch die Graphrelationen besitzen eine Referenz auf ihre Metamodell-Instanz mit Typannotation. Schließlich ist die dritte Schicht, die das Matching zwischen den Modellen darstellt, mit Grün markiert. Dabei handelt es sich um ein Set von Knoten, die nach der Orakelfunktion zu ihm passend sind. Jeder Knoten hat zusätzlich noch eine Hilfsfunktion, die einen Iterator zurückgibt, der eine einfache Breitensuche über den

'Aktion war erfolgreich' , 'ErfolgreicheAktion' (5.1)

{'aktion', 'erfolgreich', 'war'} , {'aktion', 'erfolgreiche'} (5.2)

{'aktion', 'erfolgreiche'} , {'aktion', 'erfolgreich', 'war'} (5.3)

{'**akti**on' ⊆ '**akti**on'} , {'**erfolgreic**he' ⊆ '**erfolgrei**ch'} (5.4)

Abbildung 5.2.: Anwendung des Name-Matching

Graphen implementiert.

5.2. Matching der Modelelemente

Um die Verifizierung der Modelle zu ermöglichen, muss zunächst ein dazugehöriges Matching aufgebaut werden. Dafür benötigt der *Matching-Algorithmus* eine Implementierung der Orakelfunktion. Bei der Implementierung 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 Namen in Form von 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, dass 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 implementiert dafür das *Matching*-Interface. Um diese Regeln gebündelt zu erstellen, und zu sammeln, gibt es die Hilfsklasse *Context*. Diese 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 eine anonyme Funktion. Diese werden in Kotlin mittels Lambdas dargestellt. Auf syntaktischer Ebene kann ein Lambda, sofern es der letzte Parameter einer Funktion ist, außerhalb der Parameterliste geschrieben werden. Dies erleichtert die Lesbarkeit des Quellcodes, da er sich optisch einer domainspezifischen Sprache ähnelt. Sie bildet die beiden Knoten des BPMN-Graphen und des BROS-Graphen auf einen Wahrheitswert ab. Innerhalb des Lambdas kann beliebiger Code ausgeführt werden, um die beiden Elemente zu vergleichen. Im einfachsten Fall führt das Lambda nur das Name-Matching der beiden Elemente aus. Es kann allerdings auch beliebige weitere Merkmale der Elemente, wie deren Struktur, zum Vergleich nutzen. Dabei hat sie auf alle drei Schichten des Graphen Zugriff. 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. Da der Graph gerichtet ist, das Matching allerdings bidirektional ist, müssen zwei Kanten für die Hin- und Rückrichtung hinzugefügt werden. Sonst wird das Ergebnis ignoriert, da nur Kanten zum Matching hinzugefügt werden dürfen. Das Lambda wird nur ausgeführt, wenn die Knoten zu den Filtern der ersten beiden Parameter passen. In Quelltext 9 ist das Matching für eine BPMN-Swimlane und einem BROS-RoleType gegeben. Mittels der generischen Parameter werden die Graph-Knoten gefiltert und das Lambda wird nur auf den beiden Knotentypen ausgeführt. Im Lambda wird ein einfaches Name-Matching durchgeführt und 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, dass die Namen der BPMN-Gateway Ausgänge nicht im Metamodell des Gateways, sondern im Metamodell der weiterführenden BPMN-SequenceFlows 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 zurückgibt.

```
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, dass ein BPMN-Event, ein äquivalentes Matching, wie seine per BPMN-MessageFlow verbundenen Nachbarn, besitzt. Nachdem Filtern der Graphknoten wird über alle BPMN-MessageFlows iteriert. Ein Matching wird dann hinzugefügt, wenn die Quelle oder das Ziel des BPMN-MessageFlows bereits ein Matching mit dem BROS-Event besitzt. Dies ermöglicht das Erstellen 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, dass 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, eine konstante Kantenmenge entfernt wird, reduziert dies nur den vollständigen Graphen, um 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, wird der *Verifikationsalgorithmus* ausgeführt. Für die Implementierung der Regeln stellt die *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, dass 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, dass 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 haben, 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 ihr wir überprüft, dass 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-EndEvent, das zusätzlich ein Wahrheitswert TerminationEvent besitzt. Hierfür ist die Dreiwertigkeit des Rückgabewertes wichtig. Zunächst wird in der Vorbedingung nach einem BPMN-EndEvent 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. Sollte ein BROS-ReturnEvent gefunden werden, wird eine positive Konsistenzmeldung zurückgegeben, andernfalls eine Negative. Dank des Rückgabewertes *null*, kann die Vorbedingung einfach gehalten werden. Auch während der Regelausführung kann sie noch abgebrochen werden, wodurch die Überprüfung nicht in zwei komplexere Lambdas geteilt werden muss.

```
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, dass 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, dass das BPMN-Element ein BPMN-StartEvent ist. 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 Konsistenzmeldungen 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 brosCreate = bros.relations<BrosCreateRelation>()
                    .firstOrNull()
                    ?.source as? ModelTree<BrosElement>

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

Quelltext 14: Implementierung von Regel 5

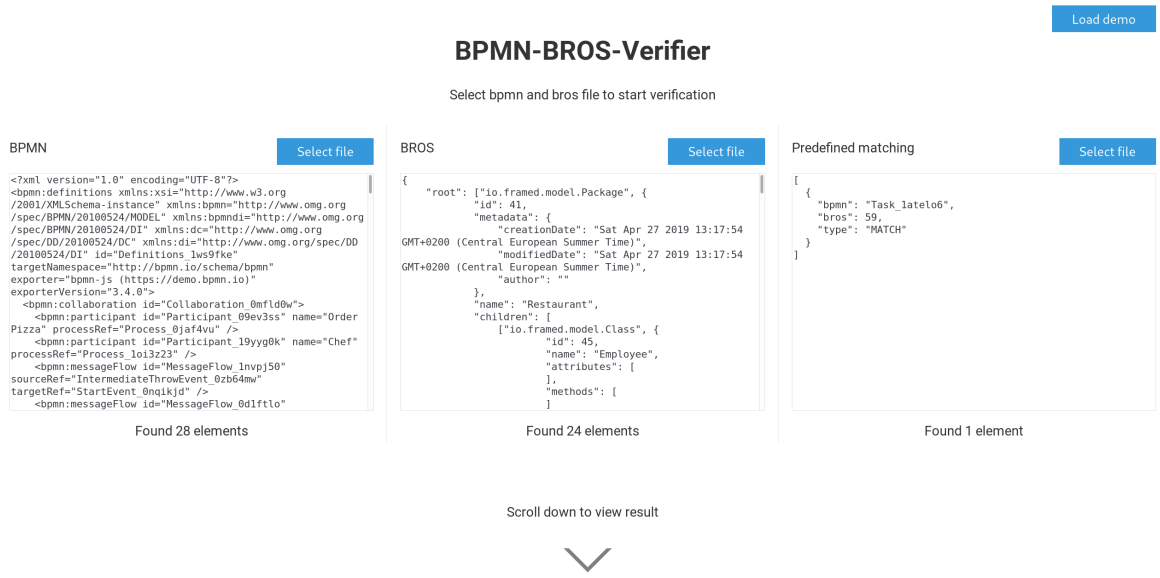


Abbildung 5.3.: Benutzerinterface des Dateiuploads

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 ersten 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 entsprechenden Dateien. Alternativ kann eine manuelle Dateiauswahl genutzt werden, oder der Inhalt der Dateien in die entsprechenden Textfelder kopiert werden. Unterhalb der Textfelder wird eine kurze Information zu dem geladenen Modell bzw. 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. Sobald jeweils ein valides BPMN- und BROS-Modell geladen wurden, wird die Konsistenzprüfung automatisch gestartet. Der Ergebnispfeil im unteren Bildschirmbereich signalisiert, dass ein Verifizierungsergebnis verfügbar ist.

Mit einem Klick auf den Ergebnispfeil 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 zur Durchführung und dem Ergebnis der Konsistenzprüfung. Dabei werden auch Informationen zum Matching angezeigt.

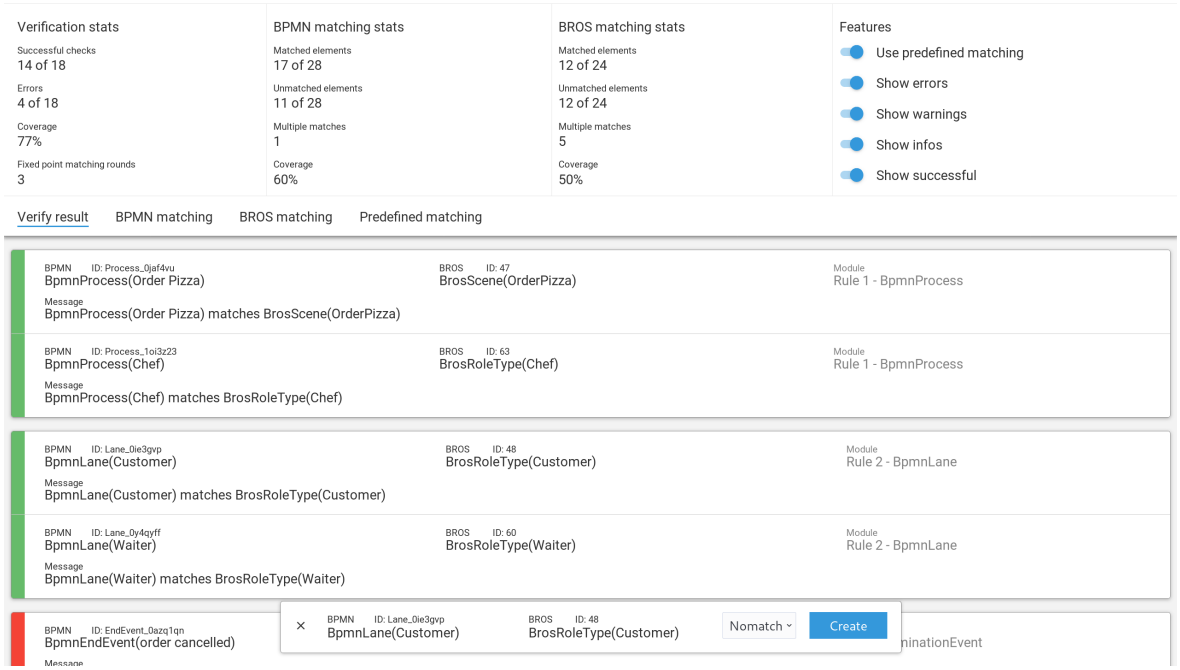


Abbildung 5.4.: Benutzerinterface für Validierungsergebnisse

- **Verification stats:** Informationen zu der aktuellen Verifikation.
  - *Successful checks (x of y):* Anzahl der x positiven Konsistenzmeldungen von allen y Konsistenzmeldungen.
  - *Errors (x of y):* Anzahl der x negativen Konsistenzmeldungen von allen y Konsistenzmeldungen.
  - *Coverage (x%):* Prozentualle 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.
    - *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%):* Prozentualle 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, dass 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 meist kein Fehler. Genauso ist das Vorhanden sein eines Matchings nicht immer korrekt, da das Matching auf die Namen angewiesen ist. Aus diesem Grund sind die Farbkodierungen 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 Regel, aus Sicht des Entwicklers, implementiert (Abschnitt 2).

6.1. Anwendung am Beispiel einer Pizzabestellung

In seiner Arbeit hat Schön et al. 2019 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 gereicht. Falls, während die Bestellung von dem Koch bearbeitet wird, 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 seine Bestellung zurücknehmen, was den Prozess vorzeitig beendet. Nachdem der Koch die Pizza fertig zubereitet hat, übergibt er sie wieder dem Kellner, der die Bestellung dem Kunden

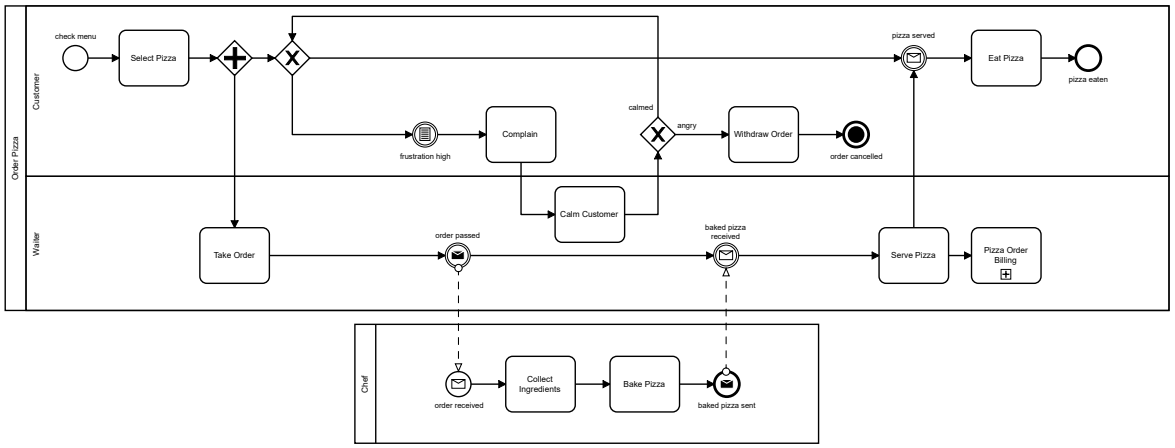


Abbildung 6.1.: BPMN-Modell der Pizzabestellung

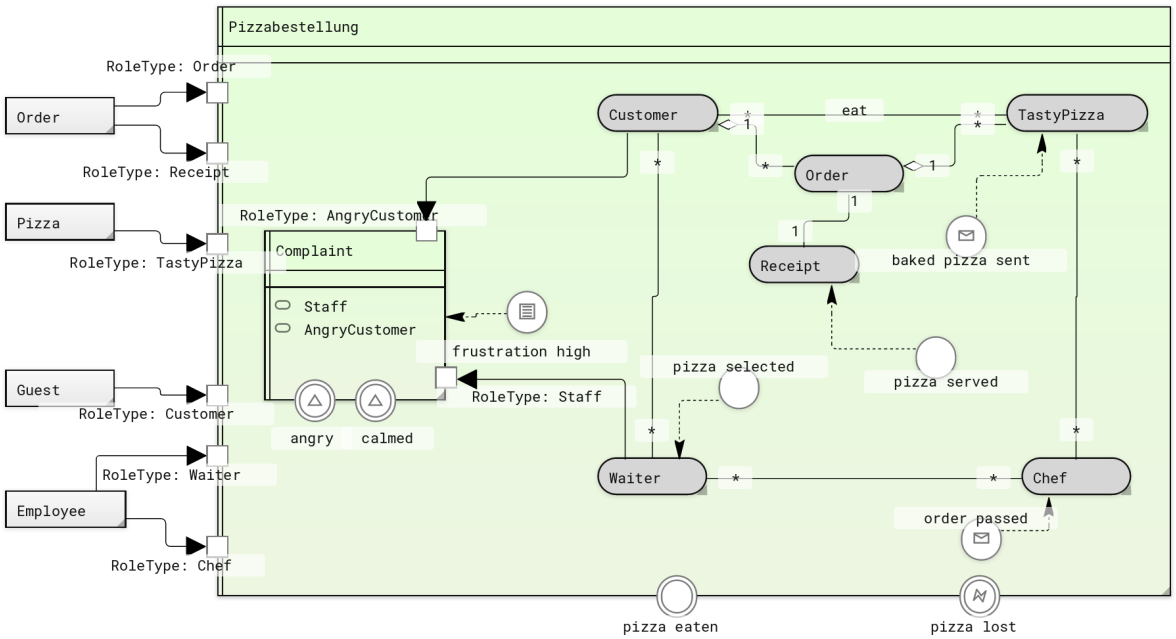


Abbildung 6.2.: Entwurf für das BROS-Modell der Pizzabestellung

überreicht. 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 NaturalType 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 einer Beschwerde von einem Kunden, wird mit einer eigenständigen Scene modelliert, die zwei verschiedene ReturnEvents besitzt. Diese stellen den Ausgang des Beschwichtigungsversuches durch den Kellner dar.

Nachdem beide Modelle erstellt wurden, kann die Konsistenzprüfung mit dem Tool 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 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.

Abbildung 6.3.: Fehlermeldungen 1

Der erste Regelverstoß (vgl. Abbildung 6.3) hat seinen Ursprung in der Regel 1 und besagt, dass für den Bpmn-Prozess "Order Pizza" kein entsprechendes BROS-Element gefunden werden konnte. Dies liegt an dem Namensunterschied gegenüber der BROS-Scene, die "Pizzabestellung" heißt. Ein derartiges Konsistenzproblem lässt sich auf zwei unterschiedlichen Wegen lösen. Zum einen kann 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 des Geschäftsprozesses zu erhalten, kann die BROS-Scene auch umbenannt werden.

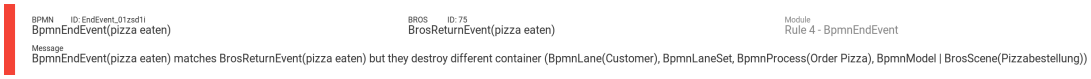


Abbildung 6.4.: Fehlermeldungen 2

Regel 4 ist die Ursache für das zweite Konsistenzproblem. Es besagt, dass das BPMN-EndEvent “pizza eaten” zwar ein Matching mit dem BROS-ReturnEvent “Pizza eaten” aufweist aber beide Elemente beenden unterschiedliche Teilprozesse (vgl. Abbildung 6.4). Dies liegt an der fehlenden Zuordnung des BPMN-Prozesses “Order Pizza” und der BROS-Scene “Pizzabestellung”. Da diese Zuordnung schon mit dem Lösen des ersten Regelverstoßes behoben wurde, ist keine weitere Änderung notwendig. Nachdem die BROS-Scene zu “OrderPizza” umbenannt (vgl. Abbildung A.1a) wurde, kann die Konsistenzprüfung mit dem Laden der neuen Datei wiederholt werden.

Abbildung 6.5.: Fehlermeldung 3

Das dritte Problem kommt von der Regel 3 und beschreibt, dass zu dem BPMN-TerminationEvent “order cancelled” kein passendes BROS-ReturnEvent gefunden wurde (vgl. Abbildung 6.5). Dieses Problem stammt von dem vergessenen BROS-ReturnEvent. Es kann leicht behoben werden, indem zu der BROS-Scene 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.

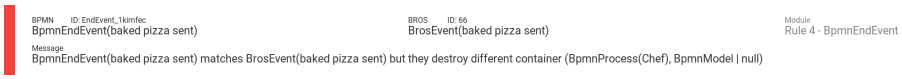


Abbildung 6.6.: Fehlermeldung 4

Für das vierte Konsistenzproblem ist Regel 4 verantwortlich. Es besagt, dass für das BPMN-EndEvent “baked pizza send” zwar ein BROS-Event im Matching gefunden werden konnte, aber die Elemente, die von den Events beendet werden, unterschiedlich sind (vgl. Abbildung 6.6). 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 vebunden, wie in Abbildung A.1c zu sehen ist.

Abbildung 6.7.: Fehlermeldung 5

Der vorletzte Regelverstoß basiert auf der Regel 5 und ist trivial zu lösen. Er beschreibt, dass zu dem BPMN-StartEvent “check menu” kein passendes BROS-Event mit einer BROS-CreateRelation zu der BROS-Scene “OrderPizza” existiert (vgl. Abbildung 6.7). Dieses BROS-

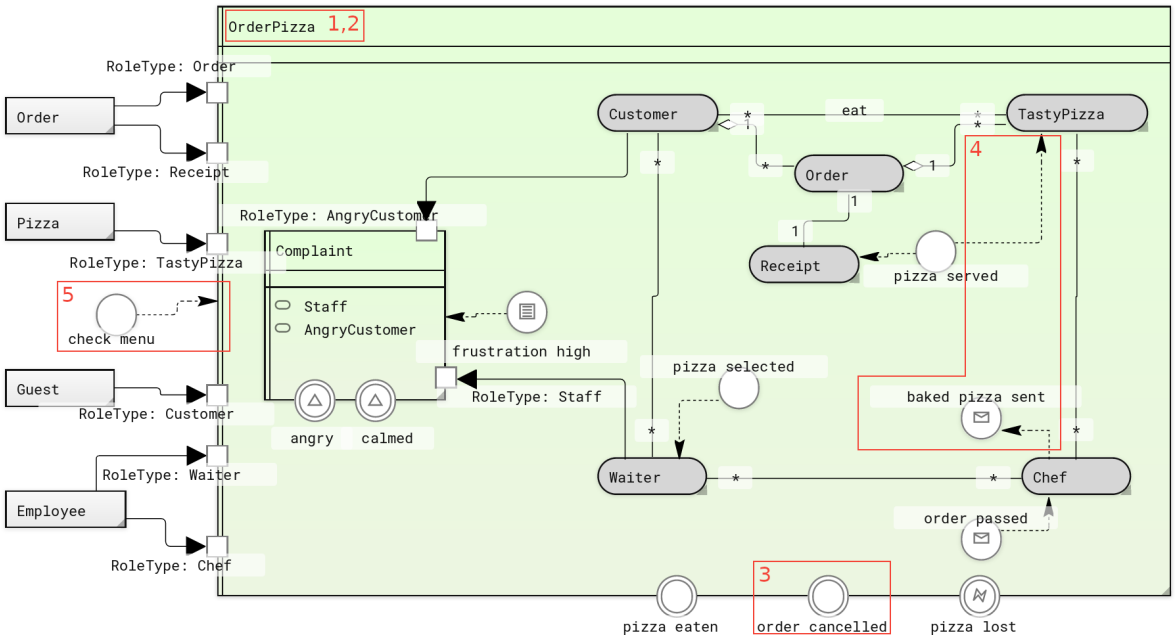


Abbildung 6.9.: Vollständiges BROS-Modell der Pizzabestellung

Event wurde im bisherigen BROS-Model vergessen. Zum Lösen des Konsistenzproblems, muss ein BROS-Event mit dem Namen “check menu” und der dazugehörigen BROS-CreateRelation hinzugefügt werden (vgl. Abbildung 6.9).

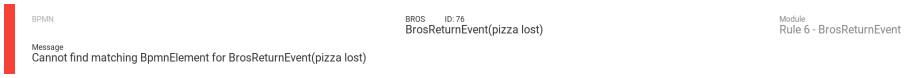


Abbildung 6.8.: Fehlermeldung 6

Anschließend bleibt nur noch ein Konsistenzproblem übrig. Dieses besagt, dass ein Verstoß gegen Regel 6 vorliegt, indem zu dem BROS-Event “pizza lost” kein passendes BPMN-Element gefunden wurde (vgl. Abbildung 6.8). 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 Funktionalität 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 Abbildung 6.9 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. Die zu Abbildung 6.2 geänderten Modellelemente sind in Abbildung 6.9 rot markiert und mit der Fehlernummer annotiert. Mit der letzten negativen Konsistenzmeldungen des Tools wurde gezeigt, dass ein Regelverstoß absichtlich herbeigeführt werden kann und dies nicht unbedingt einen Fehler darstellt. 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 implementiert, 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, dass 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 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 ihre inneren 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, BrosEvent> { 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 die äquivalente Funktion *verifyBros* genutzt. Im generischen Parameter der Funktion wird nach einem BROS-Event gefiltert. Innerhalb des Lambdas wird überprüft, ob es ein BPMN-Element im Matching des BROS-Events gibt. 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<BrosEvent> { bros ->
    for (bpmn in bros.matchingElements) {
        if (bpmn.checkType<BpmnElement>()) {
            return@verifyBros Result.match("...", bpmn = bpmn)
        }
    }
    Result.error("...")
}
```

Quelltext 16: Implementierung von Regel 6

Ein Nachteil der *Kotlin/JS* Plattform ist das Fehlen von *Reflections* und *Annotation-Processing* 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 im Tool zu registrieren, wird die Liste um einen neuer Eintrag (*Context::setupRule6BrosEvent*) ergänzt. 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. Ergebnisse und Ausblick

In diesem Kapitel wird evaluiert in wie weit diese Arbeit, die im Kapitel 1 genannten Ziele erfüllt hat. Zu diesem Zweck werden die Ergebnisse der Analyse und der Implementierung zusammengefasst und in das Feld der bestehenden Verfahren eingeordnet. Dabei werden die Ergebnisse der einzelnen Teilschritte analysiert und benutzt, um das Klassifikationsschema aus Kapitel 3 zu erweitern. Abschließend wird ein Ausblick über mögliche Erweiterungen des vorgestellten Verfahrens und weiterführende Arbeiten auf dem Gebiet des Konsistenzvergleiches zwischen BPMN- und BROS-Modellen gegeben.

7.1. Zusammenfassung

Ziel der Arbeit war, die Konsistenz zwischen BPMN- und BROS-Modellen zu untersuchen. Dieses wurde erreicht, indem eine Lösung des Konsistenzproblems für die beiden Modellarten erstellt wurde. Als Einführung in BPMN und BROS wurden deren Konzepte und Metamodelle erläutert. Um einen Einstieg in das Themengebiet der Konsistenzprüfung zu erhalten, wurden bestehende Verfahren zur Überprüfung der Konsistenz von UML-Modellen analysiert. Dafür wurde ein Klassifikationsschema auf Basis des Schemas von Usman et al. 2008) erstellt und zur Klassifizierung der Verfahren verwendet.

Mit Hilfe der erstellten Metamodelle und den Konsistenzverfahren für UML-Modelle wurden Konsistenzbedingungen zwischen BPMN- und BROS-Modellen erarbeitet. Die sechs erstellten Regeln basieren auf dem direkten Vergleich der Modelle, anhand ihrer Metamodelle. Diese in Prolog formalisierten Regeln benötigen keine Zwischendarstellung und sind daher leicht erweiterbar. Für die Automatisierung der Konsistenzprüfung wurde eine Referenzarchitektur erstellt, die die formalen Regeln ausführt. Dieses zweistufige Verfahren baut zunächst eine Zuordnung von Elementen auf und überprüft anschließend die Modelle mit dem Regelsatz auf ihre Konsistenz. Im nächsten Schritt wurde das Verfahren implementiert, um einerseits seine Funktionalität zu zeigen und andererseits die Automatisierbarkeit zu beweisen. Das dabei erstellte Tool ist in Kotlin entwickelt wurden, um zusätzlich eine mögliche Anbindung an den BROS-Editor *FRaMED.io* zu erleichtern. Wie die Modelleditoren *bpmn.io* und *FRaMED.io*, ist das Tool webbasiert um plattformübergreifend nutzbar zu sein.

Die Evaluation des Verfahrens teilt sich in zwei Abschnitte auf. Zum einen wurde die Anwendbarkeit anhand von einem Beispielsystem getestet. Dazu wurde das Beispiel einer Pizzabestellung aus der Arbeit von Schön et al. 2019 verwendet. In seiner Arbeit hat Schön et al. 2019 sowohl ein BPMN- als auch ein BROS-Modell gegeben, die zueinander konsistent sind. Zum anderen wurde die Erweiterbarkeit und Modularität des Verfahrens und der Implementierung evaluiert, indem eine weitere Regel aus dem Kapitel 4 zu dem Tool hinzugefügt wurde.

	Diagrams	Consistency Type	Consistency Strategy	Intermediate Representation	Case Study	Automatable	Tool Support	Model Extensibility	Rule Extensibility
Rasch 2003	CD, SM	Intra	Monitoring	CSP/OZ	✓	●	✗	●	◐
Shinkawa 2006	UCD, CD, SD, AD, SC	Inter	Analysis	CPN	✗	●	✗	◐	○
Mens 2005	CD, SD, SC	All	Monitoring	Extended UML	✓	●	✓	●	◐
Egyed 2001	CD, OD, SD	Intra, Inter	Construction		✗	●	∼	◐	◐
Egyed 2006	CD, SD, SC	Intra	Monitoring		✓	●	✓	○	◐
BROS	BPMN, BROS	Intra	Monitoring		✓	●	✓	○	●

●=mit geringem Aufwand; ◐=mit mittlerem Aufwand; ○=mit hohem Aufwand;
✓=ja; ✗=nein; ∼=teilweise

Tabelle 7.1.: Einordnung des neuen Verfahrens

7.2. Wissenschaftlicher Beitrag

Das in dieser Arbeit vorgestellte Verfahren zur Lösung des Konsistenzproblems basiert auf der Überprüfung eines Regelsatzes. Diese Regeln nutzen die Graphstruktur die durch die Metamodelle aufgestellt werden, um den Konsistenzvergleich durchzuführen. Damit gehört das Verfahren, nach dem Klassifikationsschema aus Kapitel 3, zu der Klasse der Verfahren ohne Zwischendarstellung. Die Nutzung eines Regelsatzes entspricht der Konsistenzstrategie des Monitorings. In Tabelle 7.1 ist die Einordnung diese Arbeit in das bestehende Feld von Konsistenzverfahren abgebildet. Die größte Übereinstimmung hat das hier vorgestellte Verfahren mit dem Konsistenzverfahren 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. Dies erschwert die Erweiterbarkeit um neue Modellarten ungemein. Nur Modellarten, die zu dem UML-Standard gehören, werden von der OCL und damit von dem Verfahren unterstützt. Auch das hier genutzte Verfahren hat einen hohen Aufwand, wenn eine neue Modellart hinzugefügt wird. Allerdings ist die Erweiterbarkeit um neue Regeln mit weniger Aufwand verbunden. Da das Verfahren direkt auf dem Graphen der Metamodell-Instanz arbeitet, können beliebige Regeln leicht erstellt werden. Das Verfahren von Egyed 2006 ist hingegen auf die Möglichkeiten der OCL beschränkt, was die Erstellung einer neuen Regel erschweren könnte.

Diese Erweiterbarkeit wurde anhand einer Implementierung nachgewiesen. Ziel der Implementierung war, den Konsistenzvergleich zu automatisieren und gleichzeitig die Modularität und damit auch Erweiterbarkeit des Verfahrens zu gewährleisten. Dies wurde mit der Aufteilung der Konsistenzbedingungen in einzelne Regeln erreicht. Jede dieser Regeln besteht aus einem,

teilweise auch mehreren Modulen, die sich unabhängig voneinander aktivieren und ergänzen lassen. Die Automatisierbarkeit wurde mit dieser Implementierung gezeigt und anhand eines Beispieles zusätzlich evaluiert. Dabei konnte ein fehlerhaftes BROS-Modell erfolgreich korrigiert werden. Da das genutzte Beispiel auf der Arbeit von Schön et al. 2019 basiert, konnte diese zur Verifizierung der Korrektheit der Konsistenz genutzt werden.

7.3. Zukünftige Arbeiten

Wie in Kapitel 6 beschrieben muss ein von dem Tool gefundener Konsistenzfehler, nicht unbedingt ein echter Konsistenzfehler sein. Dieser Fehler kann auch von dem Modellierer beabsichtigt sein und eine gewollte Erweiterung des Modelles darstellen. Momentan kann das Tool nicht mit diesem Problem umgehen. Eine Möglichkeit bestimmt Fehler zu ignorieren oder als gewollt zu kennzeichnen, wäre eine sinnvolle Erweiterung für das implementierte Tool.

Um die allgemeine Funktionalität zu verbessern, ist es erforderlich die Konsistenzbeziehungen zwischen BPMN- und BROS-Modellen genauer zu untersuchen. In dieser Arbeit wurden nur sechs Konsistenzregeln vorgestellt, die die wichtigsten Bereiche der Konsistenz abdecken. So sind noch keine Konsistenzbedingungen für zum Beispiel BROS-Compartments oder BPMN-SubProcesses definiert. Die Konsistenzregeln, die zwischen diesen und weiteren Elementen gelten, müssen dafür noch analysiert werden. Solange eine Formalisierung dieser Regeln auf Basis der Metamodelle möglich ist, können sie der bestehenden Implementierung hinzugefügt werden.

Neben der Erweiterung um neue Regeln, kann auch die Erweiterung um die syntaktische und semantische Konsistenzprüfung der Einzelmodelle untersucht werden. Momentan ist diese Aufgabe abhängig von dem Modellierer. Zwar existieren schon Verfahren und automatisierte Tools für diesen Konsistenzvergleich, allerdings sind sie nicht kompatibel mit der aktuellen Implementierung. Auch müsste untersucht werden, ob dieser Konsistenzvergleich in einem zufriedenstellenden Funktionsumfang integrierbar ist. Externe Programme, die diese Funktionalität bereits besitzen, sind zumeist Editoren für diese Modelle. Diese sind darauf spezialisiert, diese Art von Konsistenzprüfung durchzuführen. Eine eigene Implementierung hat die Gefahr einer schlechteren Erkennungsrate.

Eine weitere mögliche Erweiterung des Tools wäre die Integration in den bestehenden BROS-Editor *FRaMED.io*. Da BROS-Modelle auf den BPMN-Modellen aufbauen und teilweise wiederverwenden werden können, kann eine Integration den Entwicklungsprozess eines BROS-Modelles zu einem BPMN-Modell erheblich vereinfachen. So könnte das BPMN-Modell direkt in den Editor geladen werden. Dies würde dem Modellierer verschiedene Möglichkeiten geben. Zum einen könnten Fehlermeldungen direkt graphisch in dem BROS-Modell markiert werden. Damit würden sich Fehler leichter finden und verstehen lassen. Zum anderen könnten Teile des BROS-Modelles anhand des BPMN-Modelles autogeneriert werden. Dies würde dem Modellierer vor Allem bei großen Prozessen viel Zeit sparen. Da die Referenzimplementierung, wie auch der BROS-Editor, in Kotlin entwickelt worden, sind sie bereits technisch kompatibel. Allerdings müsste noch evaluiert werden, wie das Ergebnis der Analyse in die graphische Darstellung integriert werden kann. Sollte eine Autogenerierung implementiert werden, müsste zusätzlich noch analysiert werden, in wieweit dies möglich ist, und wie nutzbar ein solches Modell gegenüber einem manuell erstelltem Modell wäre.

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. (2019). „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.

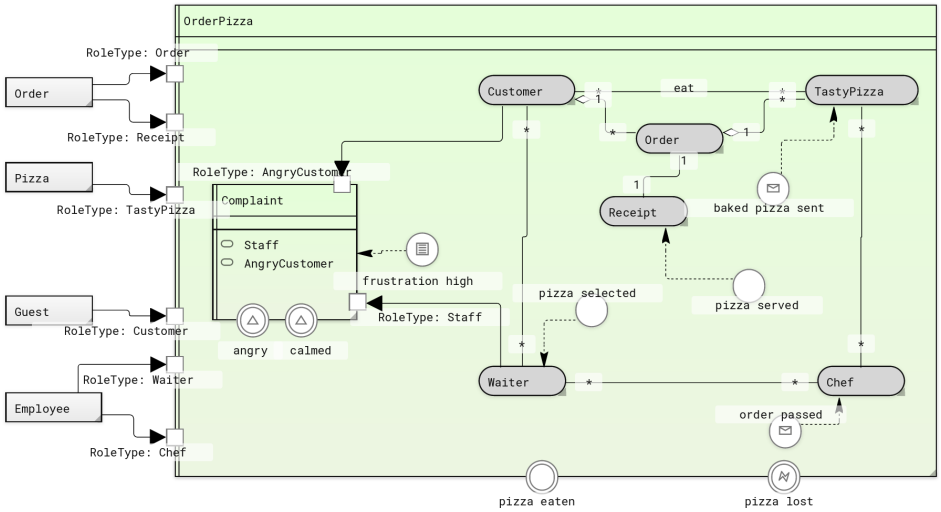
Simmonds, Jocelyn et al. (2004). „Maintaining Consistency between UML Models Using Description Logic". In: *L'OBJET* 10.2-3, S. 231–244. ISSN: 1262-1137. DOI: 10.3166/objet.10.2-3.231-244.

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.

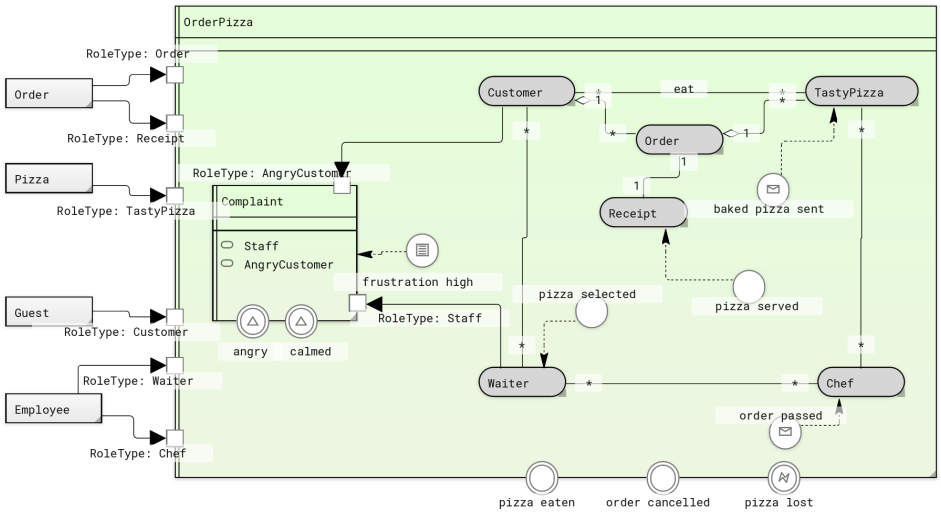
A. Appendix

```
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
```

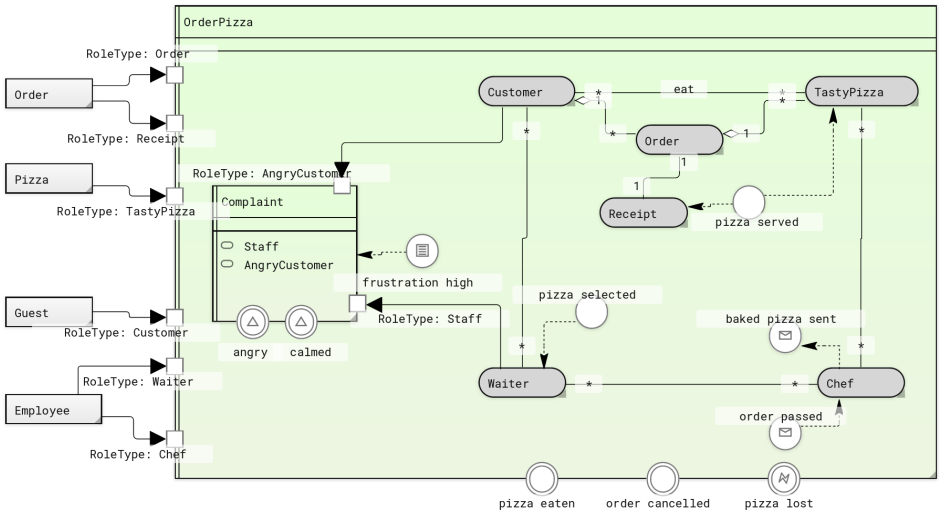
Quelltext 17: Algorithmus zum Name-Matching



(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

```
1  fun Context.setupRule6() {
2      match<BpmnGateway, BrosEvent> { bpmn, bros ->
3          bpmn.relations<BpmnSequenceFlow>().any { flow ->
4              flow.relation.name.isNotBlank() &&
5                      matchStrings(
6                          flow.relation.name,
7                          bros.element.desc
8                      )
9          }
10     }
11     match<BpmnGateway, BrosReturnEvent> { bpmn, bros ->
12         bpmn.relations<BpmnSequenceFlow>().any { flow ->
13             flow.relation.name.isNotBlank() &&
14                     matchStrings(
15                         flow.relation.name,
16                         bros.element.desc
17                     )
18         }
19     }
20
21     match<BpmnTask, BrosEvent> { bpmn, bros ->
22         matchStrings(bpmn.element.name, bros.element.desc)
23     }
24     match<BpmnTask, BrosReturnEvent> { bpmn, bros ->
25         matchStrings(bpmn.element.name, bros.element.desc)
26     }
27
28     verifyBros<BrosEvent> { bros ->
29         for (bpmn in bros.matchingElements) {
30             if (bpmn.checkType<BpmnElement>()) {
31                 return@verifyBros Result.match("...", bpmn = bpmn)
32             }
33         }
34         Result.error(".")
35     }
36     verifyBros<BrosReturnEvent> { bros ->
37         for (bpmn in bros.matchingElements) {
38             if (bpmn.checkType<BpmnElement>()) {
39                 return@verifyBros Result.match("...", bpmn = bpmn)
40             }
41         }
42         Result.error("...")
43     }
44 }
```

Quelltext 18: Implementierung der Datei Rule6BrosEvent.kt

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur angefertigt habe.

Ort, Datum

Unterschrift