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

Eingereicht am: 29.08.2019

Inhaltsverzeichnis

Abstract

Gekürzte Version der Einleitung

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

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

1. Einleitung

1.1. Motivation

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

1.2. Problemdefinition

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

1.3. Struktur der Arbeit

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

2. Hintergrund

2.1. Business Process Model and Notation

Die *Business Process Model and Notation* (**BPMN**) ist ein Standard für die grafische Beschreibung von Geschäftsprozessen. Dabei wird das Verhalten eines Systems mit einer an Flussdiagrammen angelehnten Form beschrieben. Die Hauptelemente von BPMN sind:

- **Activity:** Eine Activity beschreibt eine Tätigkeit die innerhalb des Geschäftsprozesses ausgeführt wird. Da eine Aufgabe ausgeführt wird, kann die Ausführung einer Activity längere Zeit in anspruch nehmen. Zur Darstellung wird ein Rechteck mit abgerundeten Ecken verwendet.

- **Gateway:** Ein Gateway wird für die Steuerung des Kontrollflusses verwendet. An einem Gateway können verschiedene Kontrollwege zusammenlaufen oder sich teilen. Dabei werden verschiedene Arten, wie zB. AND- und OR-Gateways unterstützt. Dargestellt wird ein Gateway mittels einem um 45 Grad gedrehtem Quadrat.

- **Event:** Ein Event beschreibt Ereignis das innerhalb des Geschäftsprozesses auftreten kann und werden mit einem Kreis dargestellt. Sie beeinflussen den Kontrollfluss, können ihn starten, pausieren oder auch beenden. Events werden in drei verschiedenen Dimensionen eingeteilt, nach ihrer Position, nach ihrer Wirkung und nach ihrer Art. Für die weitere Arbeit ist nur die Unterteilung nach ihrer Position innerhalb des Geschäftsprozesses wichtig. Dabei wird zwischen einem StartEvent (einfacher Rahmen), einem IntermediateEvent (doppelter Rahmen) und einem EndEvent (dicker Rahmen) unterschieden.

- **Flow:** Ein Flow ist eine gerichtete Verbindung zwischen anderen Modellelementen. Dabei wird zwischen SequenceFlows und MessageFlows unterschieden. Ein SequenceFlow stellt den Kontrollfluss dar und verbindet Elemente um eine Ausführungsreihenfolge festzulegen. Ein MessageFlow verbindet unterschiedliche Teilnehmer des Geschäftsprozesses und stellt die austausch von Mitteilungen dar.

- **Pool:** Ein Pool stellt eine Gruppe von zusammengehörenden Teilnehmern innerhalb eines Geschäftsprozesses dar. Dargestellt wird ein Pool mit einem Rechteck wobei der Name am linken Rand steht.

- **Swimlane:** Eine Swimlane ist ein einzelner Teilnehmer der zu einem Pool gehört und Aufgaben aus dem Geschäftsprozess erfüllt. Bei einem Pool der nur aus einer Swimlane besteht kann die Swimlane mit dem Pool vereinigt werden. Innerhalb eines Pools wird die Simelane als eine horizontale Bahn dargestellt.

- **Process:** Der Process bildet den vollständigen Geschäftsprozess ab und ist das Containerelement für die anderen Modellelemente.

bpmn.io

Aufgrund der größe des BPMN Standards wird nur ein gekürztes Metamodell auf Basis von Loja et al. 2010 betrachtet.

2.2. Compartment Role Object Model

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

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

CROM besitzt mit dem Eclipse Plugin *FRaMED-2.0* Editor Support.

2.3. Business Role-Object Specification

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

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

Mit Ausnahme von RoleConstraints wird das vollständige Metamodell aus Schön et al. o.D. betrachtet.

3. Verwandte Arbeiten

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

3.1. Klassifikationsschema für Verfahren zur Konsistenzprüfung

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

- **Nature (Usman et al. 2008):** Es beschreibt den Fokus der vergleichbaren Modelltypen. Dabei wird zwischen strukturbasierten und verhaltensbasierten Modellen unterschieden. Strukturbasierte Modelle beschreiben den Aufbau eines Systems. Dazu zählen unter anderem UML-Klassen-, UML-Komponentendiagramm und BROS-Modelle. Zu den verhaltensbasierten Modellen gehören beispielsweise UML-Zustands-, UML-Aktivitätsdiagramme und BPMN-Modelle. Diese Modelle verdeutlichen den die Abläufe und Zustände innerhalb eines Systems. Mögliche Werte sind strukturbasiert, verhaltensbasiert und beides.

- **Diagrams (Usman et al. 2008):** Es beschreibt welche konkreten Modelle von der Methode unterstützt werden. Die Arbeiten von Usman et al. 2008 und Lucas et al. 2009 beziehen sich dabei ausschließlich auf UML Diagramme. Mögliche Werte sind die Modellnamen.

- **Consistency Type (Usman et al. 2008):** Es beschreibt welche Arten der Konsistenz von der Methode überprüft werden. Dabei wird hauptsächlich unterschieden zwischen *Inter-Modell (vertikale) Konsistenz* (Konsistenz bei verschiedenen Abstraktionsstufen und gleichem Modelltyp), *Intra-Modell (horizontale) Konsistenz* (Konsistenz bei gleicher Abstraktionsstufe und verschienden Modelltypen)und *Evolutionskonsistenz* (Konsistenz der eines Modelles über verschiedende Entwicklungsstufen). Zusätzlich spezifiziert Usman et al. 2008 noch die *semantische-* und die *syntaktische Konsistenz*. Diese Beziehen sich auf die Konsistenz eines

Modelles zu seinem Metamodell. Dies wird für die nachfolgende Arbeit als Vorraussetzung angesehen und nicht näher betrachtet.

- **Consistency Strategy (Usman et al. 2008):** Es beschreibt die benutzte Validierungsstrategie. Es werden drei verschiedene Strategien genannt und zwar *Analysis* (Auf einem Algorithmus basierend), *Monitoring* (Auf einem Regelsatz basierend) und *Construction* (Auf der Generierung des zu vergleichenden Modelles basierend).
- **Intermediate Representation (Usman et al. 2008):** Es beschreibt ob die Methode eine temporäre Zwischendarstellung benötigt oder nicht. Mögliche Werte sind ja und nein.
- **Case Study (Usman et al. 2008):** Es beschreibt ob die Methode an einem Beispiel evaluiert wurde. Mögliche Werte sind ja und nein.
- **Automatable (Usman et al. 2008):** Es beschreibt ob die Methode manuell oder automatisiert von einem Programm durchgeführt werden kann. Mögliche Werte sind gut (H), mittel (M) und schlecht (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 gut (H), mittel (M) und schlecht (L).
- **Rule Extensibility:** Es beschreibt wie gut die Methode um weitere Konsistenzregeln erweiterbar ist. Mögliche Werte sind gut (H), mittel (M) und schlecht (L).

Dieses Schema ist direkt auf die Konsistenzprüfung von BPMN- und BROS-Modelle anwendbar. Wichtig für die weitere Arbeit sind Methoden deren *Nature* beide Modelltypen unterstützt und deren *Consistency Type* auf der *Intra-Modell (horizontale) Konsistenz* beruht. Den Aspekt der Erweiterbarkeit wird von Lucas et al. 2009 genannt. Da ein Hauptziel dieser Arbeit die Erweiterbarkeit ist wird dieser Aspekt nocheinmal untergliederd.

3.2. Aktuelle Verfahren zur Konsistenzprüfung

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. Anhand eines Regelsatzes wird die Konsistenz der Zwischendarstellungen geprüft. Kim et al. 2004 nutzt ein ähnliches Verfahren, spezialisiert sich dabei aber nur auf Zustandsdiagramme.

Transformation zu *Pertinetze*: Shinkawa 2006 zeigt das verhaltensbasierte UML-Modelle in *CPN* (Coloured Petri Net) überführt werden können. Mittels mehrerer Beispiele werden verschiedene Transformationsstrategien 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. *TODO: Vorteil von GSPN?*

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 eien Fallstudie durch und entwickelt eine Toolunterstützung. Ein ähnlicher Ansatz wird von Simmonds et al. 2004 genutzt. Noch weiter geht Satoh et al. (Usman et al. 2008, 15) indem UML-Klassendiagramme direkt in Logikprogramme übersetzt und ausgeführt werden.

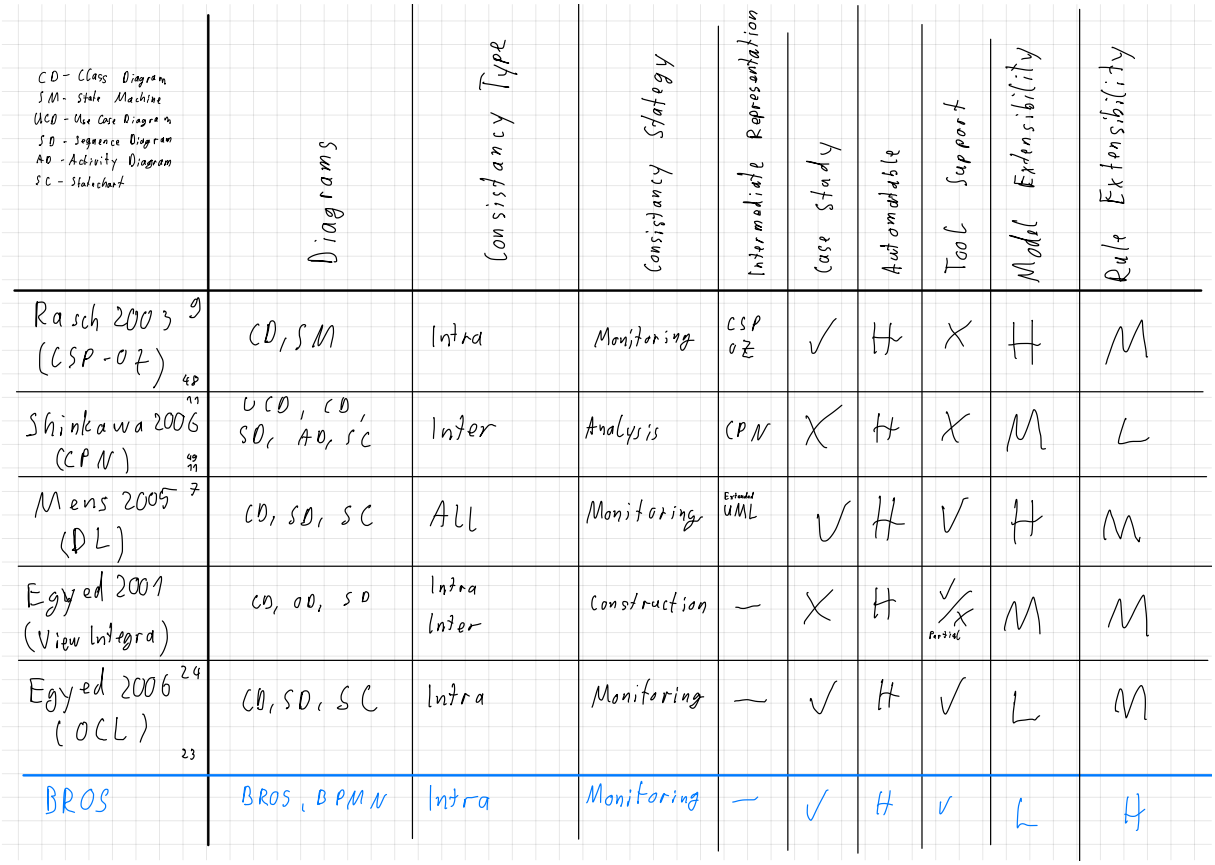

Abbildung 1.: Vergleich der bestehenden Verfahren

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

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

3.3. Vergleich der bestehenden Verfahren

Die fünf beschriebenen Verfahren lassen sich in zwei Kategorien teilen. Zum einen die die eine Zwischendarstellung benötigen und die die keine Zwischendarstellung nutzen. Rasch et al. 2003, Shinkawa 2006 und Mens et al. 2005 beschreiben Verfahren die zunächst eine Zwischendarstellung aufbauen und anschließend die Konsistenz der Zwischendarstellung überprüfen. Andererseits könne die Verfahren von Egyed 2001 und Egyed 2006 direkt auf den Modellen arbeiten. Der große

Vorteil ist die leichte Erweiterbarkeit um neue Modellarten. Es muss nur eine neue Konvertierungsmethode des Modelles zur Zwischendarstellung entwickelt werden. Anschließend können die gleichen Algorithmen zur Konsistenzprüfung wie bei den bestehenden Modellen verwendet werden. Allerdings ist dies auch eine Beschränkung der Erweiterbarkeit. Eine allgemeine Form der Zwischendarstellung hat zumeist den Nachteil des Informationsverlustes. Dadurch lassen sich nur schwer neue Regeln für die Konsistenz Zwischendarstellung hinzufügen. Es kann auch sein das eine Modellart nicht kompatibel zu der Zwischendarstellung ist. Dann ist eine Erweiterung nicht möglich. Dagegen haben die Verfahren ohne Zwischendarstellung weniger Beschränkungen der Erweiterbarkeit. 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. Egyed 2001 umgeht dieses Problem mittels einer iterativen Transformation.

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. Verfahren die mit einem Transformationsalgorithmus eine Zwischendarstellung konstruieren, zählen nicht dazu. Egyed 2001 nutzt diese Methode und beschreibt die Transformation zwischen verschiedenen UML-Modellen. Die Konsistenzprüfung der Modelle erfolgt mittels *Inter-Modell (vertikale) Konsistenz* und ist unabhängig von dem Konstruktionsverfahren. Da die eigentliche Konsistenzprüfung nicht teil eines Konstruktionsverfahrens ist, kann diese Methode orthogonal zu den anderen beiden Methoden verwendet werden.

4. Konsistenz zwischen BPMN und BROS

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

4.1. Konsistenzproblem

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

Eine formale und allgemeingültige Definition des Konsistenzproblem in der Domain der Modellierung ist in der Literatur nur schwer zu finden. Bevor das Konsistenzproblem definiert werden kann muss zunächst 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 Sprach oder mit einem formalen Ansatz beschieben sein, 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 über das aufstellen und prüfen von Konsistenzregeln.*

Diese Definition lässt sich leicht auf Konsistenzverfahren anwenden, die auf der Überprüfung von Regeln basieren. Aber auch formale Verfahren ohne einen eigenen Regelsatz lassen sich mit dieser Definition beschreiben. So können zB. die Verfahren, die auf der Transformation zu

einem Petrinetz basieren mittels der Konsistenzregel "das konstruierte Petrinetz muss lebendig sein"überprüft werden.

4.2. Konsistenzregeln

Die horizontale Konsistenzprüfung zwischen zwei Modellen ohne eine Zwischendarstellung kann 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 Regel aus der BPMN Sicht aufgestellt.

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

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

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

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

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

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

Quelltext 1: Definitionen der Faktenbasis

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

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

```
transitive_parent(Child, Parent) :-
    parent(Child, Parent).
transitive_parent(Child, Parent) :-
    transitive_parent(Child, X), parent(X, Parent).

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

Quelltext 2: Definitionen der weiterführenden Regeln

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

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

Quelltext 3: Regel 1

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

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

Quelltext 4: Regel 2

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

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

Quelltext 5: Regel 3

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

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

Quelltext 6: Regel 4

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

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

Quelltext 7: Regel 5

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

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

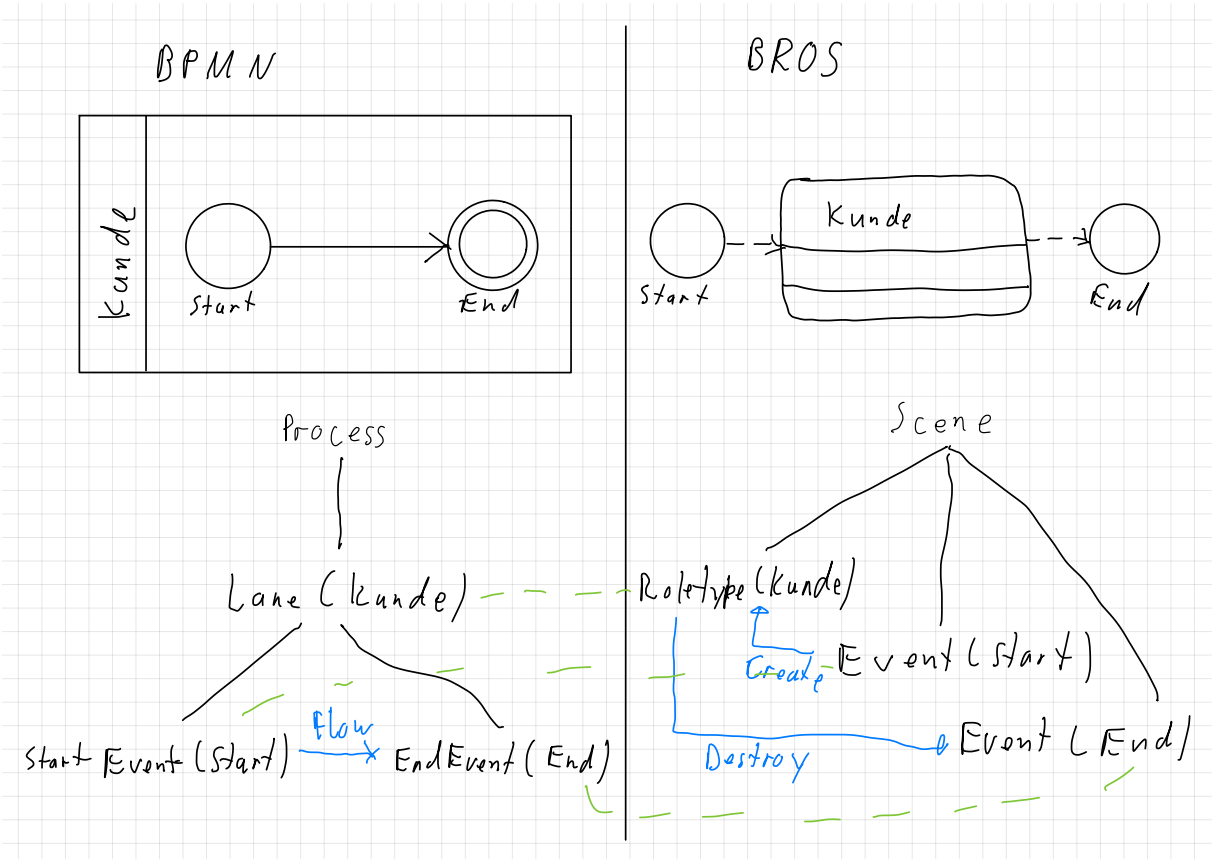

Abbildung 2.: Visualisierung des Graph Matching

).

Quelltext 8: Regel 6

4.3. Referenzarchitektur

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

Um das Verfahren anzuwenden müssen beiden Modelle in Form eines gerichteten geschichteten Graph vorliegen. Ein geschichteter Graph basiert auf einem Baum der die Grundstruktur des Modelles und die Eltern-Kind Beziehung abdeckt. Um die Relationen zwischen den Modellelementen darzustellen wird ein Schicht mit Querverweisen über den Baum gelegt. In Abbildung 2 wird die Transformation der Modelle in die dazugehörigen Graphen dargestellt. Auf der linken Seite ist ein BPMN-Modell mit dem dazugehörigen Graphen. Auf der rechten Seite ist ein vereinfachtes BROS-Modell mit seinem Graphen. Der schwarze Teil der Graphen bildet die Grundstruktur oder auch die erste Schicht. Dieser wird mit Hilfe der "containsRelation der Metamodelle direkt aufgebaut. In blau ist die zweite Schicht, die Schicht der Relationen, visualisiert. Jede Relation

ist mit ihrem Typ annotiert und hat eine feste Richtung. Die Graphen des BPMN-Modelles und des BROS-Modelles sind zueinander disjunkt.

Mit beiden Graphen kann nun der Matching-Algorithmus durchgeführt werden. Das Matching ist eine weitere Schicht innerhalb der Graphen, die den Graphen des BPMN-Modelles mit dem Graphen des BROS-Modelles verbindet. Um diese Schicht zu konstruieren wird eine Orakelfunktion genutzt. Diese ermittelt ob zwei Elemente zueinander gehören. Dabei 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 per Relation verbunden Elemente. Die dritte Schicht ist zu Beginn leer. Wenn die Orakelfunktion für das BPMN-Element x und das BROS-Element y ein Matching feststellt, werden die Kanten *(x, y)* und *(y, x)* der dritten Schicht hinzugefügt. Eine einmal hinzugefügt Kante kann nicht wieder entfernt werden. Da die dritte Schicht auch mit an das Orakel übergeben wird, kann dies aufgrund von einem bereits bestehen Matching ein eine Entscheidung treffen. Um dieses Verhalten zu nutzen wird der Vorgang des Matching aufbaues solange wiederholt wie sich die dritte Schicht nicht weiter ändert. Dieses Verhalten wird Fixpunkt-Algorithmus genannt. Innerhalb einer Iteration wird die Orakelfunktion auf alle paare von Elementen des BPMN- und des BROS-Modelles angewendet. Ein Fixpunkt-Algorithmus kann unter umständen nicht terminieren, wenn die Datenstruktur in eine oszillierenden Zustand übergeht. Dies wird verhindert indem Kanten nur zu dem Matching hinzugefügen werden und somit nie entfernt werden dürfen. Damit kann die dritte Schicht maximal zu einem Vollständigen Graphen anwachsen. Anschließend kann keine weitere Kante hinzugefügt werden und der Algorithmus terminiert automatisch.

Mit dem konstruieren Matching kann nun der Verifikations-Algorithmus ausgeführt werden. In diesem Schritt werden die unter Abschnitt 4.2 aufgestellten Konsistenzregeln überprüft. Diese Überprüfung kann mittels der genannten Prolog Regeln oder auch direkt auf den Graphen durchgeführt werden. Anders als der Matching-Algorithmus wird der Verifikations-Algorithmus nur einmal ausgeführt. Da alle Regeln die Konsistenzprüfung von einem Modell auf das andere Modell durchführen, können die Regeln unabhängig voneinander auf den Modellen ausgeführt werden. Die Regeln die die Überprüfung aus Sicht des BPMN-Modelles ausführen werden für jedes Element des BPMN-Graphen ausgeführt. 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 der beiden Elemente und der verletzen Regel kann eine genaue Fehlerbeschreibung gegeben werden. Dieses Verhalten ist analog für die Regel die aus Sicht des BROS-Modelles arbeiten. Was nicht mit den Prologregeln dargestellt werden kann sind erfolgreiche Regelprüfungen. Jede Regel hat zu beginn eine Implikation die die gültigkeit der Regel auf bestimmte Elemente einschränkt. Für eine optimale Rückmeldung an den Modellierer muss zwischen einer falschen Vorbedingung und einer wahren Vorbedingung mit wahrer Konsequenz unterschieden werden. Im Fall das eine Regel eine wahre Vorbedingung und eine wahre Konsequenz aufweist wird eine positive Konsistenzmeldung gespeichert.

Das Ergebnis der Konsistenzprüfung ist eine Liste von Konsistenzmeldung. Jede Konsistenzmeldung besteht aus ihrem Typ (positiv oder negativ), den betroffenen BPMN- und BROS-Element, der zugrundeliegenden Regel und einer textuellen Beschreibung. Die positiven Konsistenzmeldung helfen dem Modellierer. *TODO*

5. Implementierung der automatischen Konsistenzprüfung

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

5.1. Implementierung der Referenzarchitektur

Im Abschnitt 4.2 werden die 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 interagieren kann. Allerdings hat die Implementierung in Prolog einen entscheidenden Nachteil. Für den *Verifikations-Algorithmus* wird ein dreiwertiger Rückgabewert pro Regel benötigt. Dies lässt sich mit Prolog nicht direkt abbilden. Zudem lassen sich die Regel 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 verwenden. Kotlin ist als JVM-Sprache entwickelt wurden, kann aber auch zu Javascript und LLVM-Bytecode transpiliert werden. Dadurch besitzen Kotlin Programme eine hohe Erweiterbarkeit im Bezug auf andere Laufzeitumgebungen. Wie auch im BROS-Editor FRaMED.io wird die Transpilierung zu Javascript genutzt. Ziel der Implementierung ist es, bestehende BPMN- und BROS-Modelle, mit Hilfe des hier vorgestellten Verfahrens, auf Konsistenz zu überprüfen.

Als Dateiformat für BPMN-Modelle wird das auf XML basierende (**.bpmn*) Format von dem BPMN-Editor bpmn.io unterstützt. Der BPMN-Editor bpmn.io ist eine moderne und Webanwendung zum editieren von BPMN-Modellen. Das Dateiformat enthält sowohl Strukturinformation als auch graphische Informationen. Diese sind in zwei XML-Namespaces getrennt. Die Struktur wird im Namespace *bpmn*, und die graphische Notation im Namespace *bpmndi* gespeichert. Für die Konsistenzprüfung ist nur der Namespace *bpmn* relevant. Der Namespace *bpmndi* und alle seine Kindelemente wird 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 wurde und somit eine zu starke Bindung an

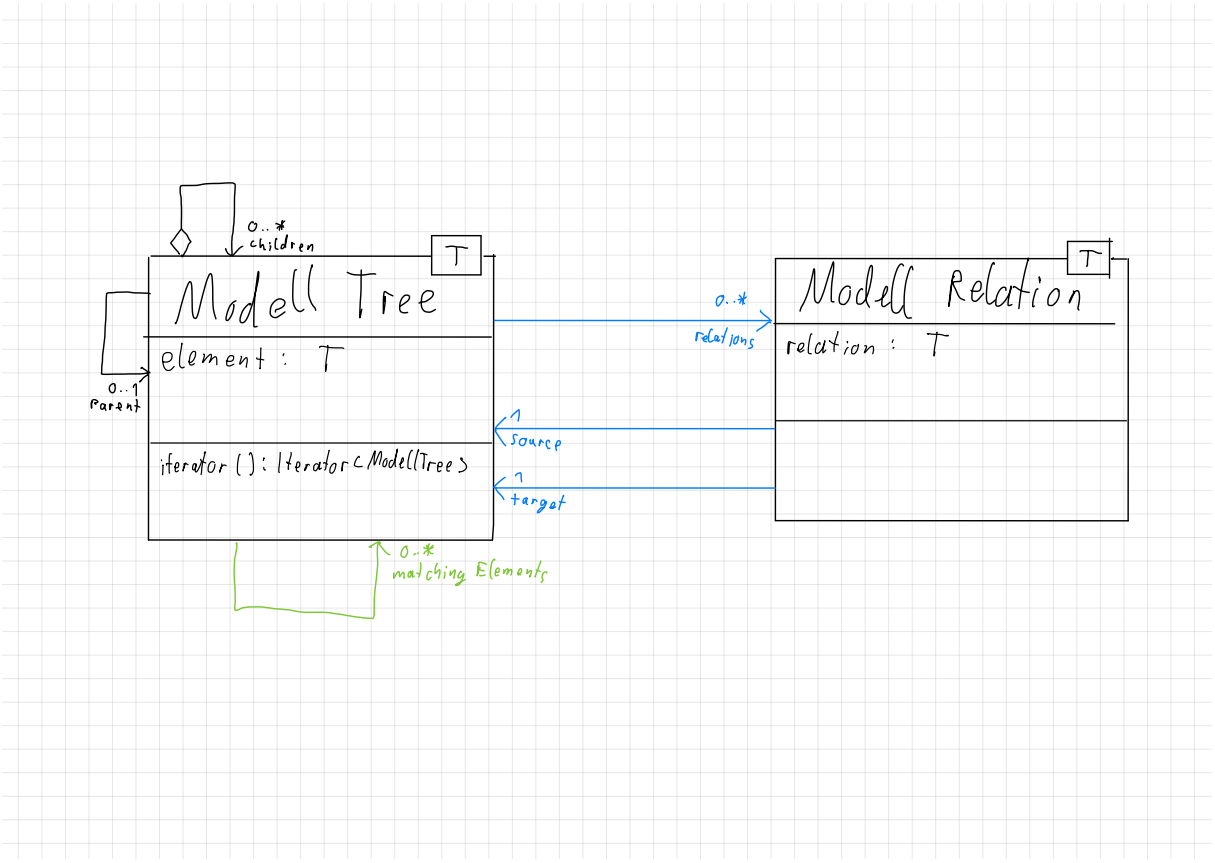

Abbildung 3.: Metamodell der Graphstruktur

die Platform Javascript erzeugt. Die integrierte Validierungsfunktion für BPMŃ-Modelle wird auch nicht benötigt da eine Vorraussetzung für das benutzen 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 Strukturinformation als auch graphische Informationen. Die graphischen Informationen befinden sich im Root-Objekt unter dem Schlüsselwort *layer*. Auch diese graphischen Informationen werden im weiteren ignoriert. Zum lesen der BROS-Dateien wird der bereits existierende Parser aus dem BROS-Editor FRaMED.io genutzt. Dieser ist in Kotlin geschrieben und benutzt den nativen JSON-Parser von Javascript. Genau wie bei dem BPMN-Parser wird auch das BROS-Modell in eine Instanz des BROS-Metamodells geladen.

Bevor der *Matching-Algorithmus* und der *Verifikations-Algorithmus* beginnen können, müssen beide Instanzen des Metamodells noch in die Form eines Graphen konvertiert werden. Die Datenstruktur des Graphen basiert auf dem Metamodell aus Abbildung 3. In dieser Abbildung sind die drei Schichten des Graphen farbig hervorgehoben. In schwarz ist die erste Schicht dargestellt. Dabei besitzt jeder Knoten eine Liste von Kind-Knoten. Jeder Knoten hat zusätzlich noch einen Verweis auf seinen Eltern-Knoten. Zusätzlich besitzt jeder Knoten noch eine Referenz auf seine Metamodell Instanz mit Typannotation. Die zweite Schicht wird in blau dargestellt un repräsentiert die Relation zwischen den Knoten. Dabei hat jeder Knoten ein Set von Verweisen auf seine Relationen. Die Relationen verweisen auf auf ihren Quell- und Zielknoten. Aus die Graphrelationen besitzen eine Referenz auf ihre Metamodell Instanz mit Typannotation. Schließlich wird die dritte Schicht, die das Matching zwischen den Modellen darstellt mit grün markiert. Dabei

$$\text{'Aktion war erfolgreich'} \text{ , } \text{'ErfolgreicheAktion'} \tag{1}$$

$$\{\text{'aktion', 'erfolgreich', 'war'}\} \text{ , } \{\text{'aktion', 'erfolgreiche'}\} \tag{2}$$

$$\{\text{'aktion', 'erfolgreiche'}\} \text{ , } \{\text{'aktion', 'erfolgreich', 'war'}\} \tag{3}$$

$$\{\text{'}\mathbf{akti}\text{on'} \subseteq \text{'}\mathbf{akti}\text{on'}\} \text{ , } \{\text{'}\mathbf{erfolgreic}\text{he'} \subseteq \text{'}\mathbf{erfolgrei}\text{ch'}\} \tag{4}$$

Abbildung 4.: Anwendung des Name-Matching

handelt es sich um ein Set von Knoten. Jeder Knoten hat zusätzlich noch eine Hilfsfunktion die einen Iterator zurückgibt. Diese implementiert eine einfache Breitensuche um über den Graphen zu iterieren.

5.2. Matching der Modelelemente

Um die Verifizierung der Modelle zu ermöglichen muss zunächst ein dazugehöriges Matching aufgebaut werden. Um den *Matching-Algorithmus* ausführen zu können muss zunächst die Orakelfunktion implementiert werden. Dabei hat es sich als ausreichend gezeigt theoretisch miteinander 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 angewendet (Vgl. Schritt 1). Zunächst werden beide Namen in ihre Bestandteile zerlegt (Vgl. Schritt 2) und nach der länge sortiert (Vgl. Schritt 3). Anschließend wird für jedes Element der ersten Menge ein passendes Element in der zweiten Menge gesucht. Die Zeichenketten müssen nicht komplett übereinstimmen. Es ist ausrechend wenn einer der beiden Strings ohne seine Endung ein Teil des anderen Strings ist. Um zu verhindern das ein sehr kurzer String in einem langen String erkannt wird, darf die Längenunterschied nicht größer als die Endungslänge sein. Wenn zu jedem Element der ersten Menge ein passendes Element der zweiten Menge gefunden wird, stimmen die beiden Namen überein (Vgl. Schritt 4).

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

```
Context.match<BpmnLane, RoleType> { 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 BPMN-Event gebildet. Hierbei ist zu beachten das die Namen der BPMN-Gateway Ausgänge nicht im Metamodell des Gateways sondern im Metamodell des weiterführenden BPMN-Flows definiert ist. Nachdem die Knoten auf gefiltert wurden, wird überprüft ob der Name eines BPMN-SequenceFlows der an das BPMN-Gateway anschließt mit dem BROS-Event übereinstimmt. Dafür bietet die Knoten-Klasse eine weitere Hilfsfunktion die die verbunden Relationen nach ihrem Typ gefiltert zurückgeben kann.

```
Context.match<BpmnGateway, Event> { bpmn, bros ->
    bpmn.relations<BpmnFlow>().any { flow ->
        flow.relation.type == BpmnFlow.Type.SEQUENCE &&
                flow.relation.name.isNotBlank() &&
                matchStrings(flow.relation.name, bros.element.desc)
    }
}
```

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

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

```
Context.match<BpmnEvent, Event> { bpmn, bros ->
    bpmn.relations<BpmnFlow>().any { flow ->
        flow.relation.type == BpmnFlow.Type.MESSAGE &&
                flow.source in bros.matchingElements ||
                flow.target in bros.matchingElements
    }
}
```

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

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

5.3. Implementierung der Konsistenzregeln

Nach dem Aufbau des Matching in der dritten Schicht, kann der *Verifikations-Algorithmus* arbeiten. Für das implementieren der Regeln stellt das Context Objekt 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 kann auch ein *null* Wert zurückgegeben werden. Ein *null* Wert bedeutet, das die Regel nicht anwendbar ist und das Ergebnis daher ignoriert werden soll. Damit wird die unter Abschnitt 4.2 beschriebende Dreiwertigkeit umgesetzt.

Um dies zu veranschaulichen werden nachfolgend die Konsistenzregeln 2, 3 und 5 in ihre Kotlin-Implementierung überführt.

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

```
Context.verifyBpmn<BpmnLane> { bpmn ->
    for (match in bpmn.matchingElements) {
        val roleType = match.model<RoleType>() ?: continue
        return@verifyBpmn Result.match("...", bros = match)
    }
    Result.error("...")
}
```

Quelltext 12: Implementierung von Regel 2

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

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

    for (match in bpmn.matchingElements) {
        val returnEvent = match.model<ReturnEvent>() ?: continue
        return@verifyBpmn Result.match("...", bros = match)
    }
    Result.error("...")
}
```

Quelltext 13: Implementierung von Regel 3

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

```
verifyBpmn<BpmnEvent> { bpmn ->
    if (
        bpmn.element.terminationEvent ||
        bpmn.element.type != BpmnEvent.Type.START
    ) return@verifyBpmn null

    for (match in bpmn.matchingElements) {
        val event = match.model<Event>() ?: continue
        val container = bpmn.container()?.second ?: continue

        val createsElement = match.relations<CreateRelationship>()
                .firstOrNull()
                ?.target as? ModelTree<ModelElement<*>> ?: continue

        if (createsElement in container.second.matchingElements) {
            return@verifyBpmn Result.match("...", bros = match)
        } else {
            return@verifyBpmn Result.error("...", bros = match)
        }
    }
    Result.error("...")
}
```

Quelltext 14: Implementierung von Regel 5

5.4. Benutzerinterface

Da das entwickelte Tool eine Webanwendung ist kann es in allen moderneren Browsern benutzt werden die Javascript aktiviert haben. Das Tool hat ein zweistufiges Interface. Als erster Schritt müssen die Quelldateien der zu überprüfenden Modelle geladen werden. Dazu können die Dateien einfach per Drag'n'Drop in das Tool geladen werden. Das Tool erkennt den Inhalt unabhängig des Namens und läd die entsprechende Datei. Alternativ kann eine manuelle Dateiauswahl genutzt werden oder der Inhalt der Datei in das entsprechende Textfeld kopiert werden. Sobald jeweils ein valides BPMN- und BROS-Modell geladen wurden, wird die Konsistenzprüfung automatisch gestartet.

Die Ergebnisse der Konsistenzprüfung werden unterhalb der Eingabemaske angezeigt. Zunächst werden statistische Informationen zu den geladen Modellen, des Matchings und der Validierung angezeigt. Diese sind in drei Abschnitte unterteilt.

- **Verification stats:** Informationen zu der aktuellen Verifikation.

- *Successful checks (x of y):* Anzahl der x positiven Konsistenzmeldung von allen y Konsistenzmeldung.
- *Errors (x of y):* Anzahl der x negativen Konsistenzmeldung von allen y Konsistenzmeldung.
- *Coverage (x%):* 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*.

Unterhalb dieser Statistiken befindet sich die Liste der Validierungsergebnisse. Mit hilfe der Tableiste kann zusätzlich das BPMN- oder BROS-Matching oder auch das geladene PredefinedMatching angezeigt werden. Jeder Eintrag der Liste hat den gleichen Aufbau. Auf der linken Seite de Eintrages ist farblich der Typ markiert. Je nach Tab gibt es unterschiedliche Farbkodierungen. Im Hauptteil des Eintrages sind drei Textfelder. Diese sind das BPMN-Element, das BROS-Element und die textuelle Beschreibung des Eintrages. Zusätzlich ist noch ein viertes Textfeld vorhanden das den Namen der Regel angibt, die diesen Eintrag erstellt hat. Mit einem klick auf das Textfeld des BPMN- oder BROS-Elementes kann direkt ein PredefinedMatching erstellt oder gelöscht werden. Die Verifizierungsergebnisse haben die Farbkodierungen grün (positive Konsistenzmeldung) und rot (negative Konsistenzmeldung). Die Matching Ergebnisse sind blau (Matching für dieses Element gefunden) und gelb (keine Matching für dieses Element gefunden) markiert. Da sich nicht alle Elemente eines Modelles auf das andere Abbilden lassen, ist ein Fehlendes Matching meißt kein Fehler. Genauso ist das vorhanden sein eines Matchings nicht immer korrekt, da das Matching auf die Namen angewiesen ist. Aus diesem Grund ist die 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 Anwenders durgeführt (Abschnitt 1) und anschließend eine neue Regeln aus sicht des Entwicklers implementiert (Abschnitt 2).

6.1. Anwendung am Beispiel einer Pizzabestellung

Dafür wird der Vorgang einer Pizzabestellung betrachtet und mit Hilfe des Tools evaluiert. Die benutzten BPMN- und BROS-Modelle basieren auf den Modellen von Schön et al. o.D.

6.2. Erweiterbarkeit des Ansatzes

Bisher wurden nur Regeln vorgestellt, die die Konsistenz von BPMN-Modellen zu BROS-Modellen prüfen. Da das BROS-Modell gegenüber dem BPMN-Modell angereichert werden kann ist dies auch die häufigste Anwendung. Allerdings können auch einige Regeln in die andere Richtung überprüft werden. Um gleichzeitig die Erweiterbarkeit des Ansatzes und der Implementierung zu zeigen wird **Regel 6** aus Abschnitt 4.2 hinzugefügt. Die Regel 6 überprüft das zu jedem BROS-Event und BROS-ReturnEvent ein dazugehöriges BPMN-Element existiert. Dabei können die BPMN Elemente von den Typen BPMN-Activity, BPMN-Gateway und BPMN-Event seien. 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 Regel befinden sich im Package *io.framed.modules*. Für die Regel 6 wird dafür die Datei *Rule6BrosEvents.kt* erstellt. Jede Datei enthält eine Funktion die einmalig zum Setup der Regeln ausgeführt wird. Dies hat die Signatur *fun Context.setupRule6BrosEvents()*. Damit wird ein Parameter definiert, der den Typ *Context* hat und als Receiver (*this*) genutzt werden kann. Innerhalb dieser Funktion können nun die Matching- und Verifikations-Funktionen definiert werden. Die explizite angabe des *Context* Receiver Parameters ist nun nicht mehr nötig, da dies durch die Setup-Funktion an innere Funktionen propagiert wird.

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

auf ein BROS-ReturnEvent gesetzt wird. Die Implementierung des Lambdas bleibt unverändert.

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

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

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

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

Quelltext 16: Implementierung von Regel 6

Ein Nachteil der Kotlin/JS Plattform ist das Fehlen von Reflection und Annotationprcessing 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 die neue Regel hinzuzufügen wird der Liste ein neuer Eintrag (*::setupRule6BrosEvents*) hinzugefügt. Der Aufruf *::* auf eine Funktion gibt eine Funktionsreferenz zurück und erlaubt eine spätere Ausführung. Die vollständige Implementierung der Regel 6 ist unter Quelltext 18 zu finden.

7. Schluss

7.1. Zusammenfassung

Ziel dieser Arbeit war die automatisierte Konsistenzprüfung von BPMN- und BROS-Modellen. Dazu wurden anhand bestehender Arbeiten ein neuer Ansatz entwickelt. Die Funktionalität dieses Ansatzes wurde anhand einer Referenzimplementierung und einer Fallstudie gezeigt.

7.2. Wissenschaftlicher Beitrag

- **Nature:** beides
- **Diagrams:** BPMN-BROS
- **Consistency Type:** *Intra-Modell (horizontale) Konsistenz*
- **Intermediate Representation:** Nein
- **Consitency Strategy:** *Monitoring* (Auf einem Regelsatz basierend)
- **Case Study:** ja
- **Automatable:** gut (H)
- **Tool Support:** ja
- **Extensibility:** gut (H)

Der Ansatz und das entwickelte Tool geben dem Modellierer Warnungen und Hinweise die auf mögliche Inkonsistenzen deuten. Dabei werden explizit keine Handlungsempfehlungen oder Lösungsmöglichkeiten gegeben. Dies liegt in der alleinigen Verantwortung des Modellierers. Des Weiteren wird keine syntaktische und semantische Konsistenzprüfung der Einzelmodelle durchgeführt. Die zu überprüfenden Modelle müssen alleinstehend Konsistent sein. Nur die Korrektheit des Dateiformats wird indirekt überprüft, da es eine Vorraussetzung zum Parsen des Modelles ist.

7.3. Zukünftige Arbeiten

Eine Abgrenzung dieser Arbeit war, das keine Handlungsempfehlungen oder Lösungsmöglichkeiten bei gefundenen Inkonsistenzen gegeben werden. In diesem Feld sollte evaluiert werden welche Inkonsistenzmuster häufig auftreten und ob diese eine Vorhersehbare Lösungsmöglichkeit aufweisen.

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

Literatur

Briand, Lionel C, Yvan Labiche und Leeshawn O'Sullivan (2003). „Impact analysis and change management of UML models". In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, S. 256–265.

Egyed, Alexander (2001). „Scalable consistency checking between diagrams-The VIEWINTEGRA approach". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001).* IEEE, S. 387–390.

– (2006). „Instant consistency checking for the UML". In: *Proceedings of the 28th international conference on Software engineering.* ACM, S. 381–390. DOI: 10.1145/1134285.1134339.

Kim, S-K und David Carrington (2004). „A formal object-oriented approach to defining consistency constraints for UML models". In: *2004 Australian Software Engineering Conference. Proceedings.* IEEE, S. 87–94. DOI: 10.1109/aswec.2004.1290461.

Loja, Luiz Fernando Batista et al. (2010). „A business process metamodel for enterprise information systems automatic generation". In: *Anais do I Congresso Brasileiro de Software: Teoria e Prática-I Workshop Brasileiro de Desenvolvimento de Software Dirigido por Modelos.* Bd. 8, S. 37–44.

Lucas, Francisco J, Fernando Molina und Ambrosio Toval (2009). „A systematic review of UML model consistency management". In: *Information and Software Technology* 51.12, S. 1631–1645. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.009.

Mens, Tom, Ragnhild Van Der Straeten und Jocelyn Simmonds (2005). „A framework for managing consistency of evolving UML models". In: *Software Evolution with UML and XML.* IGI Global, S. 1–30.

Nuseibeh, Bashar (1996). „To be and not to be: On managing inconsistency in software development". In: *Proceedings of the 8th International Workshop on Software Specification and Design.* IEEE, S. 164–169.

Rasch, Holger und Heike Wehrheim (2003). „Checking Consistency in UML Diagrams: Classes and State Machines". In: *International Conference on Formal Methods for Open Object-Based Distributed Systems.* Springer, S. 229–243. DOI: 10.1007/978-3-540-39958-2_16.

Schön, Hendrik et al. (o.D.). „Business Role-Object Specification: A Language for Behavior-aware Structural Modeling of Business Objects". In: ().

Shinkawa, Yoshiyuki (2006). „Inter-Model Consistency in UML Based on CPN Formalism". In: *2006 13th Asia Pacific Software Engineering Conference (APSEC'06).* IEEE, S. 411–418. DOI: 10.1109/apsec.2006.41.

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

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

A. Appendix

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

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

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

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

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

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

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

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

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

Quelltext 17: Algorithmus zum Name-Matching

Quelltext 18: Implementierung der Datei Rule6BrosEvents.kt