Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

FABIAN WEILBRENNER

MASTER THESIS

AUTOMATISIERTE ABLEITUNG VON VARIANTEN AUS SYSML-BASIERTEN SYSTEMARCHITEKTUREN

Eingereicht am 19. Mai 2016

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 19. Mai 2016

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 19. Mai 2016

Abstract

Product Line Engineering is used for quite some time in order to increase the reusability of architecture models. Therefore, a variant is derived by a platform model (PM) based on a feature model (FM), which abstracts the functionality of the PM. With some meta information, it is possible to create the variant automated by the PM with a tool.

This thesis surveys how a variant can be derived with as little meta information as possible by a PM, which is created with the Systems Modeling Language (SysML). For this purpose, the elements of the language are analyzed and categorized in order to abstract them. Then, language constructs has to be found for annotating the model with meta information. The constructs can be used for creating a FM. In the next step, rules are created to map these constructs to the model and to determine the membership of the remaining elements to a variant. These rules are finally transfered into an algorithm and a prototype is implemented.

Zusammenfassung

Der Ansatz der Produktlinien (engl. Product Line Engineering) wird bereits seit längerem verwendet, um die Wiederverwendbarkeit von Architekturmodellen zu erhöhen. Dafür wird aus einem Plattformmodell (PM) eine Variante abgeleitet auf Basis eines Featuremodells (FM), das die Variabilität des PM abstrahiert. Mit Hilfe von Metainformationen ist es dann möglich werkzeuggestützt automatisiert Varianten aus dem PM zu erstellen.

Diese Arbeit untersucht, wie man dies mit möglichst wenig Metainformationen aus einem Modell, das mit der Systems Modeling Language (SysML) aufgebaut wurde, erreichen kann. Dafür wurden die Sprachelemente der SysML analysiert und in Kategorien unterteilt, um diese abstrahiert betrachten zu können. Danach wurden Sprachkonstrukte für die Annotierung mit Metainformationen gefunden, um damit ein FM abzuleiten. Im nächsten Schritt wurden Regeln aufgestellt, mit deren Hilfe diese Konstrukte in das Modell abgebildet werden und daraus die Zugehörigkeit der übrigen Modellelemente zu einer Variante ermittelt werden können. Diese Regeln wurden zum Schluss in einem Algorithmus umgesetzt und in einem Prototyp implementiert.

Abbildungsverzeichnis

Listings

Inhaltsverzeichnis

1 Einleitung

Immer wieder findet man Produkte, die es in verschiedenen Ausführungen zu kaufen gibt. Wenn man sich zum Beispiel ein neues Smartphone oder einen neuen Laptop anschaffen möchte, dann ist es nicht unüblich, dass man sich von einem Modell verschiedene Variationen kaufen kann. Diese beruhen auf einer gemeinsamen Basis, die sich dann aber in einigen Eigenschaften voneinander unterscheiden. Im Beispiel des Laptops kann es sich dabei um Hardware handeln, wie die Größe des Bildschirms, des verbauten Arbeits- oder Festplattenspeichers, aber auch um die Software, das installierte Betriebssystem oder Office-Paket.

Durch die gemeinsame Basis von Funktionen einer Produktfamilie ist es nicht nötig jedes Produkt separat zu entwerfen und zu entwickeln, sondern es ist möglich diese Gemeinsamkeiten zusammenzufassen und nur in den variablen Details zu unterscheiden. Dieses Vorgehen ist unter dem Begriff Product Line Engineering (PLE) bekannt und ist eine bekannte Methode, um Produkte mit erhöhter Qualität und in geringerer Zeit herzustellen ((Clements & Northrop, 2007), (Weiss & Lai, 1999), (Bosch, 2000)). Auch das Software Engineering Institute der Carnegie Mellon University beschäftigt sich, unter anderen, in besonderer Weiße mit dem Thema der Softwareproduktlinien (SPL) (http://www.sei.cmu.edu/productlines). Sie definieren eine SPL als eine Menge von Systemen, die eine gemeinsame Menge von Features teilen, die einen spezifischen Nutzen eines bestimmten Marktes erfüllen.

Sie führen auf ihrer Webseite ebenfalls folgende Vorteile von SPL auf:

- Erhöhte Produktivität um das bis zu 10-fache
- Erhöhte Qualität um das bis zu 10-fache
- Verringerte Kosten von bis zu 60%
- Verringerter Personalbedarf von bist zu 87%
- Verringerte "time to market"von bis zu 98%
- Möglichkeit in Monaten in neue Märkte einzutreten, nicht in Jahren

Auf die genauen Zahlen wird an dieser Stelle nicht eingegangen, da diese auf jeden Fall diskussionswürdig sind. Fest steht, dass die aufgeführten Punkte nicht von der Hand zu weisen sind.

Kang, Lee und Donohoe (2002) führen in ihrem Artikel die so genannte Feature Oriented Reuse Method (FORM) ein. Diese beschreibt eine Methode um für ein Produkt ein FM zu entwickeln, mit dessen Hilfe eine konzeptuelle Architektur mit wiederverwendbaren Bestandteilen entwickelt werden kann, um wiederum verschiedene Variationen dieses Produktes erstellen zu können.

Lee und Kang (2006) gehen sogar noch einen Schritt weiter, indem sie bereits veröffentlichte Produkte dynamisch konfigurieren wollen mit dem Fokus auf Produktfamilien. Auch hier wird zunächst ein Featuremodell entwickelt, das die Funktionalität der Produktfamilie abstrahiert. Danach werden sogenannte "service features" identifiziert, indem über das FM traversiert wird. Diese werden zu einer "binding unit" zusammengefasst. Das eigentlich Modell ist an den C2-Architekturstil angelehnt, welcher auf Komponenten und Nachrichten beruht.

Atkinson (2002) behandelt in seinem Buch das PLE mit Hilfe von KobrA, was ein Akronym für Komponentenbasierte Anwendungsentwicklung ist. Die Methode unterteilt den Entwicklungszyklus in zwei Hauptteile. Zum einen wird ein Framekwork entwickelt, welches eine wiederverwendbare Menge von Softwareartefakten darstellen soll. Daraus können dann im zweiten Teil konkrete Anwendungen entwickelt werden. Die logischen Bestandteile werden in Hierarchien von Komponenten aufgebaut, welche Abhängigkeiten untereinander aufzeigen.

Pohl, Böckle und Van der Linden (2005) sehen ebenfalls zwei Entwicklungsprozesse. Auch hier wird eine wiederverwendbare Plattform gebaut, die dann im zweiten Schritt verwendet wird um davon konkrete Applikationen abzuleiten. Dieser Ansatz wird von der Masterarbeit aufgegriffen und zeigt eine konkrete Umsetzung auf Architekturebene, zugeschnitten auf SysML-basierte Modelle.

Eine Übersicht über verschiedene Designmethoden des PLE sind im Artikel von Matinlassi (2004) zu finden.

Diese Arbeit beschäftigt sich mit Architekturen. Jedes software-basierte System besitzt eine Architektur, unabhängig davon, ob sie bewusst entworfen wurde oder während der Entwicklung unfreiwillig entstanden ist. Diese Architektur beinhaltet nicht nur die Software, sondern kann Hardware oder auch mechanische Elemente besitzen. Im Idealfall ist die Architektur bewusst und zielführend entwickelt worden. Um nun das PLE auf Architekturebene zu bringen, kommt der Einsatz eines Plattformmodells (PM) in Kombination mit so genannten Featuremodellen (FM) zum Tragen, die während der Entwicklungsphasen erstellt werden. Hierzu wird das PM mit Metainformationen bestückt, mit deren Hilfe dann eine Variante aus dem PM abgeleitet werden kann. Durch entsprechende Werkzeugunterstützung soll diese Ableitung automatisiert und somit auch der Aufwand und

die Fehlerträchtigkeit gesenkt werden. Ein FM ist eine hierarchische Abstraktion der Features, die ein PM anbietet. Das eigentliche PM beinhaltet alle möglichen Kombinationen, um die Features des FM realisieren zu können.

Nun soll untersucht werden, wie man mit möglichst wenigen Metainformationen automatisiert eine Variante aus dem PM ableiten kann. Diese Plattformmodelle sind in der Systems Modeling Language (SysML) der Object Management Group (OMG) modelliert. Dazu ist es nötig im ersten Schritt die Sprache und deren Elemente zu analysieren. Das Ziel hierbei ist die Bestimmung von Sprachkonstrukten, die für die Annotierung mit Metainformationen geeignet sind. Dies entspricht der Abbildung der Funktionalitäten eines PM auf ein Featuremodell. Im zweiten Schritt folgt die Untersuchung, wie man, ausgehend von diesen Sprachkonstrukten, die zugehörigen übrigen Modellelemente zu einer Variante bestimmen kann.

Hierzu sollen Regeln für die einzelnen Sprachelemente bzw. –konstrukte erstellt werden, wie man mit diesen Elementen die Zugehörigkeit eines mit Metainformationen annotieren Modellelementes bestimmen kann. Anschließend werden diese Regeln in einen Algorithmus umgesetzt und implementiert.

Als begleitendes Beispiel kommt der modulare Roboter “Dagu Rover 5“(http://www.dagurobot.com/) zum Einsatz. Dieser kann, je nach Wunsch, aus verschiedenen Bestandteilen zusammengesetzt werden. Zum Beispiel können verschiedene Antriebe, Sensoren oder Controller eingesetzt werden. Ein Experte hat dafür ein PM für den Dagu Rover erstellt, das als Testmodell für die Evaluierung dieser Arbeit dient.

1.1 Anwendungsszenarien

Das Ergebnis der Arbeit findet auch Anwendungen in der Industrie. Die Erstellung von PM und die Ableitung von Varianten dieser PM ist sehr zeitaufwendig, wenn sie manuell durchgeführt werden müssen. Um dabei Ressourcen zu sparen und gleichzeitig die Fehleranfälligkeit zu senken, ist es sinnvoll diesen Schritt automatisiert mit Hilfe eines Werkzeugs zu machen. Beispielhaft werden nachfolgend zwei Szenarien vorgestellt, in denen der Algorithmus verwendet werden kann. Wenn man diesen dort einsetzen würde, könnte man die Entwicklungszeit eines neuen Produktes reduzieren.

1.1.1 Modulare Querbaukasten von VW

Ein Ziel eines Unternehmens ist es immer den Gewinn zu maximieren. Entweder man steigert hierfür den Umsatz oder reduziert die Kosten. Den zweiten Ansatz

verfolgte Volkswagen mit dem Modulare Querbaukasten (MQB). Jedes Auto hat ein prinzipiell ähnliches Grundgerüst, einzelne Fahrzeugklassen haben gemeinsame Elemente und unterscheiden sich in den konkreten Modellen in Details.

Der MQB ist ein System für den modularen Zusammenbau von Automobilen mit quer eingebauten Motoren. Namhafte Vertreter sind der Audi A3 8V oder der VW Golf VII. Es gibt eine Grundarchitektur, die allen Fahrzeugen gemein ist. Mit dem MQB lassen sich nun aus vorgegebenen Features neue Fahrzeugmodelle erstellen.

Da nun nicht mehr jede Architektur eines Autos einzeln erstellt werden muss, sondern aus einem gemeinsamen Pool von Elementen stammt, lassen sich neue Modelle wegen der Wiederverwendung in kürzerer Zeit bauen. Dies steigert die Profitabilität durch gesenkte Produktionskosten. Zudem senkst es die Fehleranfälligkeit durch eine robuste Grundarchitektur.

Der Algorithmus kann nun bei der Generierung eines neuen Fahrzeugs verwendet werden. Aus einem Gesamtmodell, das in diesem Fall in der Systems Modeling Language (SysML) vorliegen muss, wird dann je nach den Featureangaben aus dem Baukasten die erforderliche Variante extrahiert.

1.1.2 Handys/Smartphones

Als Nokia noch Handymodelle wie das Nokia 3310 produziert hat, konnte man bereits feststellen, dass die Handys dieser Zeit alle fast gleich aussahen. Das eine Modell hatte lediglich ein Farbdisplay zusätzlich, beim anderen kam eine eingebaute Kamera dazu. Der sonstige Aufbau war ansonsten identisch.

Heutzutage haben die Smartphones den Markt erobert, aber auch dort ist dieses Phänomen zu erkennen. In regelmäßigen Abständen kommt das neueste Modell des Samsung Galaxy auf den Markt oder Microsoft veröffentlicht ein neues Gerät ihrer Lumia-Reihe.

Diese haben oft den gleichen Aufbau wie eines ihrer Geschwister, sie bekommen lediglich eine andere Displaygröße, einen stärkeren Prozessor oder einen erweiterten Speicher. Das Lumia 950 von Microsoft zum Beispiel kommt in zwei Ausführungen, die Standardvariante und das XL-Modell. Diese sind annähernd identisch, sie unterscheiden sich elementar nur im Prozessor, der Displaygröße und der Akkukapazität. Wenn ein Systemmodell nun in SysML vorliegen würde, wäre man in der Lage die Varianten aus dem Plattformmodell zu erstellen.

1.2 Vergleich zu existierender Software

Da das Product Line Engineering inzwischen ein etabliertes Verfahren ist, gibt es bereits Softwarelösungen, die sich mit dem Thema auseinandersetzen. In diesem Abschnitt wird auf einige Produkte eingegangen, was deren Features sind und ggf. mit dem eigenen Ansatz verglichen. Die folgende Liste hat keinen Anspruch auf Vollständigkeit, sondern soll vielmehr einen Auszug darstellen.

1.2.1 pure::variants

pure::variants von der pure-systems GmbH (http://www.pure-systems.com) unterstützt das Variantenmanagement über alle Phasen des Entwicklungsprozesses eines Systems. Dabei ist es ein offenes Framework basierend auf Eclipse, um mit anderen Werkzeugen wie z.B. Codegeneratoren oder Anforderungstools zu interagieren. Die Anwendung ist in verschiedenen Ausführungen erhältlich. Diese unterstützen Modelle unterschiedlichster Hersteller, z.B. für AUTOSAR- oder Enterprise Architect Modelle.

In der Applikation ist es ebenfalls möglich Featuremodelle zu erstellen. Diese FM werden im nächsten Schritt mit den vorhandenen Modellen verknüpft. Prinzipiell ist es mit pure::variants auch möglich die FM z.B. mit Anforderungen zu verknüpfen. Da sich diese Arbeit allerdings auf Systemarchitekturen auseinandersetzt, werden diese Aspekte ausgeblendet.

Der Arbeitsablauf ist also im Grunde ähnlich zu dem eigenen Entwurf. Es wird ein FM erstellt, mit dessen Hilfe das Architekturmodell bearbeitet wird. Dieses abgeleitete Modell kann dann im Anschluss abgespeichert werden. Die Erstellung der Varianten funktioniert in pure::variants über das Hinzufügen von Metainformationen für jedes Element, das von der Aktivierung eines Features abhängig ist. Hier liegt der elementare Unterschied zum eigenen Algorithmus. Hier ist das Ziel eine Lösung zu finden, die mit möglichst wenig Metainformationen auskommt. Aus diesen Informationen wird das Architekturmodell traversiert. pure::variants bedingt jedoch die Annotierung jedes variablen Elementes.

1.2.2 Gears

Gears von BigLever Software (http://www.biglever.com) verspricht ebenfalls Hilfe bei der Umsetzung von Produktlinien, von der Anforderungsanalyse bis hin zum Testen von Quellcode. Auch in diesem Abschnitt liegt jedoch der Fokus auf der Systemarchitektur. Der Produktbeschreibung zufolge benötigt die Variantenentwicklung drei Elemente. Zum einen muss von den Systemarchitekten

ein konfigurierbares Modell erstellt werden. Des weiteren hat das Produktlinien-Management die Aufgabe ein Featureprofil zu erstellen. Diese Komponenten werden im letzten Schritt dem “Gears Product Configurator“ übergeben und dieser erstellt dann die nötigen Varianten.
Zusätzlich bietet der Hersteller eine API an, die PLE Bridge API, mit deren Hilfe Werkzeughersteller Variationspunkte in ihren eigenen Modellen definieren können.

1.2.3 EMF Feature Model

Das EMF Feature Model (https://projects.eclipse.org/projects/modeling.emft.featuremodel) ist eine Open-Source-Software, die das Eclipse Modeling Framework (EMF) von Eclipse nutzt. Es versucht einen Standard für die Repräsentation eines FM zu etablieren. Das Ziel dieses Plugins ist es Featuremodelle und Varianten zu entwickeln, die mit einem Editor erstellt werden können. Diese FM können dann von Anwendungen weiterverwendet werden, die ebenfalls das EMF verwenden.
Zudem gibt es ein Framework zum Evaluieren der erstellten Modelle. Eine wichtige Designentscheidung des Projektes ist es, dass die Editoren und das Evaluations-Framework erweiterbar sein sollen, damit sie für eigene Bedürfnisse angepasst werden können.

In Zukunft wäre es möglich diese Software mit dem hier vorgestellten Algorithmus zu verbinden, da die einfache und visuelle Erstellung eines FM mit einem Editor nicht in den Rahmen dieser Arbeit gehört. Der letzte Release hingegen war mit der Version 0.9.0 am 28. Februar 2014, weswegen das Projekt von der Eclipse-Webseite inzwischen als archiviert eingestuft wurde. Dies bedeutet, dass es in Zukunft evtl. keine Updates mehr geben wird.

1.2.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass die Anwendungen pure::variants und BigLever Software Gears am ehesten in die Richtung geht, die diese Arbeit verfolgt. Jedoch ist die Zuweisung von Architekturelementen sehr statisch und erfordert ggf. sehr hohen Eingriff in das Modell. Somit ist keines der vorgestellten Anwendungen in der Lage, das Ziel der Arbeit zu erfüllen.

1.3 Ziel der Arbeit

Das Ziel der Arbeit ist die Entwicklung eines Algorithmus, mit dessen Hilfe man in der Lage ist aus einem Plattformmodell, beschrieben in SysML, dynamisch eine Variante zu extrahieren. Das bringt neben den Vorteilen, die bereits unter Punkt 1.1 erwähnt wurden, den großen Vorteil, dass das PM nur geringfügig angepasst werden muss, da die notwendigen Metainformationen auf ein möglichst kleines Maß reduziert werden. Somit ist es möglich bereits existierende Modelle mit weniger Aufwand umbauen, um mit dem Algorithmus funktionieren zu können. Zudem kann er an verschiedenen Stellen eingesetzt werden, z.B. bei PLE-Prozessen, die in der Einleitung aufgeführt sind. Die Arbeit kann als Einstiegspunkt für weitere Forschungen verwendet werden, um den Algorithmus zu verfeinern oder anzupassen.

Um das Ziel zu erreichen, muss zunächst die Modellierungssprache SysML analysiert werden. Dabei wird eine Liste erstellt mit Sprachkonstrukten, die dafür verwendet werden können, um das Modell mit Metainformationen zu erweitern. Im nächsten Schritt muss untersucht werden, wie man anhand dieser Metainformationen bestimmen kann, welche Elemente des Plattformmodells in die Variante übernommen werden sollen. Die Regeln, die aus den vorangegangenen Schritten erstellt werden, werden dann letztlich in einem Algorithmus umgesetzt. Der Fokus dabei liegt auf dem eigentlichen Modell und nicht auf den Diagrammen. Die Diagramme werden dementsprechend aus der Arbeit ausgeklammert und nicht bearbeitet. Zur Evaluierung dient das Beispielmodell Dagu Rover 5, um das Erreichen der Ziele zu beurteilen. Im Folgenden werden die Arbeitsergebnisse aufgelistet, die am Ende der Arbeit vorliegen sollen.

Literaturrecherche

- Recherche zum Product Line Engineering für Software (mit Bezug zu Modellierungssprachen)

Analyse der Modellierungssprache SysML

- Liste von Sprachkonstrukten, die verwendet werden können um ein Modell mit Metainformationen zu erweitern.
- Erstellung von Regeln, die sich aus Sprachkonstrukten bzw. deren Konstellationen zusammensetzen, mit deren Hilfe man ableiten kann, ob ein Element zu einer Variante gehört.
- Diskussion der Regeln

Implementierung des Algorithmus

- Umsetzung der Regeln in einem Algorithmus, die mit Eclipse Papyrus modelliert wurden
- Test-Suite für den Algorithmus

Evaluierung

- Testen des Algorithmus anhand eines Beispielmodells
- Diskussion der Ergebnisse

1.4 Struktur der Arbeit

In Kapitel 2 beschäftigt sich die Arbeit mit der Modellierungssprache Systems Modeling Language der OMG. Nach einer kurzen Einführung und Beschreibung der Spracheigenschaften folgt eine Analyse der Modellierungselemente. Aus der Untersuchung folgt eine Kategorisierung dieser Elememente, für die die Regeln, die zur Algorithmuserstellung dienen, entworfen werden.

Kapitel 3 stellt die Sprachkonstrukte vor, die für die Annotierung mit Metainformationen verwendet werden und gibt einen Einblick in die Regeln für die Traversierung des Architekturmodells. Desweiteren wird die Struktur des Beispielmodells des Dagu Rover 5 vorgestellt.

Das Design des Algorithmus ist das zentrale Thema von Kapitel 4. Der Algorithmus wird in seine verschiedenen Phasen aufgeteilt und genauer beschrieben.

Eine weitere Aufgabe ist die Implementierung der in Kapitel 3 und 4 eingeführten Regeln und Designentscheidungen. Dies wird in Kapitel 5 beschrieben. Zunächst gibt es einen Einblick in die verwendeten Technologien und Werkzeuge, die zur Umsetzung verwendet werden. Danach folgt die Beschreibung der eigentlichen Implementierung.

In Kapitel 6 werden zunächst die erstellten Regeln in Form des implementierten Prototyps getestet. Anschließend werden die gewählten algorithmischen Ansätze und Ergebnisse diskutiert.

Abschließend wird die Arbeit in Kapitel 7 zusammengefasst.

2 Die Modellierungssprache SysML

Die OMG Systems Modeling Language (OMG SysML), im weiteren als SysML bezeichnet, ist eine im Jahre 2001 von der Object Management Group (OMG; http://www.omg.org/spec/SysML) ins Leben gerufene Modellierungssprache, die sich zum Ziel gesetzt hat komplexe Systeme analysieren und designen zu können. Die Sprache kann hierbei nicht nur Hardware und Software beschreiben, sondern auch Eigenschaften wie Prozesse, Informationen und Personal. Dieser Ansatz versucht den Einsatz verschiedener Modellierungssprachen einzuschränken, indem sie verschiedene Sprachen vereint, was sich positiv auf den gesamten Entwicklungsprozess des Systems auswirken kann.

SysML basiert dabei auf einer Teilmenge der Beschreibungssprache UML (http://www.omg.org/spec/UML) und erweitert diese um Elemente, die für das Systems Engineering hilfreich sind. Desweiteren möchte SysML möglichst einfach bleiben, um weitere positive Effekte im Verstehen der beschriebenen Systeme zu erwirken.

Der Zusammenhang zwischen den beiden Sprachen ist in Abbildung 2.1 dargestellt. Die Schnittmenge beschreibt dabei die Sprachelemente von UML, die von SysML verwendet werden.

Im weiteren Verlauf des Kapitels sollen nun die Sprachelemente von SysML untersucht und gerprüft werden, ob bzw. wie sie sich zur Entwicklung des Algorithmus eignen. Ziel des Kapitels ist die Kategorisierung der einzelnen Sprachelemente. Die Kategorien sind übersichtlich in Tabelle 2.1 dargestellt. Welches Element sich in welcher Kategorie befindet ist in den Tabellen 2.2, 2.3, 2.4, 2.5, 2.6 und 2.7 nochmals übersichtlich aufgelistet.

Grundlage der Sprachanalyse ist die SysML-Spezifikation von OMG in der Version 1.4 (OMG, 2015), die im September 2015 erschienen ist. SysML besteht aus Knoten- und Kantenelementen. Aus diesen Elementen werden Diagramme erstellt, die im Grunde einen gerichteten Graphen darstellen. Die Analyse wird nach einem selbst erstellten Template durchgeführt, das für jedes Sprachelement ausgefüllt wird. Dieses ist nachfolgend beschrieben:

1. **Beschreibung:** Hier wird der allgemeine Zweck des Elementes beschrieben.

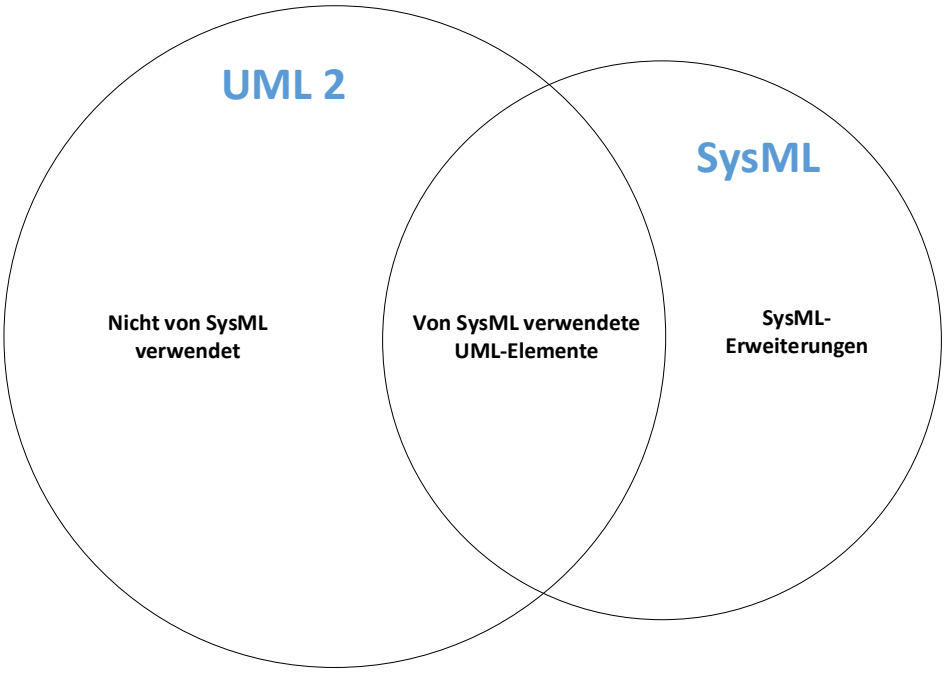


Abbildung 2.1: Zusammenhang zwischen SysML und UML

2. **Kategorie:** Einige Elemente haben ähnliche Eigenschaften, wie sie von dem Algorithmus behandelt werden müssen. Somit muss in der Implementierung nicht jedes Element einzeln betrachtet werden, sondern es kann von den konkreten Elementen abstrahiert und somit allgemeingültige Regeln für mehrere Elemente gefunden werden, was den Algorithmus schlanker macht.
3. **Papyrus-Unterstützung:** Dieser Punkt ist für den Werkzeugeinsatz wichtig. Als Tool wird das auf Eclipse (https://eclipse.org) basierende Papyrus (https://eclipse.org/papyrus) eingesetzt. Die Spezifikation 1.4 wurde erst im September 2015 veröffentlicht. Dies hat zur Folge, dass eventuell manche Elemente von diesem Tool noch nicht unterstützt werden bzw. laut Spezifikation nicht mehr vorhandene Elemente noch verwendbar sind.
4. **Abstraktion:** Da SysML auf UML basiert, ist es, in der Regel, möglich die Elemente einer Metaklasse von UML zuzuordnen. Zum Beispiel basiert ein Block auf der UML-Klasse, welche wiederum auf der abstrakten Metaklasse Classifier aufbaut. So ist es möglich Sprachelemente einfacher zu kategorisieren und zu abstrahieren, um die Anzahl an Regeln so gering wie möglich zu halten.
5. **Syntax:** Zu jeder Auflistung gehört zudem noch eine Abbildung, die die Syntax zeigt, wie ein Element laut OMG aussehen sollten.

Name	Beschreibung
Basiselement	Hauptelemente, mit denen der Algorithmus arbeitet
Strukturelement	Elemente zum Strukturieren des Systems
Diagramm	Ein Diagrammelement zum Strukturieren des Systems
Annotiertes Element	Element, das an ein anderes Element (meinst Basiselemente) angehängt wird
Bidirektionale Beziehung	Beschreibt eine Kante zwischen Elementen, die in beide Richtungen durchlaufen werden kann
Unidirektionale Beziehung	Beschreibt eine Kante zwischen Elementen, die nur in eine Richtung durchlaufen werden kann
Property	Eine Eigenschaft innerhalb eines Basiselements

Tabelle 2.1: Elementkategorien

Im Weiteren folgt die Vorstellung der einzelnen Kategorien im Detail inklusive Beschreibung eines Vertreters dieser Kategorie. Der restliche Teil der Sprachelemente sind im Anhang A zu finden. Die Einteilung der Elemente erfolgte in einem iterativen Prozess, indem diese nach und nach verfeinert und in die jeweiligen Kategorien zusammengefasst wurden. Die Ports wurden beispielsweiße zu Beginn als eigenen Kategorie angesehen, doch diese funktionieren wie Properties, weswegen die Kategorie “Port“ in die Kategorie “Property“ integriert wurde.

Jedes Element kann zusätzliche Restriktionen haben, welchen Inhalt innerhalb des Elements zulässig sind oder wie der interne Aufbau des Elementes auszusehen hat. Sofern diese Vorgaben nicht notwendig für den Algorithmus sind, werden diese ignoriert. Sie können bei Bedarf in der Spezifikation nachgelesen werden.

2.1 Kategorie “Basiselement“

Basiselemente sind die Kernelemente, auf denen die Traversierung stattfindet, die in Kapitel 3 beschrieben ist. Sie besitzen keine besonderen Eigenschaften oder Einschränkungen hinsichtlich der Bearbeitung im Algorithmus. In Tabelle 2.2 sind alle Elemente aufgelistet, die zu der Kategorie gehören. Der Block, der das wichtigste Element von SysML darstellt, wird nachfolgend vorgestellt.

1. **Beschreibung:** Der Block ist das zentrale Strukturelement. Ein Block beschreibt nicht nur eine Gruppierung zusammengehöriger Features (basierend auf den Klassen in UML), sondern dient auch zur Modellierung von

sonstigen Belangen. Dies macht den Block zu einer Art universellem Modellelement.

2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 2.2

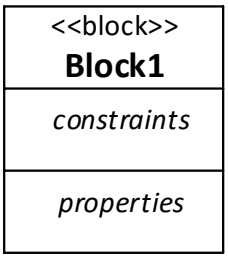


Abbildung 2.2: Syntax Block

SysML-Sprachelemente in der Kategorie "Basiselement"
Block, ValueType, Actor, Enumeration, InstanceSpecification, InterfaceBlock, Interface, AbstractDefinition, Unit, QuantityKind, Stakeholder, View, Viewpoint

Tabelle 2.2: SysML-Elemente in Kategorie "Basiselement"

2.2 Kategorie "Strukturelement"

Strukturelemente sind dafür da, um die Modelle weiter zu strukturieren bzw. zu gruppieren. Sie unterscheiden sich zu den Basiselementen in dem Punkt, dass sie weitere Struktur- oder Basiselemente enthalten können. In Tabelle 2.3 sind alle Elemente aufgelistet, die zu der Kategorie gehören. Das wichtigste Beispiel dafür ist das Package, das nun beschrieben wird.

1. **Beschreibung:** Ein Package dient zur Gruppierung von Elementen, die in irgendeiner Weise miteinander zu tun haben. Das PackageWithNameInTab und PackageWithNameInside haben dieselbe Semantik mit dem Unterschied, dass sie anders dargestellt werden.

2. **Kategorie:** Strukturelement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Package
5. **Syntax:** Abbildung 2.3

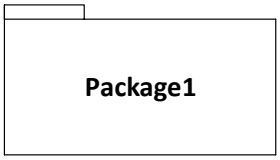


Abbildung 2.3: Syntax Package

SysML-Sprachelemente in der Kategorie "Strukturelement"
Package, Model

Tabelle 2.3: SysML-Elemente in Kategorie "Strukturelement"

2.3 Kategorie "Bidirektionale Beziehung"

Bidirektionale Kanten sind ungerichtete Kanten, die in beide Richtungen betreten werden können. Somit ist es unwichtig, von welcher Seite man die Kante erreicht, sie werden immer in die Variante mit aufgenommen, sofern ein Element, mit dem es verbunden ist, auch zur Variante gehört. Das andere Ende wird dann ebenfalls übernommen. Ausnahme davon ist die Zugehörigkeit zu den nicht ausgewählten Features eines der End-Elemente. In Tabelle 2.4 sind alle Elemente aufgelistet, die zu der Kategorie gehören. Die ReferenceAssociation dient hier als Beispiel.

1. **Beschreibung:** Die ReferenceAssociation ist das Pendant zu der normalen Assoziation in UML. Sie verbindet zwei Elemente, die in irgendeiner Form in Beziehung stehen.
2. **Kategorie:** Bidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 2.4

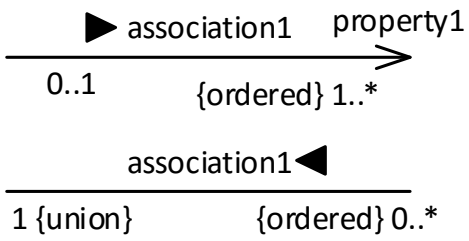


Abbildung 2.4: Syntax ReferenceAssociation

SysML-Sprachelemente in der Kategorie "Bidirektionale Beziehung"
ReferenceAssociation, PartAssociation, SharedAssociation, BindingConnector, BidirectionalConnector

Tabelle 2.4: SysML-Elemente in Kategorie "Bidirektionale Beziehung"

2.4 Kategorie "Unidirektionale Beziehung"

Eine Unidirektionale Beziehung verhält sich ähnlich zu einer Bidirektionalen Beziehung mit dem Unterschied, dass sie nur in eine Richtung betreten werden kann. Somit wird wird die Beziehung auch nur in die Variante übernommen, wenn sie in die korrekte Richtung betreten wird. In Tabelle 2.5 sind alle Elemente aufgelistet, die zu der Kategorie gehören. Die Dependency wird nachfolgend vorgestellt.

1. **Beschreibung:** Eine Dependency beschreibt die Abhängigkeitsbeziehung eines Elementes zu einem anderen. Es wird eingesetzt, wenn ein Element das andere Element benötigt für seine Spezifikation oder Implementierung. Die Dependency besteht aus einem Client und einem Supplier. Der Supplier bietet Elemente an, die der Client braucht.
2. **Kategorie:** Unidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 2.5

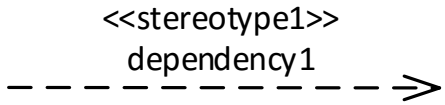


Abbildung 2.5: Syntax Dependency

SysML-Sprachelemente in der Kategorie "Unidirektionale Beziehung"
Dependency, Generalization, GeneralizationSet, PartAssociation, SharedAssociation, BindingConnector, UnidirectionalConnector, PublicPackageImport, PrivatePackageImport, PackageContainment, Realization, Refine, BlockNamespaceContainment, Conform, Expose

Tabelle 2.5: SysML-Elemente in Kategorie "Unidirektionale Beziehung"

2.5 Kategorie "Annotiertes Element"

Ein Annotiertes Element wird an eines oder mehrere Elemente angehängt, um mehr Informationen zu diesen darzustellen. Man könnte annehmen, dass diese genauso behandelt werden wie ein Basiselement, dies ist allerdings nicht ganz richtig. Ein annotiertes Element, wie z.B. der Comment, kann mehreren Elementen angehören, die allerdings nicht alle zu der Variante gehören. Somit ist es nicht möglich einfach über das Element zu iterieren. In Tabelle 2.6 sind alle Elemente aufgelistet, die zu der Kategorie gehören.

1. **Beschreibung:** Ein Comment enthält zusätzliche Informationen zu einem Element in textueller Form.
2. **Kategorie:** Annotiertes Element
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Comment
5. **Syntax:** Abbildung 2.6

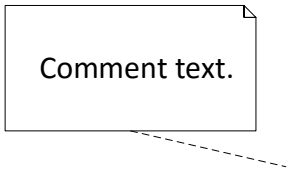


Abbildung 2.6: Syntax Comment

SysML-Sprachelemente in der Kategorie "Annotiertes Element"
Comment, ConstraintNote, ConstraintTextualNote, Problem, Rationale, ElementGroup

Tabelle 2.6: SysML-Elemente in Kategorie "Annotiertes Element"

2.6 Kategorie "Property"

Eine Property ist ein Element, das einem Basiselement zugeordnet ist. Sie können nicht alleine stehen. Somit werden sie immer dann in die Variante übernommen, wenn das Basiselement auch in der Variante vorhanden ist. Das hier gezeigte Beispiel ist das gleichnamige Sprachelement. In Tabelle 2.7 sind alle Elemente aufgelistet, die zu der Kategorie gehören.

1. **Beschreibung:** Properties sind Eigenschaften und können für einen Block definiert werden. Es gibt vier Kategorien: parts, references, values und contraints.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 2.7

SysML-Sprachelemente in der Kategorie "Property"
Property, FlowProperty, Required and Provided Features, ItemFlow, BehaviorCompartmnet, NamespaceCompartment, StructureCompartment, StereotypePropertyCompartment, BoundReference, PropertySpecificType, ParticipantProperty, ConnectorProperty, ActorPart, Port, Port (Nested), ProxyPort, FullPort, Required and Provided Interfaces

Tabelle 2.7: SysML-Elemente in Kategorie "Property"

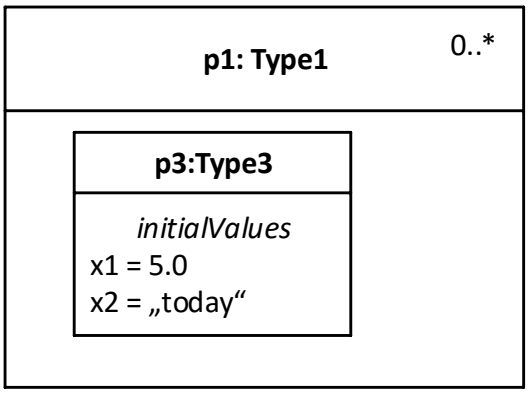


Abbildung 2.7: Syntax Property

2.7 Kategorie “Diagramm“

Diagramme sind Teil eines SysML-Modells und bieten eine Schnittstelle, um die Zusammenhänge zwischen einzelner Elemente visualisieren zu können. Die Diagramme sind nicht Teil des Umfangs dieser Arbeit, deswegen bleiben sie im weiteren Verlauf unbeachtet, sind hier allerdings der Vollständigkeit halber auch aufgeführt. In Tabelle 2.8 sind alle Elemente aufgelistet, die zu der Kategorie gehören.

1. **Beschreibung:** Das BlockDefinitionDiagram ist der Diagrammrahmen und zugleich der primäre Diagrammtyp der SysML-Erweiterung. Es basiert auf dem Klassendiagramm von UML mit allen Erweiterungen bzw. Einschränkungen von SysML. Es beinhaltet also Blöcke und ihre Beziehungen zueinander, wie Assoziationen, Abhängigkeiten oder Generalisierungen.
2. **Kategorie:** Diagramm
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Strukturdiagramm
5. **Syntax:** Abbildung 2.8

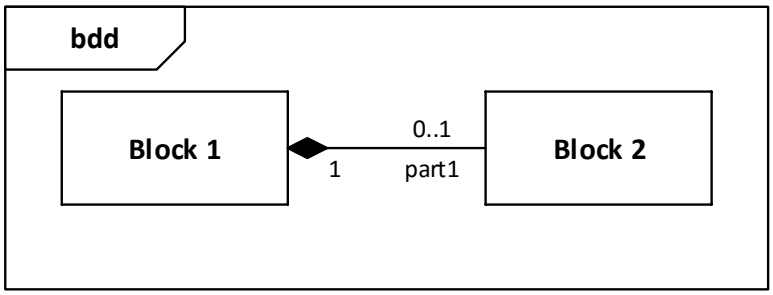


Abbildung 2.8: Syntax BlockDefinitionDiagram

SysML-Sprachelemente in der Kategorie “Diagramm“
BlockDefinitionDiagram, InternalBlockDiagramm, ActivityDiagram, SequenceDiagram, StateMachineDiagram, UseCaseDiagram, PackageDiagram, RequirementDiagram

Tabelle 2.8: SysML-Elemente in Kategorie “Diagramm“

3 Automatisierte Ableitung von Varianten

Nachdem im letzten Kapitel die Sprachelemente von SysML analysiert wurden, müssen nun Regeln erstellt werden, um die Annotierung mit Metainformationen im Plattformmodell durchführen zu können. Dazu soll eine Liste mit Sprachkonstrukten erstellt werden mit Elementen, die sich für diese Annotierung eignen. Anschließend werden die Regeln definiert, wie man ausgehend von den gefundenen Konstrukten die Zugehörigkeit eines Modellelementes zu einer Variante ableiten kann.

Um die Evaluierung des Algorithmus in Kapitel 6 durchführen zu können, wurde ein Beispielmodell entwickelt, welches als Plattformmodell dienen soll. Dieses Modell wird in diesem Kapitel eingeführt und dessen Struktur beschrieben. Im Zuge dessen sind hier auch einige Beispielkonstellationen aufgeführt, um die Verwendung der Stereotypes darzustellen.

3.1 Ableitung von Regeln für den Algorithmus

Im ersten Schritt sind geeignete Sprachkonstrukte zur Annotierung mit Metainformationen zu suchen, damit man in der Lage ist ein Featuremodell zu definieren. Mit diesem FM können dann Varianten zusammengestellt werden.

3.1.1 Sprachkonstrukte zur Annotierung mit Metainformationen

Für die Definition von Sprachkonstrukten werden, unter anderem, Stereotypes verwendet, um eine gewisse Semantik zu generieren, die speziell auf die Bedürfnisse des Algorithmus angepasst ist. Diese werden in einem eigenen Profil angelegt. In Abbildung 3.1 ist ein Überblick über das verwendete Profil zu sehen. Die einzelnen Stereotypes sind in Tabelle 3.1 erläutert.

Name	Beschreibung
Feature	Wird an eine Class angehängt; Abstraktion einer Funktionalität des PM
Variant	Wird an eine Class angehängt; Abstraktion einer Variante, das aus mehreren Features besteht
select	Wird an eine Dependency angehängt; Verweis eines Features in die funktionale Architektur des PM
include	Wird an eine Dependency angehängt; Verweis der Zugehörigkeit eines Features zu einer Variante

Tabelle 3.1: Überblick – Stereotypes

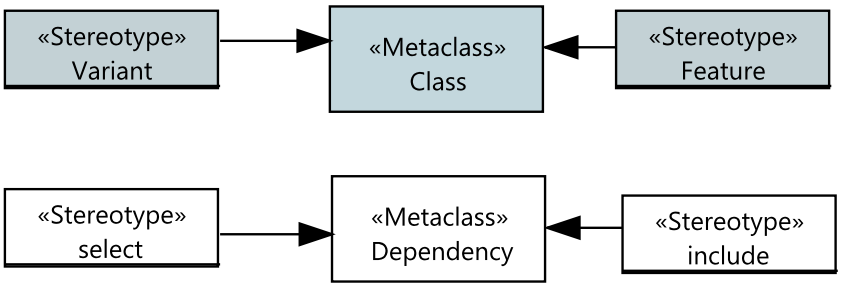


Abbildung 3.1: Überblick Profil

Featuremodell

Der erste Schritt ist die Erstellung eines Featuremodells. Hierfür wird der erste Stereotype "Feature" definiert, der an eine Class angehängt wird. Features können hierarchisch angeordnet werden (Abbildung 3.2), sprich ein übergeordnetes Feature kann Subfeatures besitzen. Dies ist sinnvoll, um Gruppierungen vorzunehmen. Falls ein Feature ein Subfeature besitzt, kann dieses nicht ausgewählt werden und wird als abstrakt markiert. In der Implementierung sind die einzelnen Features auswählbar, allerdings nur Features, die nicht abstrakt, also keine untergeordneten Features besitzen.

Definition von Varianten

Eine Variante wird dem Stereotype "Variant" deklariert und besteht aus einer Teilmenge von Features. Dies erleichtert die Erstellung von Varianten, denn dadurch ist es nicht mehr nötig bei jeder Variantenerstellung alle gewünschten Features einzeln per Hand auszuwählen, sondern sind zentral im Modell verwaltet.

Varianten und Features werden jeweils mit einer Dependency verbunden. Diese

Abbildung 3.2: Überblick Features

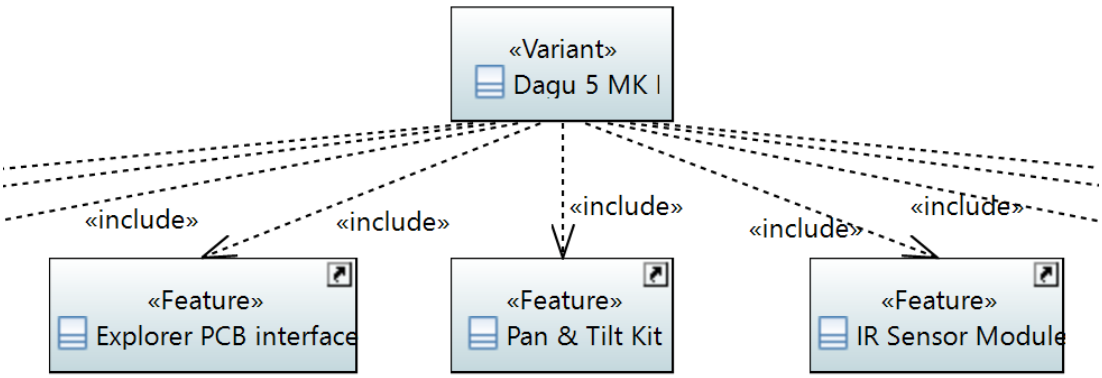
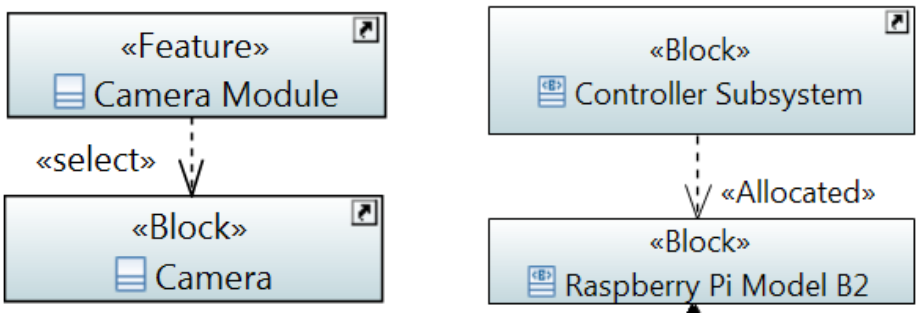


Abbildung 3.3: Überblick Variante



(a) Stereotype “select“ **(b)** Stereotype “Allocated“

Abbildung 3.4: Mapping auf das Architekturmodell

bekommt den Stereotype “include“ angehängt. Somit ist es möglich bei Auswahl einer Variante die zugehörigen Features zu finden und damit weiterzuarbeiten. Abbildung 3.3 illustriert eine beispielhafte Verwendung einer Variante.

Mapping von Features auf das Architekturmodell

Nachdem eine Auswahl an Features getroffen wurde, müssen diese Features auf das Architekturmodell abgebildet werden. Dazu werden die Stereotypes “select“ und “Allocated“ verwendet. Diese sind in Abbildung 3.4a und 3.4b zu finden.

Mit einer “select“-Beziehung kann ein Feature auf ein Element der funktionalen Architektur verweisen. Dies ist nötig, falls das Feature aus mehreren Teilen besteht und nicht durch ein einzelnes Element beschrieben werden kann. Dies ist beispielsweise der Fall, wenn ein Subsystem, welches auch ein Feature beschreibt, wiederum aus mehreren Modulen besteht, die auf verschiedene Teile des Modells verweisen. Die funktionale Architektur wird daraufhin durchsucht. Dabei kommen bei der Traversierung des funktionalen Architektur dieselben Regeln zum Einsatz, die im nächsten Abschnitt beschrieben sind.

Die eigentliche Zuteilung von Features bzw. funktioneller Architektur auf das Architekturmodell findet mittels dem Stereotype “Allocated“ statt, der in SysML definiert ist. Die hier gefundenen Elemente sind zugleich die Startpunkte für die Traversierung des Modells.

3.1.2 Traversierung des Architekturmodells

Nachdem nun die Startpunkte feststehen, müssen die Regeln für die Modelltraversierung erstellt werden. In Kapitel 2 wurden Kategorien für die einzelnen Sprachelemente gefunden, um diese zu abstrahieren. Dies ist die Grundlage, um den Algorithmus zu entwickeln.

Die Traversierung startet bei den gefundenen Startpunkten. Für jede Beziehung eines Basiselements wird dabei die getNextElement-Methode aufgerufen, um die nächsten Elemente der Traversierung zu finden. Nachdem alle Basiselemente und Beziehungen gefunden wurden, die in die Variante abgeleitet wurden, wird über alle Elemente des Modells iteriert, um Querverweise zu finden. Dabei wird für jedes Element eine handleElement-Methode aufgerufen. Dabei lehnen sich die Methoden an das Visitor-Entwurfsmuster an, da für jedes Element die entsprechenden Methoden aufgerufen werden. Die genaue Architektur des Algorithmus ist in Kapitel 4 zu finden.

Für jede Kategorie wird Pseudocode verwendet, um die Verwendung darzustellen. Dieser ist an die Programmiersprache Java angelehnt, stellt jedoch keinen Anspruch auf korrekte Java-Syntax.

Basiselement

Die SysML-Sprachelemente in der Kategorie "Basiselement" sind am einfachsten zu handhaben. Wenn die Traversierung bei einem solchen Element ankommt, wird es in die Variante mit aufgenommen. Falls das Element jedoch ein potentieller Startpunkt ist, also ein Feature repräsentiert, das nicht für die Variante ausgewählt wurde, läuft die Traversierung an dieser Stelle nicht weiter.

Listing 3.1: Pseudocode "Basiselement – handleElement-Methode"

```
[...]
// Basiselement behandeln
handleElement(Basiselement element) {
   if(element is a not selected feature) {
      // Stoppe Traversierung
      [...]
   } else {
      // Laufe weiter
      [...]
   }
}
[...]
```

Strukturelement

Strukturelemente können Basiselemente enthalten. Diese können dementsprechend erst dann aus dem Modell entfernt werden, wenn keine Elemente mehr

enthalten sind, es also leer ist.

Listing 3.2: Pseudocode "Strukturelement – handleElement-Methode"

```
HashSet variantElements;
[...]
// Strukturelement behandeln
handleElement(Strukturelement element) {
   if(element.hasNoElementsInside) {
      // Nimm Strukturelement in das Modell auf
      variantElements.add(element);
      [...]
   }
}
[...]
```

Bidirektionale Beziehung

Kanten sind die Verbindungen zwischen Modellelementen und beschreiben deren Beziehung miteinander, zum Beispiel Assoziationen oder Abhängigkeiten. Deswegen werden diese hier generell als Beziehungen bezeichnet. Diese können entweder gerichtet oder ungerichtet sein. Die ungerichteten, also bidirektionalen, Beziehungen werden nun folgend betrachtet. Die unidirektionalen Beziehungen sind Bestandteil des nächsten Abschnittes.

Bidirektionale Beziehungen können, wie der Name schon andeutet, in beide Richtungen betreten werden. Sollte also eine Beziehung dieser Kategorie vorgefunden werden, wird diese in die Variante übernommen und mit dem Element an dem gegenüberliegenden Ende fortgefahren.

Listing 3.3: Pseudocode "Bidirektionale Beziehung – getNextElement-Methode"

```
HashSet variantElements;
[...]
// Bidirektionale Beziehung behandeln
getNextElement(BidirektionaleBeziehung relationship) {
   [...]
   // Nimm Beziehung in Variante auf
   variantElements.add(relationship);
   variantElements.add(relationship.getOtherEnd());

   // Fahre mit anderem Ende fort
   handleElement(relationship.getOtherEnd());
```

```
}
[...]
```

Unidirektionale Beziehung

Wie im vorherigen Abschnitt bereits angedeutet, gibt es ebenso Kanten, die nur in eine Richtung beschritten werden können. Dementsprechend wird nur mit der Traversierung fortgefahren, wenn das Element, von dem gestartet wurde, der Anfang der gerichteten Beziehung ist. Falls das nicht der Fall ist, wird die Traversierung gestoppt.

Listing 3.4: Pseudocode "Unidirektionale Beziehung – handleElement-Methode"

```
HashSet variantElements;
[...]
// Unidirektionale Beziehung behandeln
getNextElement(UnidirektionaleBeziehung relationship) {
   if(relationship.getOtherEnd() == relationship.getDirectedEnd()) {
      // Nimm Beziehung in Variante auf
      variantElements.add(relationship);
      variantElements.add(relationship.getOtherEnd());

      // Fahre mit dem anderen Ende fort
      handleElement(relationship.getOtherEnd());
   } else {
      // Stoppe Traversierung
      [...]
   }
}
[...]
```

Annotiertes Element

Annotierte Elemente werden entweder an Knoten- oder Kantenelemente angehängt. Das sind zum Beispiel Kommentare, die genauere Details oder sonstige Informationen über dieses Element enthalten. Ein annotiertes Element wird in die Variante übernommen, wenn ein Modellelement, an das es annotiert wurde, ebenfalls in der Variante vorhanden ist.

Listing 3.5: Pseudocode "Annotiertes Element – handleElement-Methode"

```
HashSet variantElements;
[...]
// Annotiertes Element behandeln
handleElement(AnnotiertesElement element) {
   // Wenn annotiertes Element in Variante vorhanden, dann nimm es in
      die Variante auf
   for(Element annotatedElement in element.annotatedElements) {
      if(variantElements.contains(annotatedElement)) {
         variantElements.add(element);
         break;
      }
   }
}
[...]
```

Property

Ein Property ist ein Element, das innerhalb eines Basiselementes existiert. Es erweitert es also um geeignete Eigenschaften. Wenn das Basiselement in der Variante vorhanden ist, dann muss auch das Property übernommen werden.

Listing 3.6: Pseudocode "Property – handleElement-Methode"

```
HashSet variantElements;
[...]
// Property behandeln
handleElement(Property element) {
   // Wenn das Basiselement in der Variante vorhanden ist, nimm das
      Element in die Variante auf
   if(variantElements.contains(element.getBase())) {
      variantElements.add(element);
   }
}
[...]
```

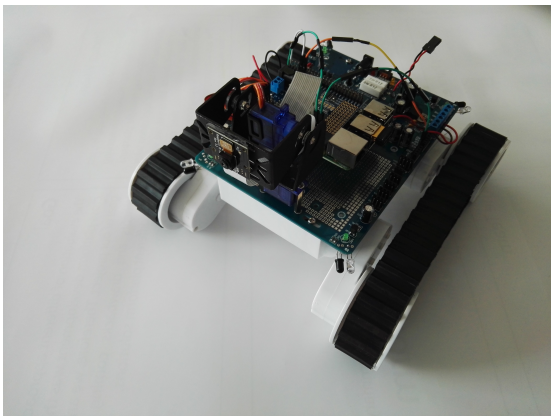


Abbildung 3.5: Dagu Rover 5

3.2 Beispielmodell Dagu Rover 5

Ein Architekturmodell, das mit dem Algorithmus bearbeitet werden soll, wird mit Metainformationen angereichert. Deswegen muss man die Struktur von diesem Modell entsprechend erweitern. Die Struktur wird nachfolgend anhand des begleitenden Beispielmodells des Dagu Rover beschrieben.

In Abbildung 3.6 ist eine Übersicht über das Dagu Rover Modell zu finden. Es unterteilt sich in insgesamt acht Abschnitte. Die Modellerweiterungen befinden sich in den Teilen “10_Product Line“ und ”30_Functional Architecture“. Der Rest ist Teil des ursprünglichen Modells. Das Modell ist mit dem auf Eclipse basierenden Tool Papyrus modelliert.

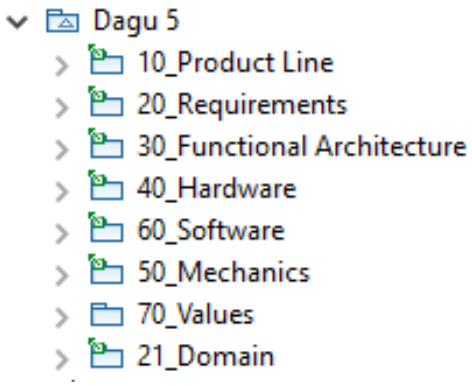
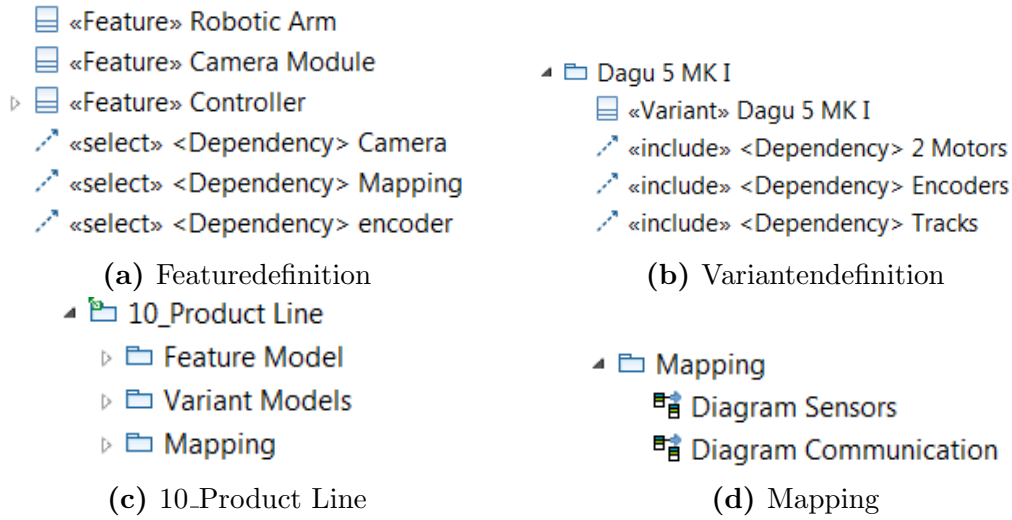


Abbildung 3.6: Modellstruktur



(a) Featuredefinition **(b)** Variantendefinition

(c) 10_Product Line **(d)** Mapping

Abbildung 3.7: Übersicht Product Line

10_Product Line

In diesem Abschnitt wird das Featuremodell nach den eingeführten Methoden definiert. Dafür ist das Package in drei weitere unterteilt (Abbildung 3.7c). Im "Feature Model" sind die Features definiert. Die select-Beziehungen sind ebenfalls hier abgelegt (Abbildung 3.7a). Die Varianten beschreibt man in "Variant Models". Jede Variante bekommt dafür ein eigenes Package, in dem der eigentlich Varianten-Block und die zugehörigen "include"-Beziehungen, die auf die genutzten Features der Variante zeigen, gespeichert sind (Abbildung 3.7b). In "Mapping" befinden sich die Diagramme für die Abbildung der Features auf die funktionale Architektur (Abbildung 3.7d). Diese bestehen aus Sprachkonstellationen wie in Abbildung 3.4a.

20_Requirements

Robuste und langlebige Architekturen benötigen ausreichend formulierte Anforderungen. Diese werden nicht nur schriftlich in einer Architekturdokumentation notiert, sondern auch in dem Architekturmodell mit verschiedenen Diagrammen visualisiert. Diese sind in "20_Requirements" zu finden. Da die Bearbeitung der Anforderungen nicht Bestandteil der Arbeit ist, wird hier nicht weiter darauf eingegangen.

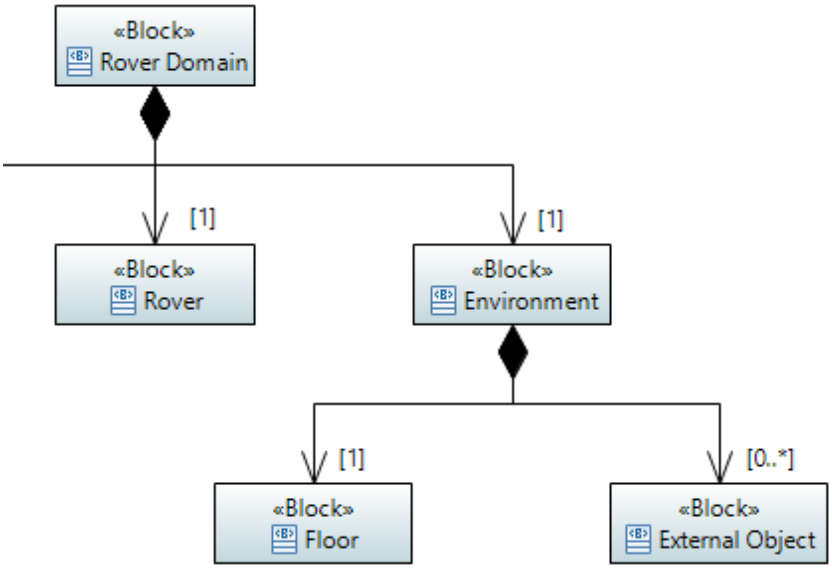


Abbildung 3.8: Beispiel Domänenmodell

21_Domain

Architekturen sind in der Regel in einer bestimmten Domäne angesiedelt, zum Beispiel in der Automobilbranche. Softwarearchitekten erstellen dabei oft ein Modell, das die Domäne widerspiegelt, für das das System entwickelt wird, ein sogenanntes Domänenmodell (DM). Dieses DM wird in diesem Abschnitt abgelegt und ist dafür zuständig die Grenzen des System inkl. Schnittstellen und der Interaktion externer Systeme mit diesen zu definieren.

30_Functional Architecture

Das Featuremodell spiegelt die oberste Abstraktionsschicht wider, welche Merkmale das Systemmodell nach außen hin zur Verfügung stellt. Features können direkt mit dem Architekturmodell verbunden werden. Es gibt allerdings Features, bei denen dies nicht möglich ist, da sie aus mehreren Elementen bestehen. In Abbildung 3.9 ist beispielsweise das Feature “Kommunikations-Subsystem“ zu sehen, das aus zwei weiteren Subsystemen, dem WLAN-Subsystem und dem Bluetooth-Subsystem, besteht. Ein Subsystem wird dabei in weitere Subsysteme mittels Komposition unterteilt (wie im Beispiel zu sehen). Diese Blöcke bilden dann wiederum Blöcke aus dem Architekturmodell ab. Solche Konstellationen werden im Abschnitt “30_Functional Architecture“ abgelegt. Zusätzlich ist in Abbildung 3.10 ein weiteres Beispiel für die Allokierung der funktionalen Architektur in das Hardware-Modell mit dem Allocated-Stereotype zu sehen.

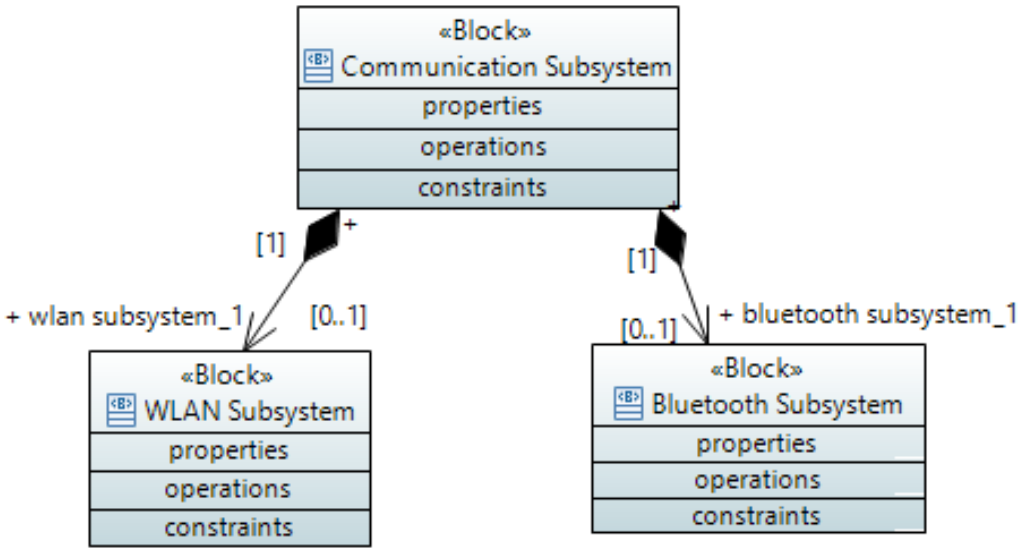


Abbildung 3.9: Beispiel Funktionelle Architektur

40_Hardware

Ein softwareintensives System besteht nicht nur aus Software, sondern auch aus Hardwarebestandteilen. Naheliegend sind natürlich Computer, auf denen die Software laufen soll. Dabei kann es sich aber auch um Mikrocontroller oder Infrastruktur wie Bussysteme handeln.

In Abbildung 3.10 ist das Beispiel des Kommunikationssystems weitergeführt. Das WLAN-Subsystem wird auf den WLAN-Dongle abgebildet. Dies ist ein Startpunkt für die Modelltraversierung. Der Dongle steht per USB in Verbindung mit einem Raspberry Pi. Diese Elemente werden in die Variante mit aufgenommen, sofern das WLAN-Subsystem in der Variante vorhanden sein soll.

50_Mechanics

Zu der Hardware kann es ebenfalls mechanische Bauteile geben. Wenn man das Beispiel des Dagu Rovers betrachtet, besteht es, unter anderem, wie in Abbildung 3.11 zu sehen, aus einem Fahrgestell, das über eine Leiterplatte mit der Batterie und einer weiteren Leiterplatte verbunden ist. Die Verbindungen können z.B. Kabel oder sonstige Leitungen sein.

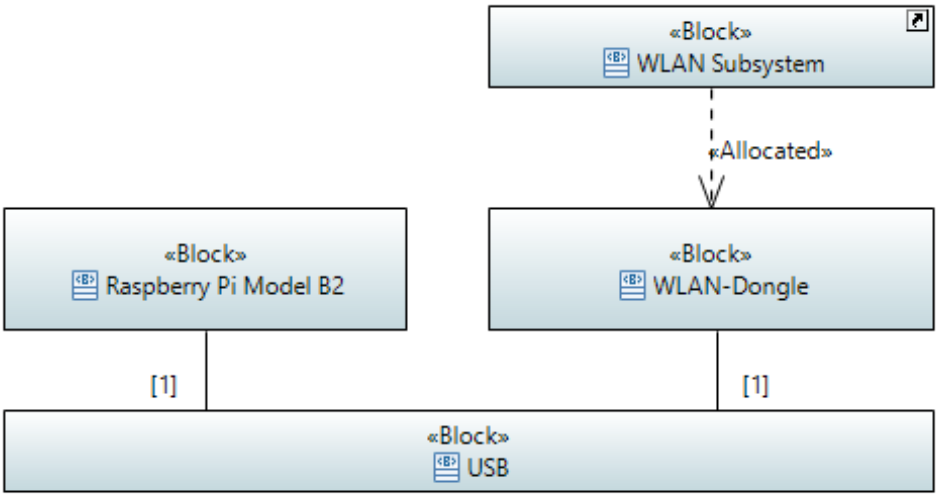


Abbildung 3.10: Beispiel Hardware

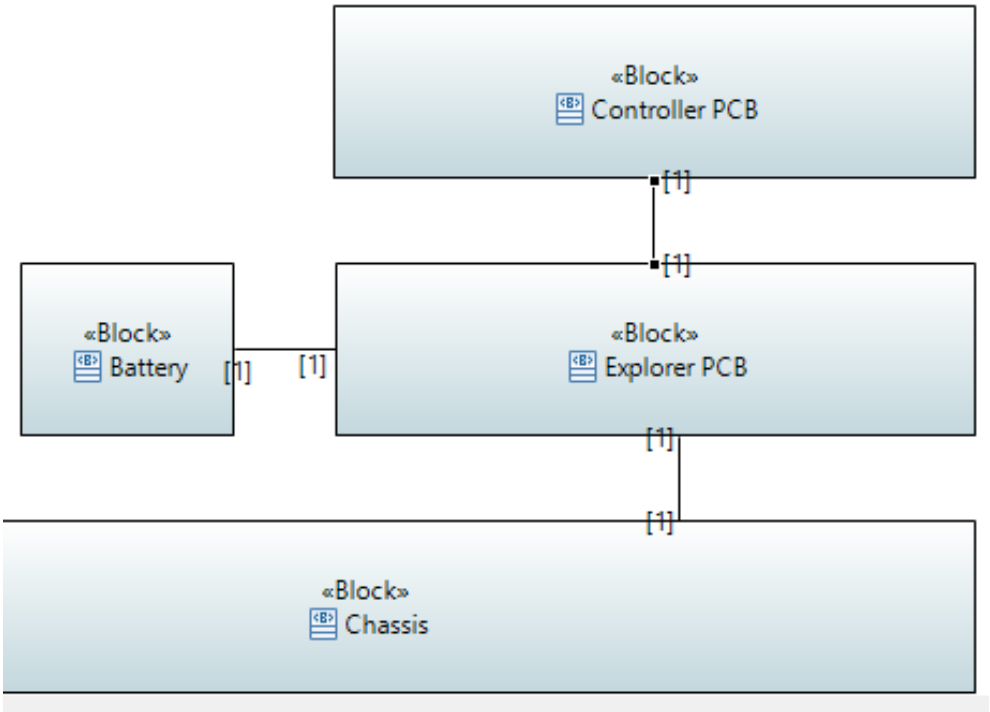


Abbildung 3.11: Beispiel Mechanics

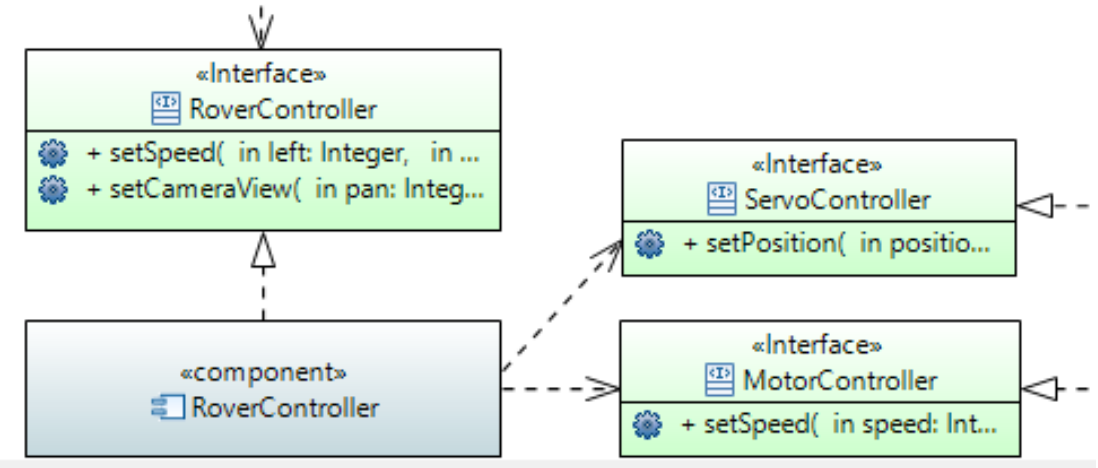


Abbildung 3.12: Beispiel Software

60_Software

Ein elementarer Teil eines System ist natürlich die Software, welche auf der Hardware ausgeführt werden soll, um seinen Zweck zu erfüllen, für den es entwickelt wird. Ein Beispiel dafür ist in Abbildung 3.12 zu sehen.

70_Values

In manchen Systemen werden bestimmte Werte oder Einheiten verwendet. Diese sind in der SysML-Spezifikation genauer beschrieben.

4 Architektur und Design des Algorithmus

Dieses Kapitel beschäftigt sich mit dem Design, also dem Aufbau des Algorithmus. Dieser ist unterteilt in insgesamt 7 Schritte, welche nachfolgend aufgeführt sind.

1. Auswählen einer Variante
2. Lesen des Modells
3. Finden aller Startpunkte für die Traversierung
4. Traversierung des Plattformmodells
5. Finden von Querverweisen
6. Löschen der nicht erreichten Elemente
7. Speichern der Variante

Im weiteren Verlauf des Kapitels werden die einzelnen Schritte genauer beschrieben.

Auswählen einer Variante

Bevor eine Variante abgeleitet werden kann, müssen vom Benutzer alle gewünschten Features ausgewählt werden, die in der Variante vorhanden sein sollen. Dies geschieht entweder durch manuelles Hinzufügen von Features oder über vorgefertigte Variantendefinitionen, wie sie in Kapitel 3.1.1 beschrieben sind. Zusätzlich muss der Nutzer dem neuen Modell einen Namen geben.

Lesen des Plattformmodells

Bevor etwas an dem PM bearbeitet wird, muss es zuerst eingelesen werden. Es wird nur auf dieser separaten Kopie gearbeitet, die vorher erstellt wird. Dies hat

den Vorteil, dass das originale Modell nicht verändert wird.

Finden aller Startpunkte für die Traversierung

Nun müssen die Startpunkte im Modell für die Traversierung gefunden werden. Wie in Kapitel 3.1.1 gezeigt, kann es notwendig sein einen Umweg über die funktionale Architektur zu gehen. Wenn Features direkt in das Modell verweisen werden, dann werden die Verweise in die Queue mit Startpunkten geschrieben. Falls ein Verweis auf die funktionale Architektur besteht, muss diese für das Element durchsucht werden. Alle dort gefunden Verweise werden ebenfalls zu den Startpunkten hinzugefügt. Eine Übersicht der Vorgehensweise ist in Abbildung 4.1 zu sehen.

Traversierung des Plattformmodells

Nachdem alle Startpunkte gefunden wurden, steht nun die Traversierung des PM auf dem Plan. In Abbildung 4.2 ist diese als Aktivitätsdiagramm dargestellt. Die Traversierung wird erst beendet, wenn es keine Elemente mehr zu untersuchen gibt. Wenn es diese gibt, werden sie aus der Queue rausgenommen. Wenn das Element zu den bereits untersuchten Elementen gehört, also schon in der Variante vorhanden ist, dann wird mit dem nächsten Element in der Queue fortgefahren. Andernfalls wird das Element zu der Variante hinzugefügt (es handelt sich dabei um Basiselemente). Wenn das Element nun noch weitere Beziehungen hat, werden diese untersucht. Auch hier stellt sich wieder die Frage, ob die Beziehung bereits zu der Variante gehört. Falls ja, läuft die Traversierung an dieser Stelle nicht weiter. Falls nicht, dann wird die Beziehung zu der Variante hinzugefügt. Zum Schluss wird das andere Beziehungsende der Queue hinzugefügt, damit die Traversierung dort weiterlaufen kann.

Finden von Querverweisen

Da bei der Traversierung nur die Basiselemente gefunden werden, müssen nun die Querverweise ermittelt werden, also Properties, Ports und annotierte Elemente, die zu einem dieser Basiselemente gehören. Dafür wird über alle Elemente des Modells iteriert und bei jedem Element geprüft, ob es eine Referenz zu einem Basiselement besitzt. Ist dies der Fall und dieses Basiselement in der Variante enthalten, so wird es ebenfalls in die Variante aufgenommen.

Der Schritt ist notwendig aufgrund der Implementierung des Eclipse Modeling Framework. Man kann ein annotiertes Element, z.B. einen Comment, von einem Basiselement nicht direkt erreichen. Lediglich das annotierte Element hat

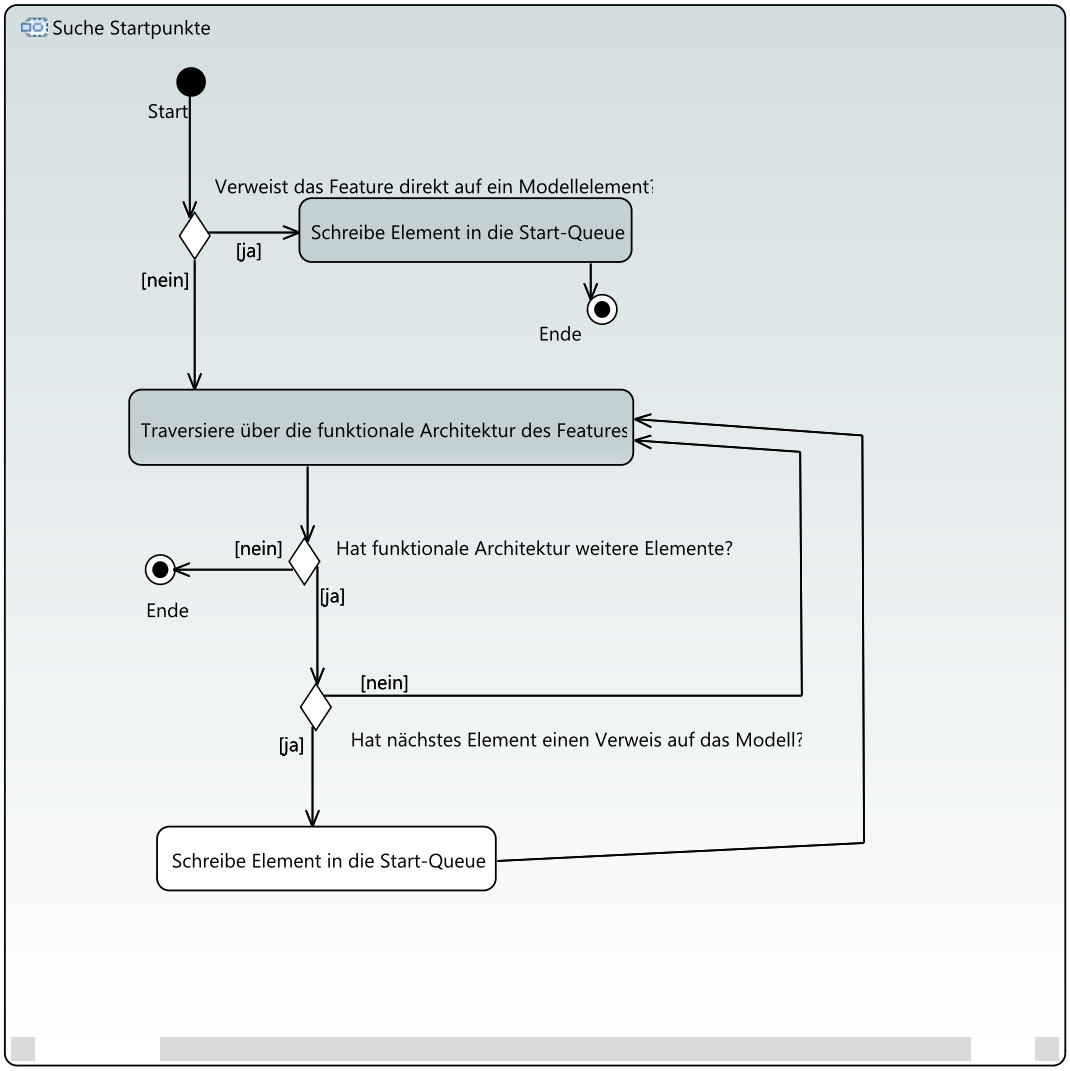


Abbildung 4.1: Suche Startpunkte

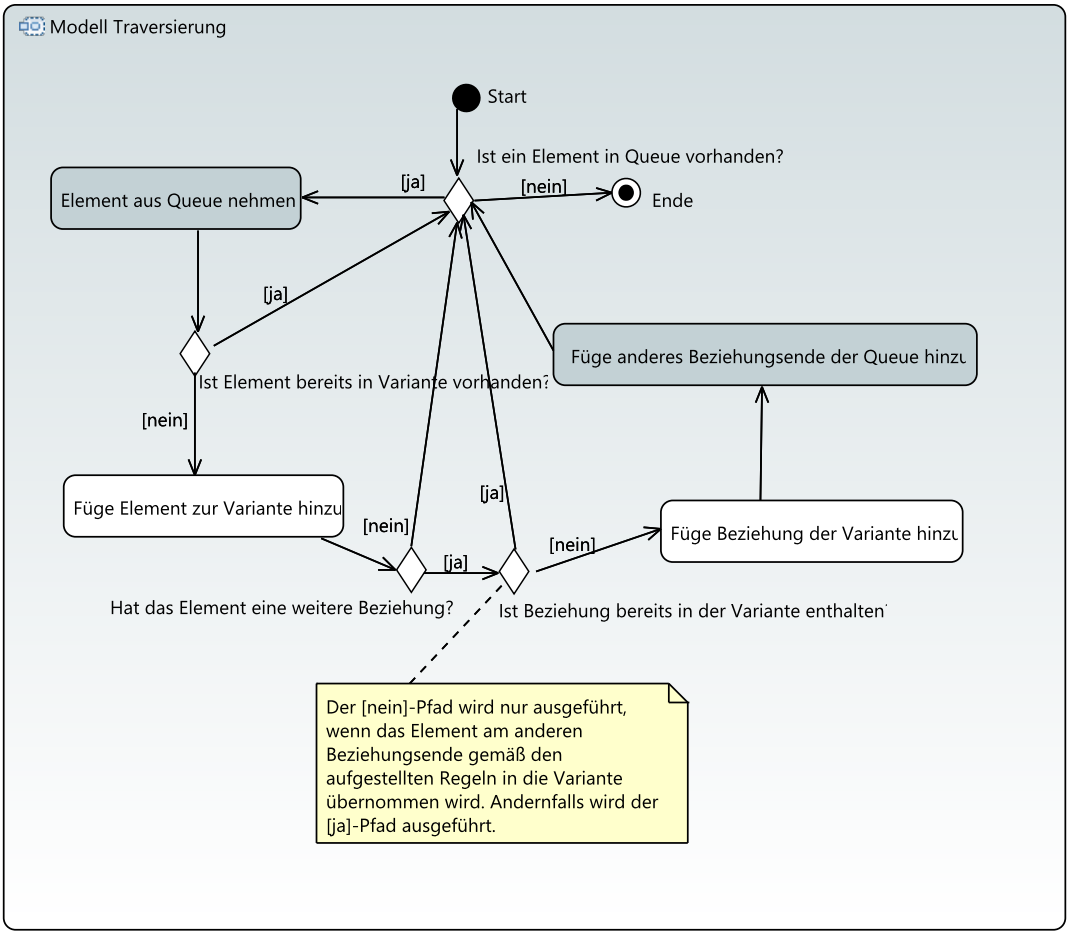


Abbildung 4.2: Traversierung des Modells

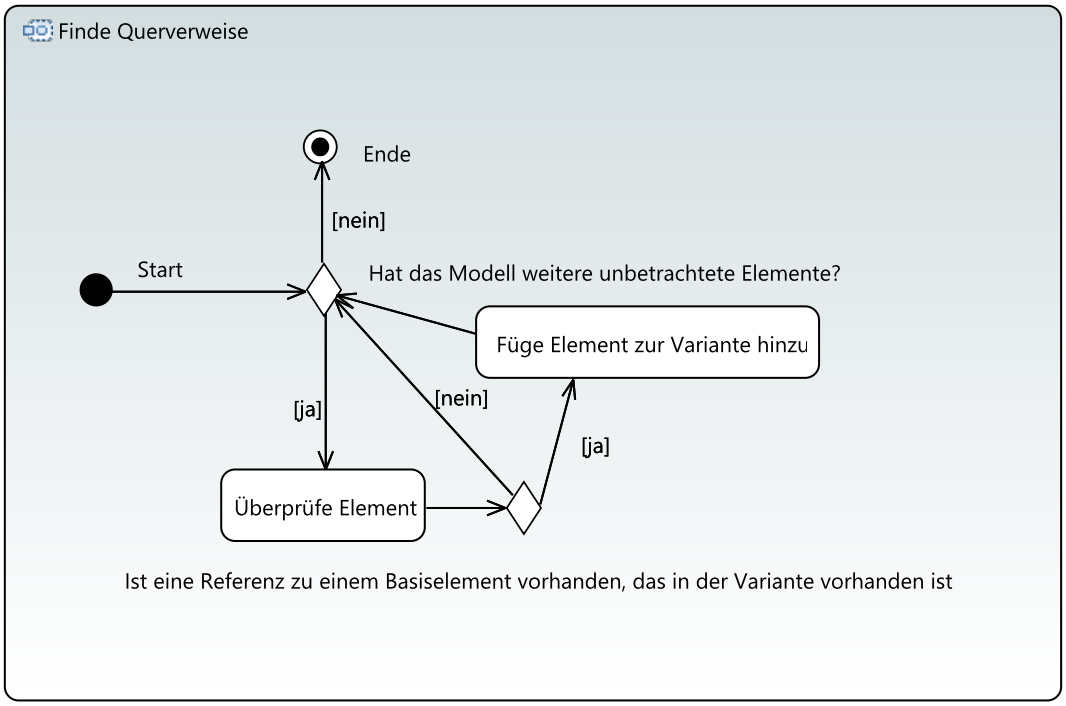


Abbildung 4.3: Querverweise finden

die Informationen, zu welchem Element es gehört. Deshalb werden diese bei der Traversierung nicht gefunden. Zusätzlich werden auch Ports und Properties erst in diesem Schritt markiert. Dies wäre nicht zwingend notwendig, wird allerdings erst an dieser Stelle gemacht, um die Traversierung zu vereinfachen.

Löschen der nicht erreichten Elemente

Nach der Traversierung und dem Finden der Querverweise sind alle Elemente markiert, die in der Variante vorhanden sein sollen. Diese werden in diesem Schritt endgültig aus dem Modell gelöscht.

Speichern der Variante

Die Variante ist jetzt komplett erstellt, liegt allerdings noch im Arbeitsspeicher. Das heißt, dass das Variantenmodell noch abgespeichert werden muss. Dies wird im letzten Schritt gemacht.

5 Implementierung

Nachdem im vorherigen Kapitel die Architektur des Algorithmus besprochen wurde, ist es nun an der Zeit einen Prototypen des Algorithmus zu implementieren. Doch bevor mit der eigentlichen Umsetzung begonnen wird, sind zunächst einige grundlegende Anforderungen festzuhalten, an die sich die Implementierung halten muss.

5.1 Anforderungen an die Implementierung

Anforderungen sind wichtig, um einen Rahmen dafür zu bilden, was die Software leisten muss. Nachfolgend sind die Anforderungen aufgeführt, die für die Implementierung des Algorithmus angewendet werden.

Modellierungssprache SysML

Diese Arbeit behandelt die Ableitung von Varianten in SysML-basierten Systemarchitekturen. Offensichtlich ist also die Modellierungssprache, die bei den Architekturmodellen verwendet wird, die Systems Mongdeling Language. Der Vollständigkeit halber sei dies hier auch aufgeführt.

Struktur des Architekturmodells

Die Architekturmodelle sind mit dem auf Eclipse basierenden Papyrus erstellt. Die Implementierung muss also mit diesen Papyrus-Modellen umgehen können. Zudem hat das Modell die in Kapitel 3 beschriebene Struktur.

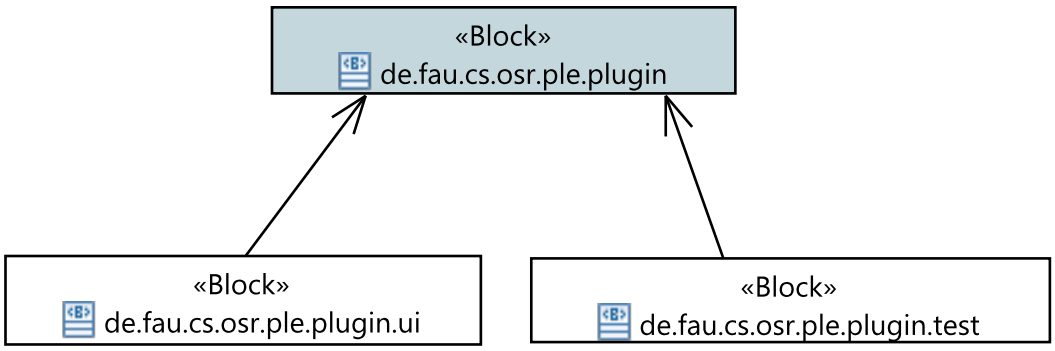


Abbildung 5.1: Struktur Plugin

Plugin für Eclipse

Da das zu verwendende Werkzeug Papyrus auf der Entwicklungsumgebung Eclipse basiert, liegt es nahe ein Plugin für Eclipse zu schreiben, in dem der Algorithmus umgesetzt wird.

Funktionalität und Zuverlässigkeit

Um die Korrektheit der Implementierung zu gewährleisten, wird eine Test-Suite bestehend aus mehreren Unit-Tests erstellt. Dies soll zum einen die Funktionalität der Software überprüfen, ob die benötigten Funktionen überhaupt umgesetzt sind, zum anderen die Zuverlässigkeit der umgesetzten Funktionen, ob diese korrekt funktionieren.

5.2 Umsetzung der Implementierung

Die Implementierung wird als Eclipse-Plugin umgesetzt, da das zu bearbeitende Modell mittels Papyrus erstellt wurde, das ebenfalls auf Eclipse basiert. Als Programmiersprache wird dafür zum einen natives Java, zum anderen Xtend (http://www.eclipse.org/xtend/) verwendet. Xtend ist ein Java-Dialekt, der die Sprache u.a. um Aspekte der funktionalen Programmierung erweitert. Um die Aufgaben zu verteilen, wurden insgesamt drei Plugins erstellt. Dies ist in 5.1 zu sehen.

5.2.1 Struktur der Plugins

Der eigentliche Algorithmus ist in dem Plugin “de.fau.cs.osr.ple.plugin“ umgesetzt. Hier liegen sämtliche Funktionalitäten, die benötigt werden, um eine Variante aus einem Plattformmodell abzuleiten.

Zur Überprüfung der Korrektheit der Funktionalität wurden Unit-Tests geschrieben. Diese sind in einem eigenen Plugin “de.fau.cs.osr.ple.plugin.test“ abgelegt.

Um eine Variante erstellen zu können, ist ebenfalls eine Benutzerinteraktion erforderlich. Dafür muss die Oberfläche von Eclipse erweitert werden. Diese befinden sich im Plugin “de.fau.cs.osr.ple.plugin.ui“. Dies trennt die Zuständigkeiten klar zwischen Oberfläche und der eigentlichen Logik. Zudem beschleunigt das die Ausführung der Unit-Tests, da keine unnötigen Abhängigkeiten geladen werden müssen.

5.2.2 Struktur der Algorithmusimplementierung

Das Plugin “de.fau.cs.osr.ple.plugin“, das den eigentlichen Algorithmus enthält, ist in drei wesentliche Packages unterteilt. Eine Übersicht über die Struktur ist in Abbildung 5.2 zu finden.

Das Package “de.fau.cs.osr.ple.plugin.meta“ enthält die Klasse ModelMetaInformation. Diese verwaltet alle Metainformationen, die dazu benötigt werden, um eine Variante zu erstellen. Dazu gehören unter anderem die Namen der Stereotypes zur Erstellung der Featuremodelle, die erreichten Elemente, die während der Graphtraversierung in die Variante übernommen werden, die ausgewählten Features, die Startpunkte und sämtliche Informationen, die das eigentliche Modell betrifft. Eine vollständige Übersicht der Schnittstelle ist im Listing 5.1 zu finden. Für den korrekten Start des Algorithmus sind folgende Metainformation notwendig: SelectedFeatures, InputModelPath, OutputModelName, Model.

Listing 5.1: Schnittstelle “ModelMetaInformation“

```
public interface ModelMetaInformation {

   /*****************************************************************
   * Constants
   *****************************************************************/
   String STEREOTYPE_FEATURE = "PLE-Profile::Feature";
   String STEREOTYPE_INCLUDE = "PLE-Profile::include";
   String STEREOTYPE_SELECT = "PLE-Profile::select";
   String STEREOTYPE_VARIANT = "PLE-Profile::Variant";
   String STEREOTYPE_ALLOCATED = "SysML::Allocations::Allocated";
```

```
    /*****************************************************************
    * Getter and Setter
    *****************************************************************/
    HashSet<EObject> getReachedElements();
    List<String> getSelectedFeatures();
    NamedElement getSelectedVariant();
    HashSet<NamedElement> getNotSelectedFeatures();
    HashSet<NamedElement> getStartPoints();

    String getInputModelName();
    void setInputModelName(String inputModelName);

    String getInputModelPath();
    void setInputModelPath(String inputModelPath);

    String getOutputModelName();
    void setOutputModelName(String outputModelName);

    String getOutputModelPath();
    void setOutputModelPath(String outputModelPath);

    void setModel(Model model);
    Model getModel();

    /*****************************************************************
    * Methods
    *****************************************************************/
    boolean containsFeature(String featureName);
    boolean isVariantSelected(NamedElement variant);
    void toggleFeature(String featureName);
    void toggleVariant(NamedElement selectedVariant);
}
```

Im Package "de.fau.cs.osr.ple.io" sind zwei Hilfsklassen vorhanden, die sich um das Lesen bzw. Schreiben der Modelle kümmern, der PleReader und der PleWriter. Diese sind in den Listings 5.2 und 5.3 dargestellt. Um die Verwendung einfach zu halten, gibt es jeweils nur eine öffentliche Methode.

Listing 5.2: Schnittstelle "PleReader"

```
public interface PleReader {

    public UMLResource readUMLResource(String path, String name);
```

```
}
```

Listing 5.3: Schnittstelle “PleWriter“

```
public interface PleWriter {

   public void writeModels();
}
```

Der Hauptteil des Algorithmus ist im dritten Package “de.fau.cs.osr.ple.algorithm“ implementiert.

Die Schnittstelle PleManager dient, wie der Name schon andeutet, als Manager-Objekt und ist gleichzeitig der Einstiegspunkt für Benutzer, um den Algorithmus zu verwenden (in diesem Fall das UI-Plugin). Sie verteilt die Anfragen dann an die entsprechenden Stellen. Es beinhaltet zudem die Namen der einzelnen Teile der Modellstruktur, wie sie in 3.2 beschrieben sind.

Listing 5.4: Schnittstelle “PleManager“

```
public interface PleManager {

   /**
   * Constants which are used during variant extraction.
   */
   static final String ALGORITHM_PRODUCT_LINE = "10_Product Line";
   static final String ALGORITHM_REQUIREMENTS = "20_Requirements";
   static final String ALGORITHM_DOMAIN = "21_Domain";
   static final String ALGORITHM_FUNCTIONAL_ARCHITECTURE =
       "30_Functional Architecture";
   static final String ALGORITHM_HARDWARE = "40_Hardware";
   static final String ALGORITHM_MECHANICS = "50_Mechanics";
   static final String ALGORITHM_SOFTWARE = "60_Software";
   static final String ALGORITHM_VALUES = "70_Values";
   static final String ALGORITHM_MAPPING = "Mapping";

   /**
   * Extracts the model of a given selected object in the
       ModelExplorer and return the feature line package
   */
   Object prepareModel(Object selected);

   /**
```

```
    * Returns the ModelMetaInformation object.
    */
    ModelMetaInformation getModelMetaInformation();

    /**
    * Indicates whether or not the algorithm is executable.
    */
    boolean isAlgorithmExecutable();

    /**
    * Starts the PLE algorithm.
    */
    void startPleAlgorithm();

    /**
    * Handles the selection of a feature in the GUI.
    */
    void handleFeatureSelection(Object selectedFeature);

    /**
    * Indicates if the given variant is already selected.
    */
    boolean isVariantSelected(Object possibleVariant);

    /**
    * Indicates if the given feature is already selected.
    */
    boolean isFeatureSelected(Object possibleFeature);

    /**
    * Sets the project which includes the model
    */
    void setProject(IProject project);
}
```

Die Schnittstelle PleAlgorithmWorker ist für die Ausführung des Algorithmus zuständig. Dieser funktioniert wie in Kapitel 4 beschrieben. Die Schnittstelle beinhaltet lediglich eine Methode, um den Algorithmus zu starten. Alle weiteren Schritte geschehen automatisch durch die Klasse mit Hilfe der Metainformationen, die dem Algorithmus zur Verfügung stehen.

Listing 5.5: Schnittstelle “PleAlgorithmWorker“

```
public interface PleAlgorithmWorker {
```

```
    /**
    * Starts the execution of the algorithm.
    */
    void startAlgorithm();
}
```

Die Schnittstelle PleTreeWalker führt die Traversierung des Modells aus und sucht sich die Modellelemente nach den definierten Regeln, die sie dann für die Mitnahme in die Variante entsprechend markiert.

Listing 5.6: Schnittstelle "PleTreeWalker"

```
public interface PleTreeWalker {

    /**
    * Starts the traversing.
    */
    void walkTree();
}
```

Behandlung nicht relevanter Modellteile

Das Modell beinhaltet auch Requirements, die nicht Umfang der Arbeit sind. Diese werden im Prototyp bei der Variantenerstellung gelöscht. Ebenfalls werden die Features und die funktionale Architektur nicht mit in die Variante übernommen.

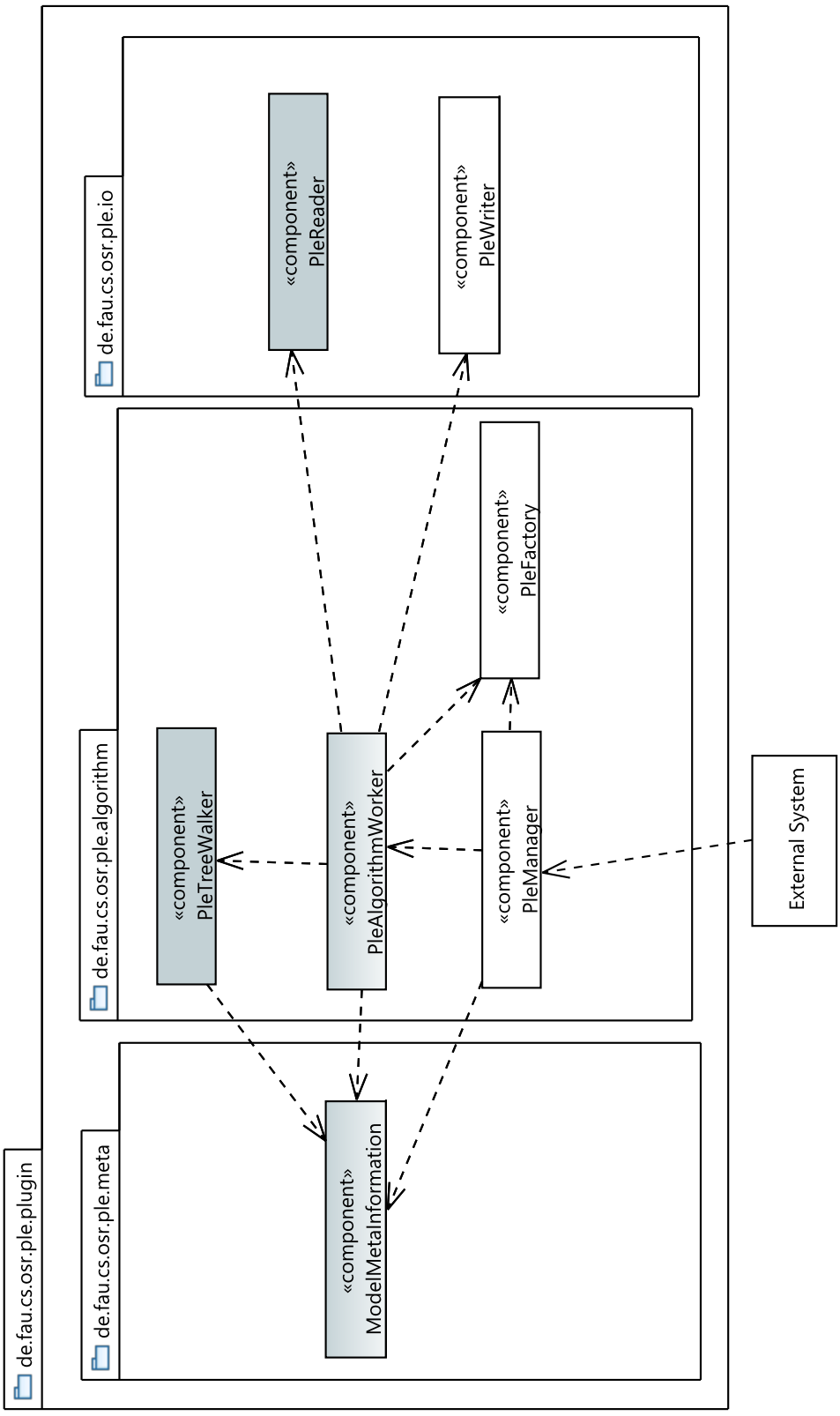


Abbildung 5.2: Komponentendiagramm des Algorithmus

6 Evaluation

Für die Evaluation wird zuerst das Beispielmodell mit dem Prototypen getestet. Dabei wird konkret auf ein Feature eingegangen. Anschließend werden die Ergebnisse der Arbeit diskutiert.

6.1 Testen des Prototyp am Beispielmodell

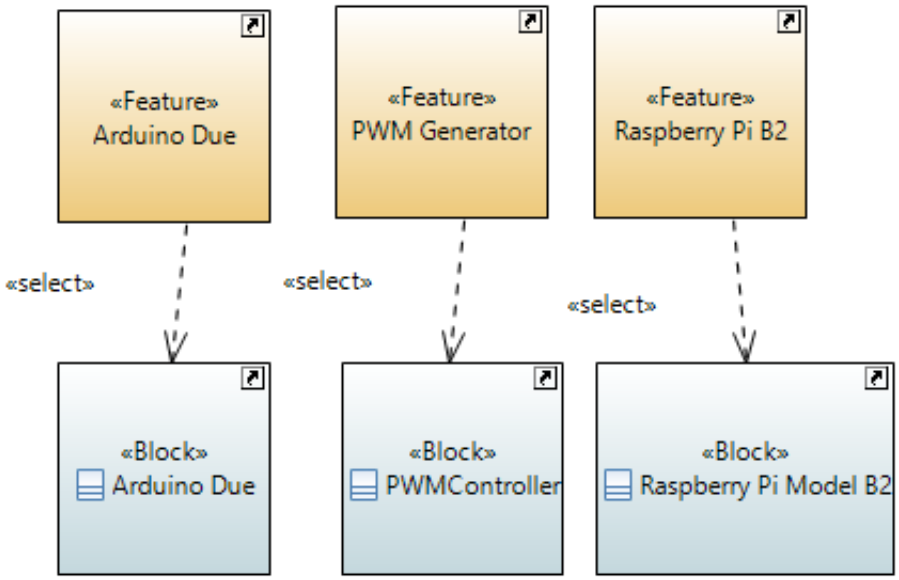


Abbildung 6.1: Mapping der Controller

Um die Funktionsweise neben den Unit-Tests auch anschaulich darzustellen, wird an dieser Stelle die Ausführung der Implementierung anhand eines konkreten Punktes des Dagu Rover Modells durchexerziert. Dafür wird das Beispiel der Controller genommen, die auf einen Roboter gebaut werden können. Im konkreten Fall handelt es sich dabei entweder um einen Raspberry Pi (https://

www.raspberrypi.org/) oder um einen Arduino Due (https://www.arduino.cc/en/Main/ArduinoBoardDue). Es ist jeweils eine Variante für beide Möglichkeiten vorhanden. In Abbildung 6.1 werden die Features "Adruino Due" und "Raspberry Pi B2" auf die jeweiligen Elemente im Modell verwiesen. Jeder Controller-Typ hat innerhalb des Hardware-Bereiches ein eigenes Diagramm bekommen. Diese dienen als Vergleichsbasis. Es werden insgesamt zwei Durchläufe gestartet, einen mit der Raspberry Pi-Variante, einen mit der Arduino-Variante. Die Ergebnisdiagramme können im Anhang B betrachtet werden.

Abbildungen 7.53 und 7.54 zeigen die jeweiligen Diagramme der Raspberry Pi-Variante. Das Diagramm, das den Raspberry Pi-Controller zeigt, beinhaltet noch alle Modellelemente. Die nicht gefundenen Elemente sind die ursprünglichen Metainformationen, die in der Implementierung ebenfalls gelöscht werden. Man kann sie allerdings noch in dem Diagramm sehen, da die Diagramme nicht bearbeitet werden, sondern lediglich die dahinter liegenden Elemente des Modells gelöscht wurden. Das Arduino-Diagramm hingegen beinhaltet fast nur gelöschte Elemente. Die noch vorhandenen Elemente wurden nicht gelöscht, da sie vom Raspberry Pi-Controller benötigt werden.

Abbildungen 7.55 und 7.56 zeigen das gespiegelte Bild der Arduino-Variante. Hier beinhaltet das Raspberry Pi-Diagramm fast nur gelöschte Elemente und das Arduino-Diagramm ist noch vorhanden.

Dies zeigt, dass die vorgestellten Prinzipien funktionieren und somit das Ziel dieser Arbeit erfüllt ist. Jedoch sind verschiedene Aspekte noch zu diskutieren. Dies findet im nächsten Abschnitt statt.

6.2 Diskussion

In diesem Abschnitt sollen einige Aspekte der Arbeit aufgegriffen und diskutiert werden. Der eigene Ansatz war die Ableitung von Varianten durch Annotierung eines Architekturmodells mit Metainformationen. Folgende Schritte in dem Prozess werden dabei diskutiert:

- **Analyse von SysML:** Die Analyse fand mit der neuesten Spezifikation 1.4 von SysML statt, die erst im Jahr 2015 erschien. Dies hat zur Folge, dass einige Sprachelemente, die neu in der Version sind, noch nicht in dem verwendeten Tool Papyrus umgesetzt sind. Man hätte die Analyse auch mit Version 1.3 durchführen können, um den aktuellen Stand von Papyrus eher zu entsprechen. Es enthält dann aber Elemente, die nach Version 1.4 veraltet sind und früher oder später auch aus Papyrus verschwinden. Die theoretischen Überlegungen bzw. die Kategorisierungen der Elemente ist davon unabhängig, jedoch gab es dadurch zunächst Probleme bei der

Implementierung, da sie allerdings auch in die verschiedenen Kategorien unterbringen lassen, sind diese auch abstrahiert und werden korrekt behandelt.

- **Kategorisierung der Sprachelemente:** Es wurden insgesamt sieben Kategorien erstellt, wobei nur sechs davon effektiv verwendet werden, da die Diagramme außerhalb des Umfangs der Arbeit liegen. Die Kategorisierung macht Sinn, da viele Elemente gleiche Eigenschaften haben, somit muss nicht jedes Element einzeln betrachtet werden, sondern nur zugehörigen Kategorien. Die Kategorie "Strukturelement" beinhaltet lediglich zwei Elemente. Es wäre also denkbar diese Kategorie ebenfalls zu eliminieren und den Basiselementen hinzuzufügen. Die Inhalte der Strukturelemente werden dann als Properties betrachtet.
- **Identifikation von Sprachkonstrukten:** Die Annotierung findet in zwei Bereichen statt. Zum einen muss das FM abstrahiert werden, zum anderen muss ggf. eine geeignete funktionale Architektur erstellt werden, falls ein Feature sich nicht direkt auf das Modell abbilden lässt. Dazu werden eigens erstellte Stereotypes verwendet, die die gewünschte Semantik abbilden. Dazu ist es nötig immer ein zusätzliches Profil einzubinden. Es wäre möglich Elemente von SysML zu verwenden, um die Stereotypes zu ersetzen. Jedoch verwendet man diese Elemente dann falsch und nicht der Semantik entsprechend. Der Aufbau des FM ist eine Eigenentwicklung, es ist aber auch sinnvoll evtl. einen bereits etablierten Standard zu verwenden. Zudem könnte man Features auch direkt an mehrere Elemente im Modell verweisen, um damit der funktionalen Architektur zu entgehen. Dadurch würde allerdings schnell die Übericht verloren gehen und die Änderbarkeit würde sich schwierig gestalten.
- **Identifikation von Regeln:** Die Regeln zur Traversierung werden auf die Kategorien angewendet, nicht auf einzelne Sprachelemente. Dabei gibt es zwei unterschiedliche Methoden. Dies könnte zunächst verwirrend erscheinen, da es nicht einheitlich aussieht. Es handelt sich aber um zwei verschiedene Schritte. Die getNextElement-Methoden kommen bei der eigentlichen Modelltraversierung zum Einsatz. Es werden nur die Beziehungen und dabei auch die Basiselemente betrachtet. Die handleElement-Methoden gehören zum Finden von Querverweisen. Dabei wird über jedes Element des Modell iteriert und festgestellt, ob es zur Variante gehört. Die Aufteilung der Schritte ist nötig, da bei der Traversierung nicht jedes Element erreicht werden kann, z.B. die annotierten Elemente.
- **Umsetzung des Algorithmus:** Der Algorithmus sieht vor, dass Elemente nicht sofort gelöscht werden, sondern sie werden zuerst markiert und dann in einem nächsten Schritt gelöscht. Dies ist notwendig, da die Modelle Zyklen enthalten können oder andere Startpunkte, die gewisse Abhängigkeiten

benötigen. Deswegen kann man bei Erreichen eines Elementes nicht sofort feststellen, ob dieses von einem anderen Element benötigt wird. Daher werden die Elemente erst nach Vollenden der Traversierung gelöscht. Auch der Schritt des "Finden von Querverweisen" ist notwendig. Es gibt Elemente, die von einem Basiselement nicht erreicht werden können, obwohl dies laut Standard eigentlich möglich sein sollte, z.B. annotierte Elemente. Dort setzt das Tool Papyrus den Standard nicht korrekt um, weswegen nach der Traversierung noch Abhängigkeiten gefunden werden müssen.

- **Implementierung des Prototyps:** Die Implementierung wurde in drei Plugins realisiert, um die Zuständigkeiten zu trennen. Dies ist sinnvoll, jedoch nicht notwendig. Es wurde darauf geachtet, dass die Schnittstellen möglich simpel und einfach zu verwenden sind. Es gibt z.B. nur eine Methode um den Algorithmus zu starten, alles andere übernimmt dann die Anwendung, sofern die nötigen Metainformationen vorhanden sind. Zudem ist die Löschung der Elemente aus dem Modell sehr ineffizient. Es wird der Löschmechanismus des EMF verwendet, welcher das Modell bei jedem Löschvorgang mehrmals durchläuft, um auch alle Querverweise zu löschen. Hier könnte eine performantere Lösung gesucht werden.

7 Zusammenfassung

Da die Produktlinienentwicklung immer wichtiger wird, hat sich diese Arbeit der Problematik der automatisierten Ableitung von Varianten aus Systemarchitekturen angenommen. Dafür wurde aus einem Plattformmodell eine Variante abgeleitet auf Basis eines Featuremodells, das die Variabilität des PM abstrahiert. Mit Hilfe von Metainformationen war es dann möglich werkzeuggestützt automatisiert Varianten aus dem PM zu erstellen. Dabei sollten möglichst wenig Metainformationen zum Einsatz kommen. Es wurde eine konkrete Möglichkeit gezeigt, wie dieses entwickelte FM auf Basis des Tools Papyrus implementiert und die einzelnen Features auf das Architekturmodell allokiert werden können.

Um das Ziel zu erreichen wurden zunächst die Sprachelemente von SysML untersucht und einer Kategorie zugewiesen, um die einzelnen Elemente auf eine höhere Ebene zu abstrahieren. Dies vereinfachte die Algorithmusentwicklung, da nicht mehr jedes Element einzeln betrachtet werden musste. Danach wurden Sprachkonstrukte für die Annotierung mit Metainformationen gefunden. Dadurch war es möglich ein FM zu entwickeln inkl. einer funktionalen Architektur. Im nächsten Schritt wurden Regeln aufgestellt, mit deren Hilfe diese Konstrukte in das Modell abgebildet werden konnten. Die Zugehörigkeit der übrigen Modellelemente zu einer Variante wurde durch die Traversierung des Modells und das Finden von Querverweisen anschließend erreicht. Die gefundenen Regeln sind in einem Algorithmus umgesetzt worden. Dafür wurden die einzelnen Schritte aufgezeigt, die nötig waren. Die Implementierung wurde in Papyrus als Plugin realisiert. Dabei wurde die Struktur der Plugins und der Algorithmusimplementierung beschrieben sowie die Schnittstellen der einzelnen Klassen vorgestellt.

Zum Schluss wurde die Implementierung des Prototyps und damit die Funktionsweise des Algorithmus evaluiert anhand des Dagu Rover-Modells, welches ebenfalls während der Arbeit vorgestellt wurde. Es wurde das Feature der Controller untersucht und anhand von zwei verschiedenen Varianten getestet. Anschließend wurden noch einige Aspekte der Arbeit diskutiert.

7.1 Ausblick

Für die Zukunft gibt es auch einige Ansatzpunkte der Arbeit, an denen angesetzt werden kann. Die nächsten Versionen von SysML können dabei relativ leicht hinzugefügt werden, indem neue Sprachelemente kategorisiert bzw. veraltete Elemente eliminiert werden.

Es ist möglich den Algorithmus in verschiedene PLE-Prozesse einzubauen, z.B. in den in der Einleitung vorgestellten FORM-Prozess. Dieser zeigt Analogien zu der funktionalen Architektur, die im eigenen Ansatz verwendet wird. Auch in den KobrA-Prozess kann man den Algorithmus einbauen. Ein ähnliches Vorgehen findet nämlich im eignen Ansatz statt, um die Features über die funktionale Architektur an das Architekturmodell zu binden. Das eigentliche Modell in KobrA ist an den C2-Architekturstil angelehnt, welcher auf Komponenten und Nachrichten beruht. Der eigene Ansatz unterstützt sämtliche Architekturstile, die sich mit SysML modellieren lassen. Mit Anpassungen ist es sicherlich auch möglich den Algorithmus an anderen Stellen einzusetzen.

Es wäre auch denkbar andere Standards einfließen zu lassen. Beispielsweise kann man einen Standard für das FM verwenden (das FM der Arbeit ist eine Eigenentwicklung). Das EMF Feature Model hat dies versucht, diesen könnte man aufgreifen oder man unterstützt von vornherein mehrere Datenstrukturen.

Außerhalb des Umfangs der Arbeit lagen die Diagramme von SysML. Diese liegen unbearbeitet weiterhin innerhalb des Papyrus-Modells. Diese müssen in Zukunft noch bearbeitet werden, um den Algorithmus konkret einsetzen zu können.

Ein weiterer Ansatz ist das Ausweiten des Algorithmus auf UML. Dazu wäre es nötig alle Elemente von UML zu untersuchen und zu kategorisieren.

Zusätzlich könnte man den Algorithmus auch für andere Tools implementieren.

Anhang A Weitere Sprachelemente

A.1 Blöcke

ValueType

1. **Beschreibung:** Ein ValueType definiert Werte und zugehörige Operationen, die während der Modellbeschreibung verwendet werden können. Es basiert auf dem UML-Element "data type".
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.1

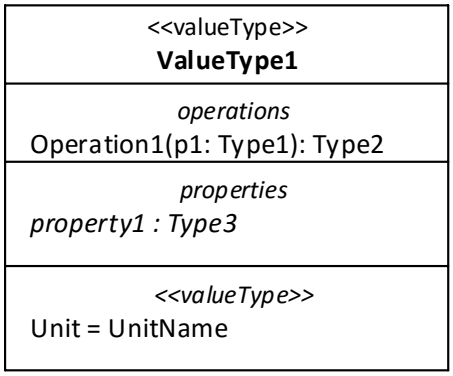


Abbildung 7.1: Syntax ValueType

Actor

1. **Beschreibung:** Der Actor beschreibt ein Element, das in irgendeiner Form mit einem System interagiert. Dabei kann der Actor ein Mensch, aber auch ein anderes System sein. In der Regel findet man einen Actor in UseCase-Diagrammen.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja

4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.2

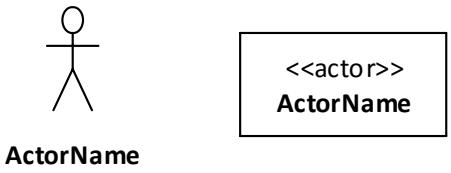


Abbildung 7.2: Syntax Actor

Enumeration

1. **Beschreibung:** Die Enumeration ist eine Aufzählung von definierten Werten, die in dem Model verwendet werden (vergleichbar mit dem enum-Typ in Java). Es basiert auf dem UML-Element "data type".
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.3

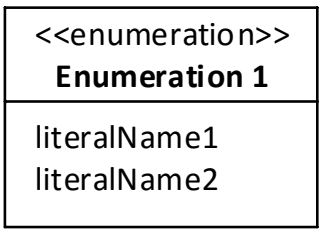


Abbildung 7.3: Syntax Enumeration

Generalization

1. **Beschreibung:** Die Generalization beschreibt die Beziehung eines generellen Elementes und einem spezialisierten Elementes (z.B. die Klassenhierar-

chie in einem Softwaresystems).

2. **Kategorie:** Unidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.4

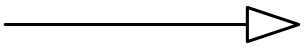


Abbildung 7.4: Syntax Generalization

BidirectionalConnector

1. **Beschreibung:** Der BidirectionalConnector verbindet zwei Elemente innerhalb eines InternalBlockDiagram. Die Verbindung ist ungerichtet.
2. **Kategorie:** Bidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.5

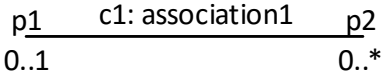


Abbildung 7.5: Syntax BidirectionalConnector

InternalBlockDiagram

1. **Beschreibung:** Das InternalBlockDiagram beinhaltet die internen Strukturen eines Blockes. Es veranschaulicht die Eigenschaften des Blockes und die Beziehungen zwischen diesen (vergleichbar mit dem Objektdiagramm in UML).
2. **Kategorie:** Diagramm
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Strukturdiagramm
5. **Syntax:** Abbildung 7.6

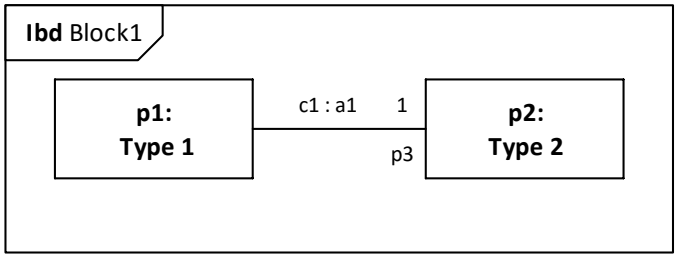


Abbildung 7.6: Syntax InternalBlockDiagram

PartAssociation

1. **Beschreibung:** Die PartAssociation verhält sich wie die ReferenceAssociation mit der Eigenschaft "aggregationKind = composite".
2. **Kategorie:** Bidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.7

SharedAssociation

1. **Beschreibung:** Die PartAssociation verhält sich wie die ReferenceAssociation mit der Eigenschaft "aggregationKind = shared".

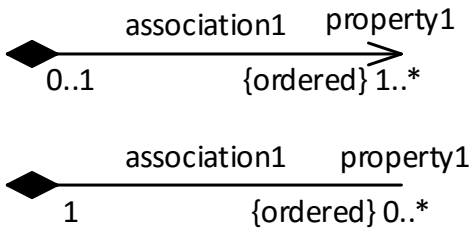


Abbildung 7.7: Syntax PartAssociation

2. **Kategorie:** Bidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.8

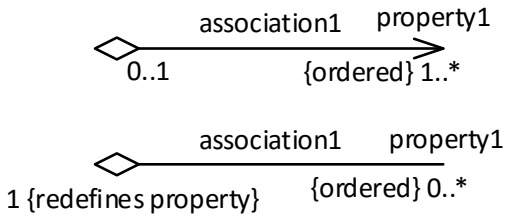


Abbildung 7.8: Syntax SharedAssociation

InstanceSpecification

1. **Beschreibung:** Eine InstanceSpecification repräsentiert eine konkrete Instanz eines abstrakten Konstruktes innerhalb des Modells.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** PackageableElement

5. **Syntax:** Abbildung 7.9

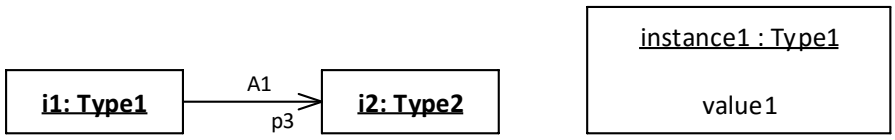


Abbildung 7.9: Syntax InstanceSpecification

AbstractDefinition

1. **Beschreibung:** Die AbstractDefinition repräsentiert die abstrakte Definition eines Elementes. Sie besitzt keinerlei Inhalt wie Eigenschaften oder Operationen, sondern lediglich den Namen.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Keine direkte Implementierung, allerdings indirekt über abstrakte Blöcke
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.10

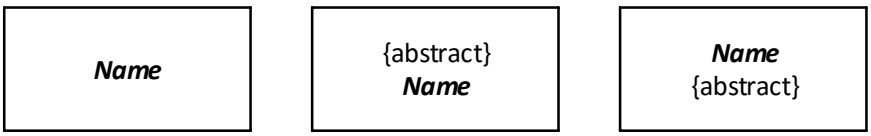


Abbildung 7.10: Syntax AbstractDefinition

Unit

1. **Beschreibung:** Eine Unit ist die Beschreibung einer Einheit. Sie basiert auf der InstanceSpecification.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja

4. **Abstraktion:** PackageableElement
5. **Syntax:** Abbildung 7.11

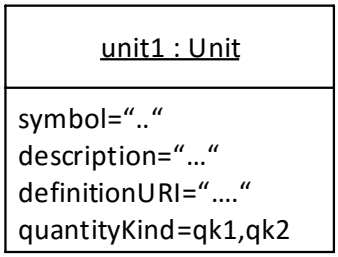


Abbildung 7.11: Syntax Unit

QuantityKind

1. **Beschreibung:** Das QuantityKind-Element basiert auf der InstanceSpecification und beschreibt das Maß, in der die Unit misst, z.B. wird das Gewicht in der Einheit Gramm gemessen.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Nein, allerdings indirekt über die Dimension
4. **Abstraktion:** PackageableElement
5. **Syntax:** Abbildung 7.12

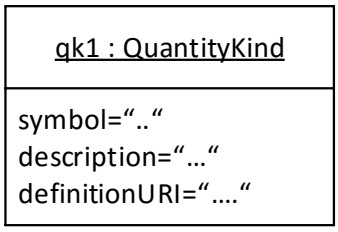


Abbildung 7.12: Syntax QuantityKind

GeneralizationSet

1. **Beschreibung:** Das GeneralizationSet fasst mehrere Generalization-Beziehung zusammen, die das gleiche generelle Element haben. Es ist gleich zu behandeln wie die Generalization.
2. **Kategorie:** Unidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** PackageableElement
5. **Syntax:** Abbildung 7.13

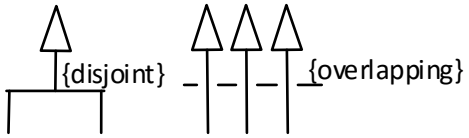


Abbildung 7.13: Syntax GeneralizationSet

BehaviorCompartment

1. **Beschreibung:** Das BehaviorCompartment ist ein Verweis auf das Verhalten eines Blockes. Jeder Block kann genau ein "classifier behavior" und zusätzlich mehrere "owned behavior" besitzen. Diese sind in textueller Form repräsentiert.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** -
5. **Syntax:** Abbildung 7.14

NamespaceCompartment

1. **Beschreibung:** Das NamespaceCompartment ist in der Lage Blöcke innerhalb eines Blockes darzustellen, die in dessen Namespace definiert sind.

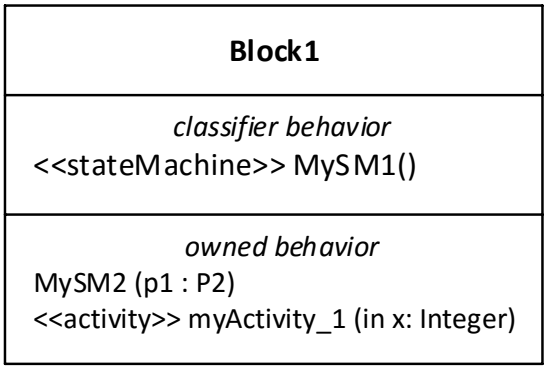


Abbildung 7.14: Syntax BehaviorCompartment

Dieser Abschnitt beinhaltet grafische Elemente.

2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** -
5. **Syntax:** Abbildung 7.15

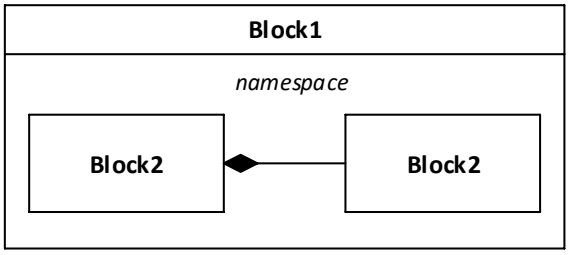


Abbildung 7.15: Syntax NamespaceCompartment

StructureCompartment

1. **Beschreibung:** Das StructureCompartment beinhaltet Verbindungen und andere interne Strukturen, die den beinhaltenden Block betreffen. Dieser Abschnitt beinhaltet grafische Elemente.
2. **Kategorie:** Property

3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** -
5. **Syntax:** Abbildung 7.16

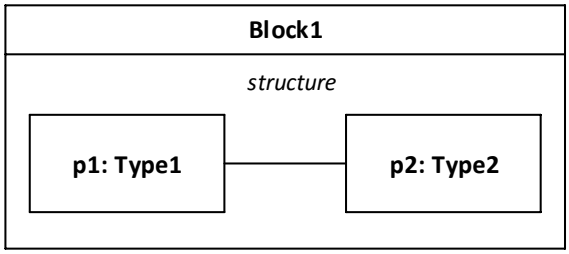


Abbildung 7.16: Syntax StructureCompartment

StereotypePropertyCompartment

1. **Beschreibung:** Das StereotypePropertyCompartment definiert ein Stereotype genauer innerhalb eines Blocks.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** -
5. **Syntax:** Abbildung 7.17

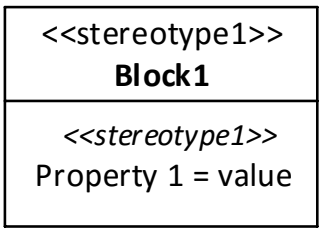


Abbildung 7.17: Syntax StereotypePropertyCompartment

BoundReference

1. **Beschreibung:** Die BoundReferences werden innerhalb eines Blockes definiert. Sie geben die oberen und unteren Grenzen einer Eigenschaft an.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.18

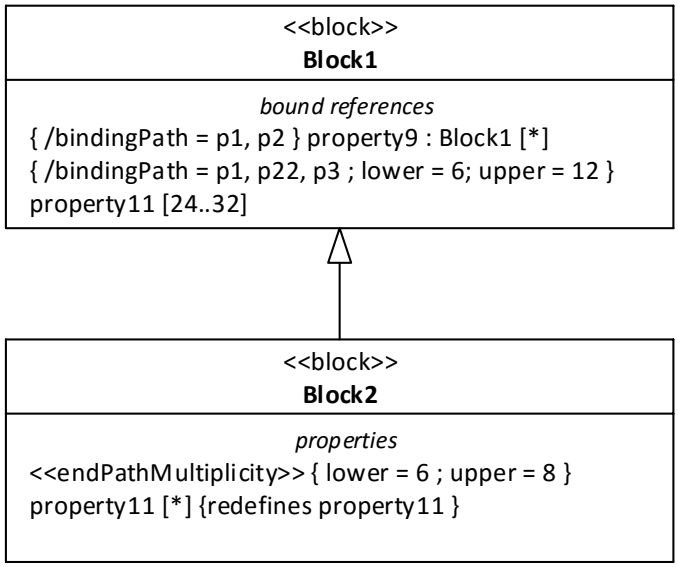


Abbildung 7.18: Syntax BoundReference

PropertySpecificType

1. **Beschreibung:** Der PropertySpecificType spezialisiert den angegebenen Typ des beinhaltenden Blockes. Der Typ wird dazu in eckige Klammern gesetzt und die spezifischen Änderungen in einen eigenen Abschnitt geschrieben.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Classifier

5. **Syntax:** Abbildung 7.19

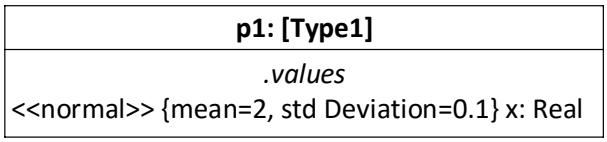


Abbildung 7.19: Syntax PropertySpecificType

ParticipantProperty

1. **Beschreibung:** Das ParticipantProperty ist eine Eigenschaft, die mit dem Stereotype participant markiert wird, um die beteiligten Elemente einer Assoziation genauer beschreiben zu können.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.20

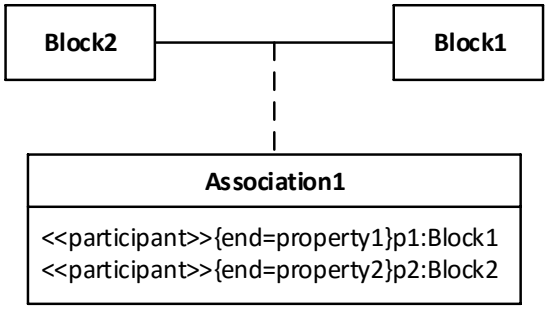


Abbildung 7.20: Syntax ParticipantProperty

ConnectorProperty

1. **Beschreibung:** Das ConnectorProperty wird mit dem Stereotype connector markiert und verweist auf eine Assoziation, die mit dem Block verbunden ist.

2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.21

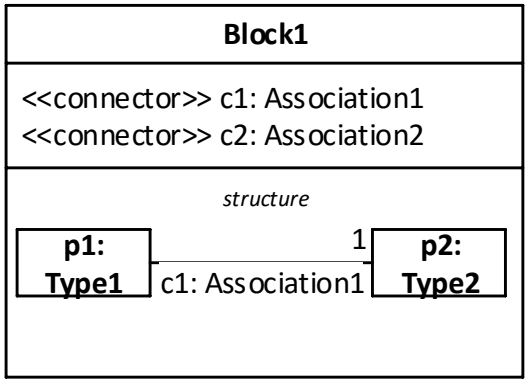


Abbildung 7.21: Syntax ConnectorProperty

BlockNamespaceContainment

1. **Beschreibung:** Das BlockNamespaceContainment zeigt an, in welchem Package oder Namespace sich ein Block befindet.
2. **Kategorie:** Unidirektionale Beziehung
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** NestedClassifier
5. **Syntax:** Abbildung 7.22

ActorPart

1. **Beschreibung:** Ein ActorPart ist eine Eigenschaft, die einem Block angehängt wird. Sie verweißt auf einen Actor.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Ja

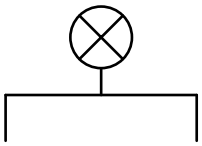


Abbildung 7.22: Syntax BlockNamespaceContainment

4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.23

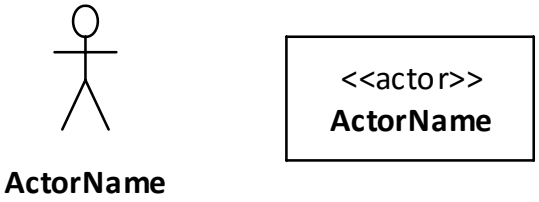


Abbildung 7.23: Syntax ActorPart

BindingConnector

1. **Beschreibung:** Der BindingConnector verbindet zwei Elemente miteinander und stellt sicher, dass an beiden Enden alle Properties den gleichen Wert haben.
2. **Kategorie:** Bidirektionale Kante
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.24

UnidirectionalConnector

1. **Beschreibung:** Der BidirectionalConnector verbindet zwei Elemente innerhalb eines InternalBlockDiagram. Die Verbindung ist gerichtet.

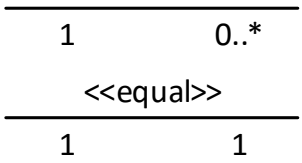


Abbildung 7.24: Syntax BindingConnector

2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.25

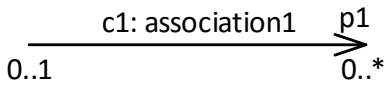


Abbildung 7.25: Syntax UnidirectionalConnector

A.2 Modellierungselemente

ConstraintNote

1. **Beschreibung:** Eine ConstraintNote enthält eine Bedingung, die für ein Element erfüllt werden muss.
2. **Kategorie:** Annotiertes Element
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Constraint
5. **Syntax:** Abbildung 7.26

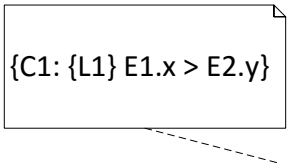


Abbildung 7.26: Syntax ConstraintNote

Realization

1. **Beschreibung:** Realization is eine Abhängigkeit zwischen einer Spezifikation und seiner Implementierung.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Dependency
5. **Syntax:** Abbildung 7.27

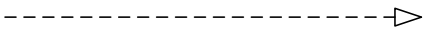


Abbildung 7.27: Syntax Realization

Refine

1. **Beschreibung:** Refine zeigt an, dass eine Anforderung von einem Element verfeinert bzw. genauer beschrieben wird.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Dependency
5. **Syntax:** Abbildung 7.28

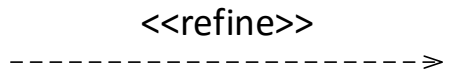


Abbildung 7.28: Syntax Refine

Stakeholder

1. **Beschreibung:** Ein Stakeholder beschreibt eine Person oder eine Gruppe, die in jeglicher Hinsicht mit dem zu beschreibenden System und dessen Entwicklung zu tun hat.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.29

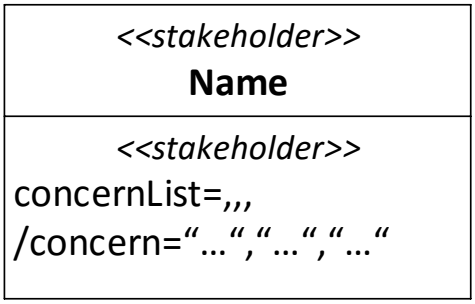


Abbildung 7.29: Syntax Stakeholder

View

1. **Beschreibung:** Eine View zeigt eine konkrete Sicht auf die Architektur eines Systems.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier

5. **Syntax:** Abbildung 7.30

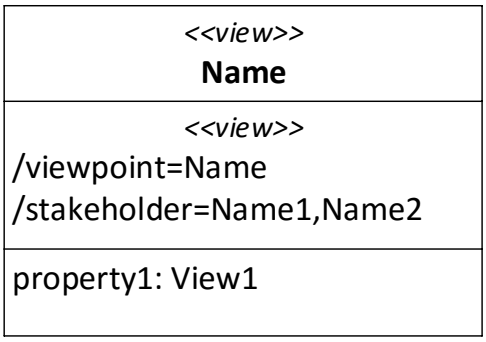


Abbildung 7.30: Syntax View

Viewpoint

1. **Beschreibung:** Ein Viewpoint ist ein Sichtpunkt auf ein System. Dieser betrifft je nach Stakeholder verschiedene Aspekte der Architektur. Aus einem Viewpoint wird eine View erstellt.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.31

Problem

1. **Beschreibung:** Ein Problem ist eine Erweiterung des Comments. Das Element wird verwendet um jegliche Probleme, seien es Missachtungen oder Limitierungen, während des Entwicklungsprozesses gegenüber der Anforderungen zu dokumentieren.
2. **Kategorie:** Annotiertes Element
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Comment
5. **Syntax:** Abbildung 7.32

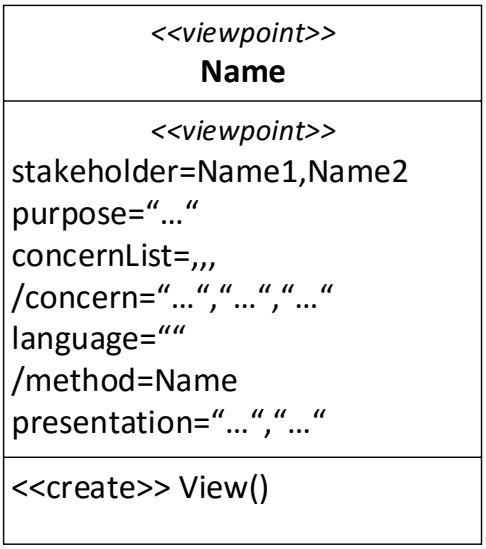


Abbildung 7.31: Syntax Viewpoint

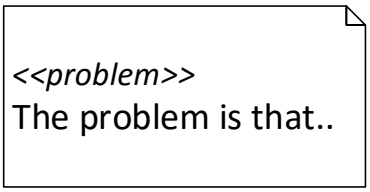


Abbildung 7.32: Syntax Problem

Rationale

1. **Beschreibung:** Ein Rationale dokumentiert jegliche Art von Entscheidungen bzw. Begründungen für Designentscheidungen. Sie werden wie ein Comment an ein Modellelement annotiert.
2. **Kategorie:** Annotiertes Element
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Comment
5. **Syntax:** Abbildung 7.33

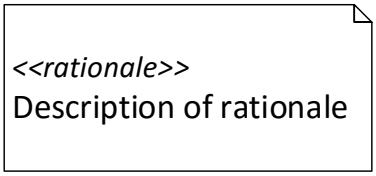


Abbildung 7.33: Syntax Rationale

ElementGroup

1. **Beschreibung:** ElementGroup gruppiert verschiedene Elemente anhand eines oder mehrerer Merkmale, die innerhalb der ElementGroup definiert sind. Sie wird an die jeweiligen Elemente annotiert.
2. **Kategorie:** Annotiertes Element
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Comment
5. **Syntax:** Abbildung 7.34

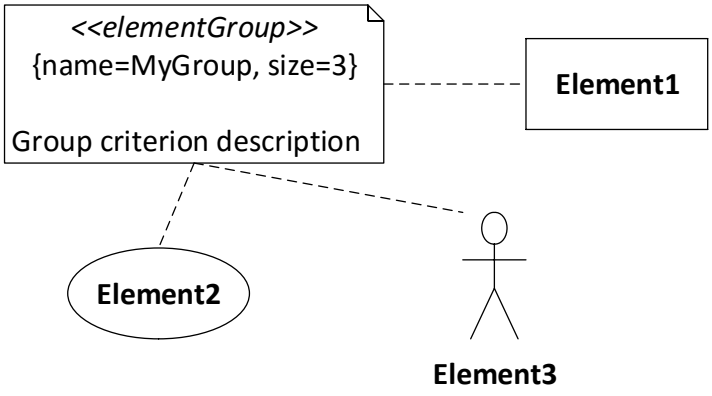


Abbildung 7.34: Syntax ElementGroup

ConstraintTextualNote

1. **Beschreibung:** Eine ConstraintTextualNote enthält eine Bedingung in textueller Form, die direkt neben einem Element geschrieben wird.
2. **Kategorie:** Annotiertes Element

3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Constraint
5. **Syntax:** Abbildung 7.35

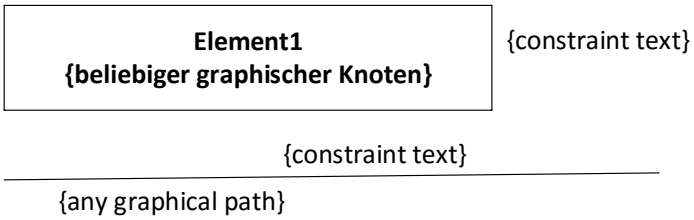


Abbildung 7.35: Syntax ConstraintTextualNote

Model

1. **Beschreibung:** Ein Model zeigt ein System aus einer bestimmten Sicht.
2. **Kategorie:** Strukturelement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Package
5. **Syntax:** Abbildung 7.36

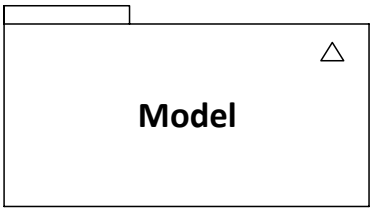


Abbildung 7.36: Syntax Model

PackageDiagram

1. **Beschreibung:** Ein PackageDiagram zeigt ein System auf Paketebene.
2. **Kategorie:** Diagramm

3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Diagram
5. **Syntax:** Abbildung 7.37

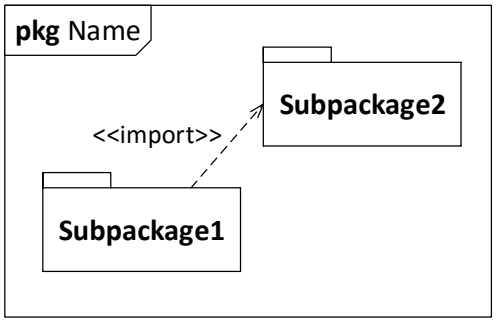


Abbildung 7.37: Syntax PackageDiagram

PublicPackageImport (visibility = public)

1. **Beschreibung:** Der PublicPackageImport zeigt die Abhängigkeit von einem Package.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.38

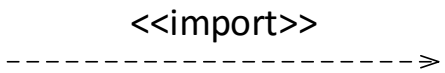


Abbildung 7.38: Syntax PublicPackageImport

PrivatePackageImport (visibility = private)

1. **Beschreibung:** Der PrivatePackageImport hat dieselbe Semantik wie der PublicPackageImport mit einer anderen Sichtbarkeit.

2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Relationship
5. **Syntax:** Abbildung 7.39

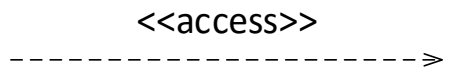


Abbildung 7.39: Syntax PrivatePackageImport

PackageContainment

1. **Beschreibung:** Das PackageContainment beschreibt die Zugehörigkeit eines Elementes zu einem Package.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** DirectedRelationship
5. **Syntax:** Abbildung 7.40

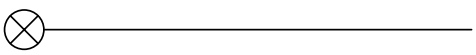


Abbildung 7.40: Syntax PackageContainment

Conform

1. **Beschreibung:** Conform ist eine spezielle Form der Generalisierung zwischen einem View und einem Viewpoint. Sie wird als Stereotyp an eine Generalisierung angehängt.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Ja

4. **Abstraktion:** Generalization
5. **Syntax:** Abbildung 7.41

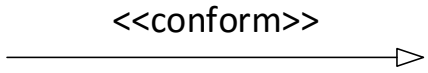


Abbildung 7.41: Syntax Conform

Expose

1. **Beschreibung:** Expose verbindet eine View mit einem oder mehreren Modellelementen. Diese dienen der View als Schnittstelle, um alle gewüschten Informationen zu extrahieren, um diese darstellen zu können.
2. **Kategorie:** Unidirektionale Kante
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Generalization
5. **Syntax:** Abbildung 7.42

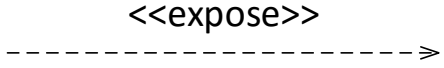


Abbildung 7.42: Syntax Expose

A.3 Ports und Flows

Port

1. **Beschreibung:** Ein Port ist eine Schnittstelle, über die ein Element mit anderen Elementen oder internen Bestandteilen interagieren kann. Die Varianten “Compartment Notation“ und “Port with Compartment“ haben dieselbe Semantik, sie besitzen lediglich eine andere Darstellungsform.
2. **Kategorie:** Property

3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.43

Abbildung 7.43: Syntax Port

FlowProperty

1. **Beschreibung:** FlowProperties spezifizieren die Art der Elemente, die zwischen dem eigenen Block und seiner Umgebung hin- und herfließen. Dies können sowohl Daten sein, als auch abstrakte Dinge wie z.B. Strom oder sonstige Materialien.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Ja, als FlowProperties innerhalb einer FlowSpecification (wird als Basiselement behandelt)
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.44

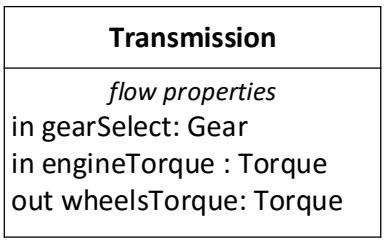


Abbildung 7.44: Syntax FlowProperty

Item Flow

1. **Beschreibung:** Item Flows geben an, welche Daten oder Materialien konkret über Konnektoren oder Assoziationen zwischen Blöcken fließen.
2. **Kategorie:** Gerichtete Kante
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Dependency
5. **Syntax:** Abbildung 7.45

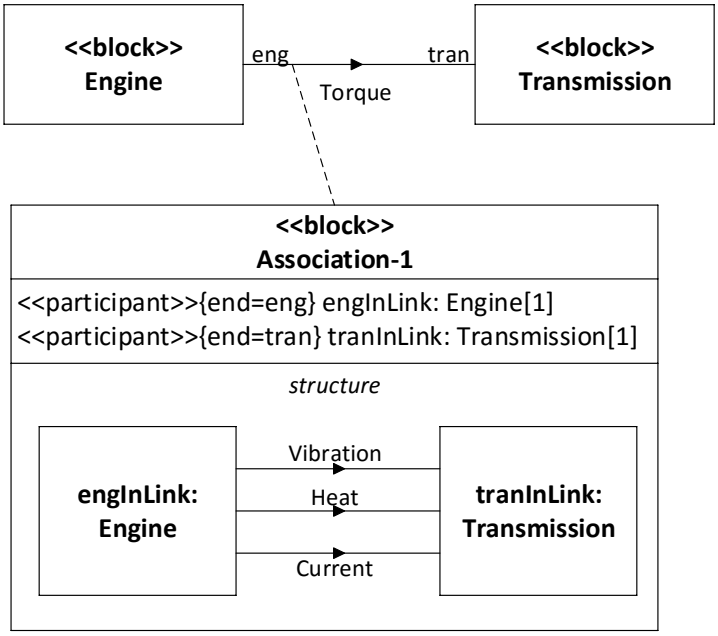


Abbildung 7.45: Syntax Item Flow

Interface

1. **Beschreibung:** Ein Interface gibt an, welche Schnittstelle ein Block zur Verfügung stellt, der von dem Interface erbt.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Classifier
5. **Syntax:** Abbildung 7.46

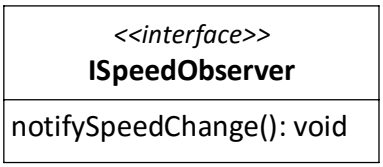


Abbildung 7.46: Syntax Interface

Required and Provided Interfaces

1. **Beschreibung:** Ein Port kann angeben, welche Schnittstellen er anbietet oder benötigt.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Ja
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.47

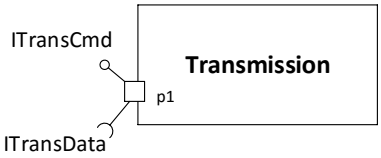


Abbildung 7.47: Syntax Required and Provided Interfaces

Port (Nested)

1. **Beschreibung:** Der Port(Nested) ist eine Erweiterung des normalen Ports, indem ein Port innerhalb eines Portes liegt, diesen also hierarchisch verfeinert. Er unterliegt denselben Regeln wie der Port.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.48

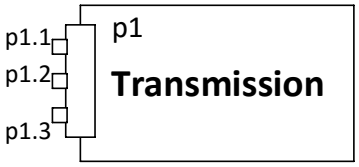


Abbildung 7.48: Syntax Port (Nested)

ProxyPort

1. **Beschreibung:** Ein ProxyPort erweitert einen Port. Er gibt an, welche Teile des eigenen Blocks für Konnektoren nach außen hin sichtbar sein sollen. ProxyPorts sind auch als "Compartment Notation" vorhanden.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.49

Abbildung 7.49: Syntax ProxyPort

FullPort

1. **Beschreibung:** Ein FullPort erweiter einen Port. Er definiert die Schnittstelle mit den Features des eigenen Blocks. FullPorts sind auch als "Compartment Notation" vorhanden.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Property
5. **Syntax:** Abbildung 7.50

Abbildung 7.50: Syntax FullPort

Required and Provided Features

1. **Beschreibung:** Ein Block kann Properties oder Operationen angeben, die er entweder zur Verwendung für andere Blöcke deklariert oder die er selbst benötigt, um korrekt arbeiten zu können.
2. **Kategorie:** Property
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Abhängig von dem Typ, der gebraucht bzw. angeboten wird
5. **Syntax:** Abbildung 7.51

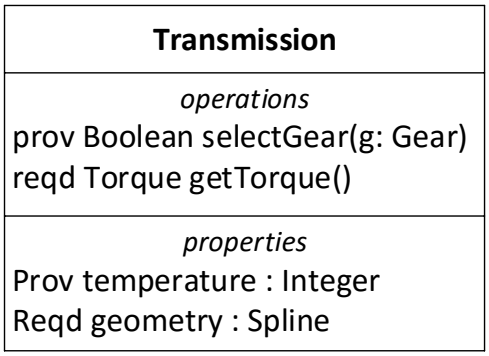


Abbildung 7.51: Syntax Required and Provided Features

InterfaceBlock

1. **Beschreibung:** Ein InterfaceBlock funktioniert analog zu einem Interface, er wird lediglich als Block definiert.
2. **Kategorie:** Basiselement
3. **Unterstützung in Papyrus:** Nein
4. **Abstraktion:** Classifier

5. **Syntax:** Abbildung 7.52

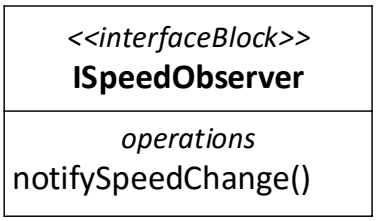


Abbildung 7.52: Syntax InterfaceBlock

Anhang B Evaluation

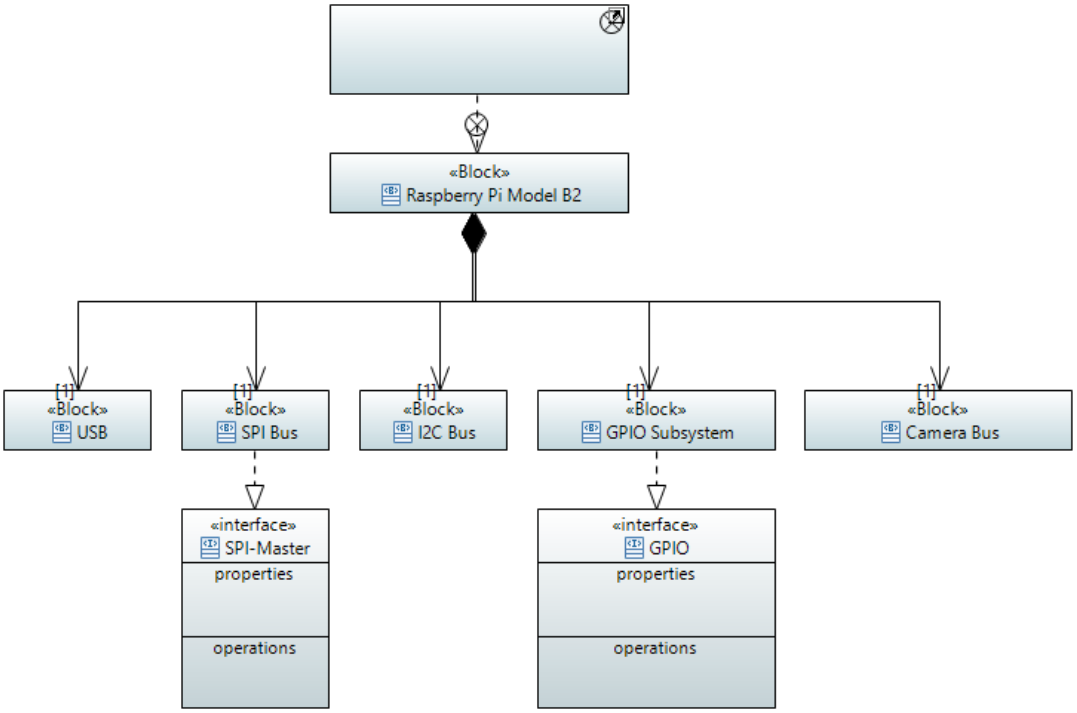


Abbildung 7.53: Raspberry Pi-Variante – Raspberry Pi Diagramm

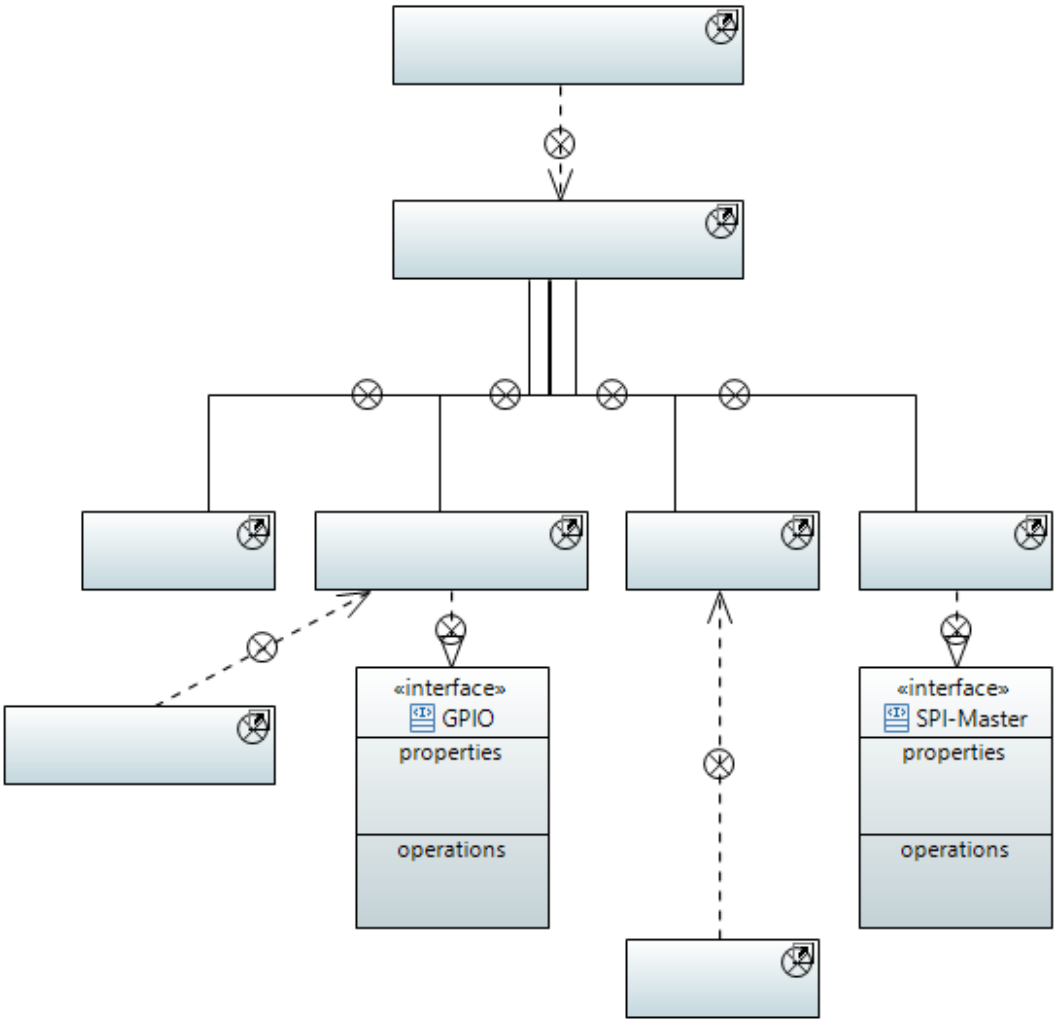


Abbildung 7.54: Raspberry Pi-Variante – Arduino Diagramm

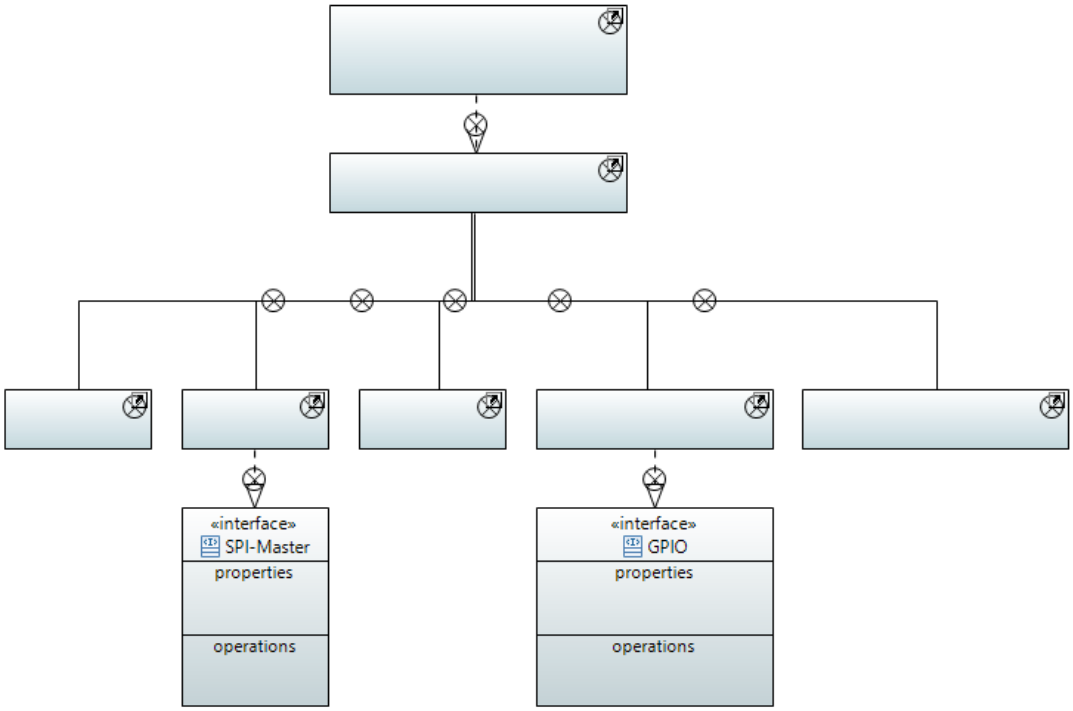


Abbildung 7.55: Arduino-Variante – Raspberry Pi Diagramm

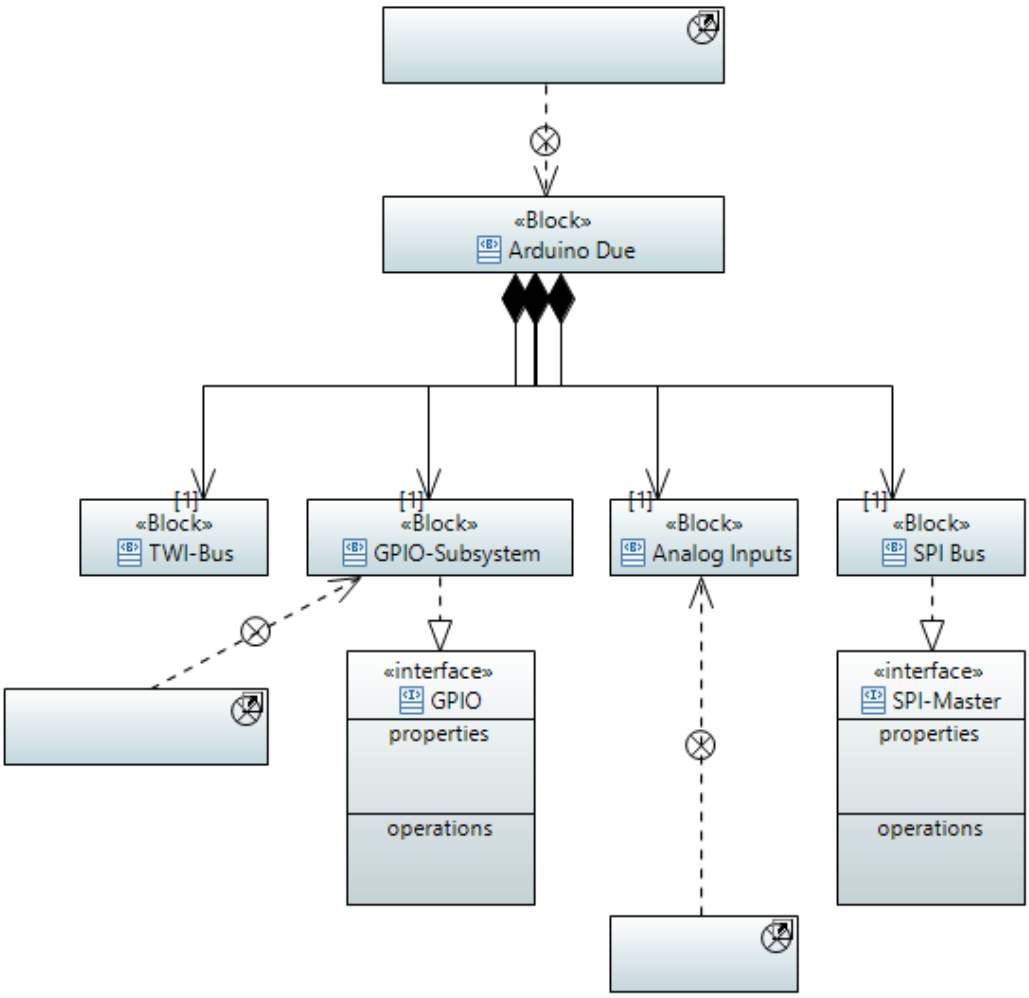


Abbildung 7.56: Arduino-Variante – Arduino Diagramm

Glossar

Dagu Rover 5 Ein modularer Roboter, der für diese Arbeit als Beispiel dient.

Eclipse Eclipse ist eine Entwicklungsumgebung für verschiedenste Anwendungen basierend auf Java.

Featuremodell Ein Featuremodell ist eine Abstraktion aller Funktionalitäten, die ein Plattformmodell anbietet.

Papyrus Papyrus ist ein Modellierungswerkzeug, welches auf Eclipse basiert.

Plattformmodell Ein Plattformmodell ist ein Modell, das sämtliche Variationen eines Systems enthält. Aus diesem können Varianten abgeleitet werden.

Product Line Engineering Unter Product Line Engineering versteht man eine Methode, um verschiedene Variationen eines Produktes zu entwickeln, die auf einer gemeinsamen Basis beruhen.

SysML SysML ist eine Modellierungssprache und basiert auf einer Teilmenge von UML mit eigenen Erweiterungen.

UML UML ist eine Modellierungssprache zur Spezifikation von Software oder anderen Systemen.

Variante Eine Variante ist eine konkrete Variation, die aus dem Plattformmodell abgeleitet wurde.

Akronyme

DM Domänenmodell.

EMF Eclipse Modeling Framework.

FM Featuremodell.

MQB Modulare Querbaukasten.

OMG Object Management Group.

PLE Product Line Engineering.

PM Plattformmodell.

SPL Softwareproduktlinien.

SysML Systems Modeling Language.

UML Unified Modeling Language.

Literaturverzeichnis

Atkinson, C. (2002). *Component-based product line engineering with uml*. Pearson Education.

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co. New York.

Clements, P. & Northrop, L. M. (2007). *Software product lines: practices and patterns*. Pearson Education.

Kang, K. C., Lee, J. & Donohoe, P. (2002). Feature-oriented product line engineering. *Software*, (4), 58–65.

Lee, J. & Kang, K. C. (2006). A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. *Software Product Line Conference, 2006 10th International*, 10pp.–140.

Matinlassi, M. (2004). Comparison of software product line architecture design methods: copa, fast, form, kobra and qada. *Proceedings of the 26th International Conference on Software Engineering (ICSE04)*.

OMG. (2015). Omg systems modeling language. Zugriff unter http://www.omg.org/spec/SysML/1.4/

Pohl, K., Böckle, G. & Van der Linden, F. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science+Business.

Weiss, D. M. & Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co. Boston.