EIN SIMULATIONSBASIERTES VERFAHREN ZUR GEWICHTSREDUZIERUNG VON KFZ-KABELBÄUMEN

Karlsruher Institut für Technologie

Elektrotechnik und Informationstechnik

Haoyu Sun
Matrikelnummer: 2498100
uofcz@student.kit.edu

Masterarbeit

Gutachterin: Prof. Dr. Jasmin Aghassi-Hagmann
Zweiter Gutachter: Prof. Dr.-Ing. Ahmet Cagri Ulusoy
Betreuer: Maximilian Müller
maximilian.mueller@edag.com

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AUTOSAR Automotive Open System Architecture

BLA Behavioral Logical Architecture

CAN Controller Area Network

DSM Data Store Memory

DSR Data Store Read

DSW Data Store Write

E/E Elektrik/Elektronik

EAST-ADL Electronics Architecture and Software Technology - Architecture Description Language

FCS Function Call Subsystem

HA Hardware Architektur

HWC Hardware Component

JAXB Java Architecture for XML Binding

JDOM Java Document Object Model for XML

KBL Kabelbaumliste

LA Logische Architektur

ME-API-J Matlab Engine API for Java

MSBE Model Based Systems Engineering

PLPT Power Line Parameters Tool

PSL Pi Section Line

SA Software Architektur

SESPS Simscape Electrical Specialized Power Systems

SWC Software Component

VEC Software Component

XML Extensible Markup Language

1 Einleitung

1.1 Einführung

Der Bereich der Fahrzeugelektrik und Elektronik hat in den letzten Jahrzehnten beeindruckende Fortschritte erlebt. Diese Entwicklung begann in den 1970er und 1980er Jahren mit der Einführung von Motorsteuergeräten und Tempomaten (Cruise Control). Anschließend wurden Systeme wie das Anti-Blockier-System, Getriebesteuerungen (Gearbox Control) und die Traktionskontrolle entwickelt. Ab den 1990er Jahren lag der Fokus auf der Entwicklung des Elektronisches Stabilitätsprogramm, Airbags und der Weiterentwicklung bestehender Systeme, wie etwa Adaptive Cruise Control und Adaptive Gearbox Control. Um die Jahrtausendwende kamen die ersten Fahrerassistenzsysteme auf den Markt, die seitdem kontinuierlich weiterentwickelt werden [6].

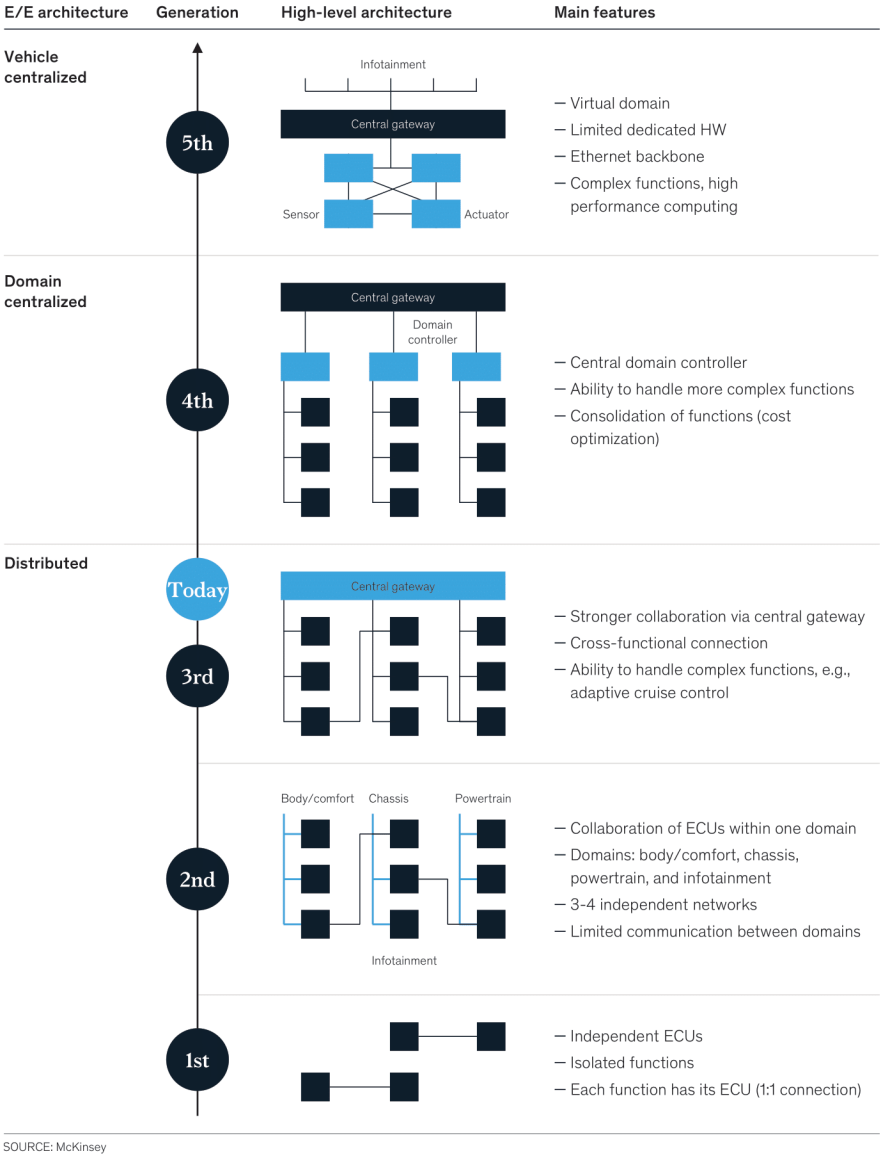

Abbildung 1.1: Evolution der E/E-Architektur (Quelle: [4, S. 11])

Die Integration neuer Funktionalitäten und Technologien in Fahrzeuge führen zu einer signifikanten Erhöhung der Fahrzeugkomplexität. Diese muss durch Software und die zugrunde liegende E/E-Architektur kompensiert werden [11]. Die E/E-Architektur umfasst sämtliche elektrischen und elektronischen Komponenten eines Systems, die zur Erfüllung von Funktionen und Funktionalitäten im Fahrzeug dienen. Elektrische Bauteile sind dabei die analogen Komponenten des Systems, während elektronische Bauteile die digital arbeitenden Komponenten umfassen. In klassischen, verteilten Architekturen wurde neue Funktionalität durch die Einführung zusätzlicher Steuergeräte sowie entsprechender Sensorik und Aktorik umgesetzt. Der aktuelle Trend geht jedoch in Richtung funktionale Vernetzung durch Konsolidierung und Integration von Funktionalität in einem einzigen Steuergerät.

Abbildung 1.1 veranschaulicht schematisch die Entwicklung der E/E-Architektur hin zu einer hierarchischen Domänen-Architektur der vierten Generation. Diese besteht aus mehreren zentralen Domänen-Steuergeräten, wie beispielsweise für Infotainment und Fahrerassistenzsysteme, sowie einem zentralen Kommunikations-Gateway. Diese Domänen-Architektur wird bereits von einigen Herstellern implementiert. In der fünften Generation soll schließlich eine vollständig zentralisierte Architektur entstehen. Es wird erwartet, dass diese aus leistungsfähigen, intelligenten Sensoren und Aktuatoren sowie wenigen hochperformanten Rechenplattformen besteht.

1.2 PREEvision

Mit der zunehmenden Komplexität, insbesondere der steigenden Softwarekomplexität, hat das Model Based Systems Engineering (MSBE) seine Bedeutung in der Entwicklung von E/E-Architekturen eindrucksvoll unter Beweis gestellt.

PREEvision[1] ist eine auf Eclipse basierende Entwurfs- und Analysewerkzeug und vollständig in Java implementiert. Durch den Plugin Mechanismus von eclipse ist es einfach erweiterbar. Die zugrunde liegende Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL) [2] bildet als komplexe Datenstruktur den Kern von PREEvision® zur durchgängigen und detaillierten Beschreibung von E/E-Architektur auf allen Abstraktionsebenen.

Abbildung 1.2 zeigt die in verschiedene Abstraktionsebenen unterteilte Struktur von PREEvision. Jede dieser Ebenen bietet angepasste Diagramme, die jeweils eine spezifische Perspektive widerspiegeln. Da der Verfasser dieser Arbeit selbst keine direkten Erfahrungen mit PREEvision hat, basiert die folgende Darstellung der Abstraktionsebenen auf der Arbeit von H. Bucher [3].

1. **Produktziele**: Auf der ersten Ebene werden die Produktziele dargestellt, die in drei Kategorien unterteilt sind: Anforderungen, Kundenfunktionen und Use Cases.

   Anforderungen und Kundenfunktionen bieten eine strukturierte und systematische Methode zur Entwicklung und Verwaltung von Anforderungen an einer E/E-Architektur (EEA). Ziel ist es, ein gemeinsames Verständnis zwischen Auftraggeber

[1] `https://www.vector.com/de/de/produkte/produkte-a-z/software/preevision/`, zuletzt aufgerufen am 15.06.2024.

und -nehmer zu schaffen. Kundenfunktionen beziehen sich auf erlebbare Funktionen, wie zum Beispiel eine automatisch geregelte Klimaanlage, während Anforderungen spezifische technische Merkmale beschreiben, wie etwa die zulässige Betriebstemperatur eines Steuergeräts. Anforderungen werden textuell formuliert, hierarchisch in Paketen organisiert und automatisch mit IDs versehen.

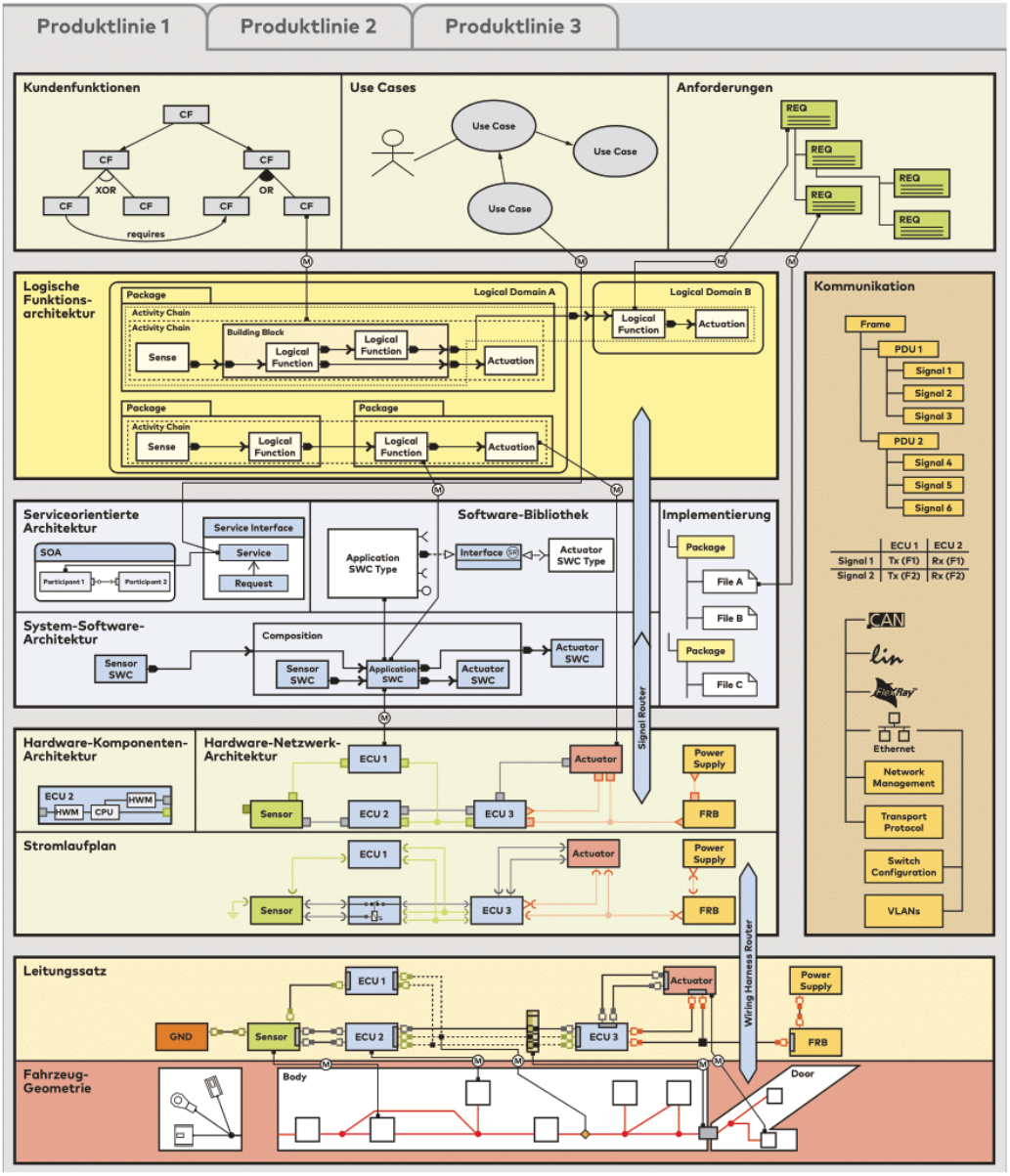

Abbildung 1.2: Abstraktionsebenen in PREEvision (Quelle: [12, S. 127])

Diese auf UML basierenden Use Case Diagramme ermöglichen die Modellierung von Systemfunktionen, wie der Fahrersitzeinstellung, aus der Sicht des Nutzers.

2. **Logische Architektur (LA)**: Die LA stellt die logische Perspektive verteilter Fahrzeugfunktionen dar. Die Modellierung der LA erfolgt durch Blockdiagramme, wobei atomare Funktionen und zusammengesetzte Blöcke zur Hierarchisierung

eingesetzt werden. Die Funktionen können mit Software- oder Hardwareartefakten verknüpft werden, wobei die Schnittstellenspezifikation die Grundlage für die Definition der zwischen den Funktionen bzw. zwischen den physikalischen Steuergeräten auszutauschenden Signale bildet. Zusätzlich können Funktionen mit Kundenfunktionen und/oder Anforderungen verknüpft werden, um nachzuvollziehen, welche Funktion die verknüpften Artefakte implementiert.

3. **Software Architektur (SA)**: Die Automotive Open System Architecture (AUTOSAR) ist ein 2003 gegründetes Konsortium, das sich aus Automobilherstellern, Zulieferern und weiteren Unternehmen zusammensetzt. AUTOSAR verfolgt das Ziel, die Entwicklung von Steuergeräte-Software durch eine standardisierte Softwarearchitektur, eine einheitliche Entwurfsmethodik und Applikationsschnittstellen für verschiedene Anwendungsbereiche, wie z. B. Komfortfunktionen, voranzutreiben [15, Kap. 7]. Zu den Hauptzielen gehören die Bewältigung der zunehmenden Software- und E/E-Architektur-Komplexität infolge der wachsenden Funktionsintegration im Fahrzeug, die Skalierbarkeit und Übertragbarkeit von Softwarefunktionen auf unterschiedliche Fahrzeugplattformen, die Wiederverwendbarkeit von Software, die Verkürzung der Entwicklungszyklen sowie die Reduktion von Entwicklungsrisiken und -kosten [2]. Das zentrale Konzept von AUTOSAR besteht darin, die Entwicklung der Anwendungssoftware unabhängig von der zugrunde liegenden Steuergeräte-Infrastruktur zu gestalten.

   Die System-Softwarearchitektur beschreibt die beteiligten Softwarekomponenten sowie deren Interaktion und ermöglicht ein AUTOSAR-konformes Softwaredesign. Das Design wird mithilfe einer Funktionsbibliothek bestehend aus Softwaretypen sowie deren Port-Typen.

4. **Hardware Architektur (HA)**: Die HA kann in drei Subebenen mit zunehmender Detaillierungsgrad unterteilt werden, die jeweils eigene Ansichten und Diagrammtypen umfassen: die logische Ebene, die elektrische Ebene (Stromlaufplan), die physikalische Ebene (Leitungssatz und Topologie).

   Die logische Ebene umfasst alle Hardwarekomponenten wie Steuergeräte (ECUs), Sensoren und Aktuatoren sowie die Netzwerktopologie der verwendeten Bussysteme, durch die die Komponenten mit logischen Busverbindungen verbunden sind. Zusätzlich können die Komponenten über konventionelle Verbindungen miteinander verbunden sein und analoge Signale austauschen. Die logische Sicht wird in einem Hardware-Vernetzungsdiagramm dargestellt.

   Die elektrische Subebene oder der Stromlaufplan verfeinert die logischen Verbindungen und Konnektoren durch eine oder mehrere dazugehörige elektrische Verbindungen, Beispielsweise wird die logische Verbindung eines Controller Area Network (CAN)-Busses durch je zwei elektrische Verbindungen verfeinert, die die differentielle Signalübertragung des CAN-Bussystems über die CAN-High- und CAN-Low-Pegel realisieren.

   Im Leitungssatz wird der Kabelbaum der E/E-Architektur modelliert, indem die elektrischen Verbindungen durch die physikalische Leitungen oder Kabeln verfeinert werden. Die Leitungen werden mit komponentenseitigen Pins an die Kompo-

[2] `https://www.autosar.org/about/`

nenten angeschlossen, die typischerweise Steckern zugeordnet sind. Der Import des standardisierten Austauschformats KBL wird ebenfalls unterstützt.Die Topologie beschreibt und visualisiert die Fahrzeuggeometrie

Mappings ermöglichen es, Artefakte sowohl auf unterschiedlichen Abstraktionsebenen als auch innerhalb derselben Ebene miteinander zu verknüpfen. In Abbildung 3.2 werden diese Verknüpfungen durch ein M symbolisiert. Sie gewährleisten eine durchgängige und nachvollziehbare Verbindung zwischen den Ebenen.

Mappings zwischen logischen Funktionen der LA und Softwarekomponenten ermöglichen eine detailliertere Verfeinerung der Funktionen in der Software, die anschließend auf Hardwarekomponenten abgebildet werden können. Alternativ kann eine logische Funktion direkt auf Recheneinheiten abgebildet werden, ohne die Softwareebene zu durchlaufen.

Ähnlich wie bei der LA können Softwarekomponenten auf Hardwarekomponenten wie Recheneinheiten oder andere Hardwaremodule abgebildet werden. Softwareports lassen sich mit logischen Hardware-Konnektoren oder schematischen Pins verknüpfen.

Um den Einbauort von Hardwarekomponenten wie Steuergeräten (ECUs), Sensoren oder Aktuatoren festzulegen, werden spezifische Zuordnungen genutzt. Auch Verbindungsstellen, sogenannte Splices, werden entsprechenden Einbauorten oder Anschlussstellen zugeordnet. Leitungen und Kabel müssen ebenfalls präzise zu den jeweiligen Segmenten der Topologie zugewiesen werden, damit nach dem Routing die exakten Längen bestimmt werden können. Diese Zuweisungen erfolgen während des Routing-Prozesses automatisch.

1.3 Zielsetzung

Die aktuelle Version von PREEvision ist nicht in der Lage, Aussagen über das Verhalten der entworfenen E/E-Architektur zu treffen. In der Praxis besteht jedoch ein erhebliches Interesse an solchen Informationen, da Fehler, die in der Entwurfsphase auftreten, oft kostspielig zu korrigieren sind. Diese Arbeit wird bei der EDAG Engineering AG durchgeführt, wo PREEvision für den Entwurf von E/E-Architekturen verwendet wird.

PREEvision kann Kabelbauminformationen aus der HA im KBL-Datenformat [10] exportieren. Ursprünglich war die Idee dieser Arbeit, eine Lösung zu entwickeln, die programmatisch ein ausführbares Simulationsmodell auf elektrischer Ebene generiert, wobei KBL als Informationsquelle dient. Diese Idee wurde jedoch später als nicht umsetzbar verworfen, da das KBL keine Komponenteninformationen enthält. Es bietet nämlich auch keine Verhaltensmodellierung der Komponenten wie ECUs, Sensoren oder Aktoren. Darüber hinaus konzentriert sich das KBL ausschließlich auf dem Kabelbaum und liefert keine Informationen zur SA. Für eine Simulation auf elektrischer Ebene wäre jedoch auch solche Informationen erforderlich, da die Software das Verhalten, wie den Stromverlauf eines Motors oder den gesamten Energieverbrauch, maßgeblich beeinflusst.

Was hingegen mit KBL als Informationsquelle möglich ist, ist eine thermische Simulation. Das Ziel dieser Arbeit besteht darin, eine Software in Java zu entwickeln, die in der Lage ist, aus KBL-Daten ein Simulink-Modell zu generieren, um den Temperatur-

verlauf der Leitungen zu simulieren. Am Ende soll durch die Simulation die optimale Querschnittsfläche der Leitungen ermittelt werden, um das Gewicht zu reduzieren.

Die Gewichtsreduktion des Bordnetzes ist ein anspruchsvolles Ziel, da elektrische Komponenten in Straßenfahrzeugen über Jahrzehnte hinweg an Bedeutung gewonnen haben. Heutzutage machen sie einen großen Teil der Herstellungskosten aus, ebenso wie das Leitungssystem immer schwerer, komplexer und teurer geworden ist. Mittelklassewagen können beispielsweise bereits mehr als 3 km Kabel mit einer Gesamtmasse von über 50 kg haben.

Die Länge der Leitungen ergibt sich aus den Einbauorten der elektrischen Komponenten und dem Routing. Eine optimale E/E-Architektur könnte zu kürzeren Gesamtlängen der Leitungen führen, ist jedoch nicht das Thema dieser Arbeit. Eine andere Möglichkeit, das Gewicht des Kabelbaumes zu reduzieren, besteht darin, eine optimale Querschnittsfläche der Leitungen zu finden. Aus Sicherheitsgründen werden in der Praxis die Querschnittsflächen der Leitungen oft unnötig groß dimensioniert. Leitungen müssen zwei Anforderungen erfüllen: Erstens darf der durch die Leitung verursachte Spannungsabfall einen maximalen Wert nicht überschreiten, was sich mit einer einfachen Berechnung überprüfen lässt. Zweitens darf die Temperatur der Leitung nicht zu hoch sein, da dies zum Verschmelzen der Isolierung führen könnte.

2 Technische Grundlagen und relevante Forschungen

PREEvision entstand aus einem gemeinsamen Forschungsprojekt zwischen Daimler-Chrysler (heute Daimler) und dem Forschungszentrum Informatik (FZI), unterstützt vom Institut für Technik der Informationsverarbeitung (ITIV) an der Universität Karlsruhe (TH), das im Jahr 2003 initiiert wurde. Das Projekt fokussierte sich auf die Konzeption von Elektrik/Elektronik-Architekturen in der frühen Entwurfsphase. Ein frühes Ergebnis dieses Projekts war das EEKonzeptTool, ein Vorläufer von PREEvision. Im Jahr 2005 wurde das Spin-Off-Unternehmen aquintos GmbH gegründet, um die Projektergebnisse weiterzuentwickeln und zur Marktreife zu bringen. Seit 2010 ist aquintos Teil der Vector-Gruppe, und PREEvision wird seither gemeinsam weiterentwickelt [6].

Vector Informatik GmbH am Standort Karlsruhe hat zahlreiche Doktoranden unterstützt, darunter H. Bucher und K. Neubauer, die während ihrer Promotion die Simulationsmöglichkeiten für E/E-Architekturen erforscht haben [3][9]. Vector bietet intern eine auf Eclipse basierende Entwicklerplattform an, die den Entwicklern vollen Lese- und Schreibzugriff auf die Daten des Architekturmodells gewährt. Über das Metrik-Framework können Entwickler zudem eigene Logiken implementieren. Im Gegensatz dazu ist EDAG lediglich Anwender von PREEvision, wobei die Entwicklungsplattform nicht zur Verfügung steht. In diesem Kapitel wird zunächst die Arbeit von H. Bucher und K. Neubauer vorgestellt. Anschließend wird auf die thermische Simulation eingegangen. Aufgrund des fehlenden Zugangs zur Entwicklerplattform sind die Forschungsarbeiten bei EDAG nicht nachvollziehbar, wodurch die thermische Simulation das einzig umsetzbare Thema bleibt.

2.1 Simulationsmöglichkeiten basierend auf Ptolemy II

H. Bucher verwendet Ptolemy II als Simulationswerkzeug. Ptolemy II ist ein quelloffener Simulator, der in Java implementiert ist. Die Modelle in Ptolemy II setzen sich aus einer Vielzahl miteinander verschachtelter Actors zusammen, die mit den Blöcken in Simulink vergleichbar sind. Da Ptolemy II, genau wie PREEvision, ebenfalls in Java implementiert ist, lässt es sich problemlos als Plugin in PREEvision integrieren. Somit können sowohl die Synthese der Simulationsmodelle als auch die Durchführung der Simulationen vollständig in PREEvision abgewickelt werden.

Abbildung 2.1 veranschaulicht den Prozessablauf der integrierten Lösung zur Simulation. Die grau dargestellten Bereiche repräsentieren die bereits in PREEvision vorhandenen Abstraktionsebenen, während die grün markierten Teile die neu implementierte Logik zeigen. Der E/E-Model Interpreter extrahiert aus den Daten des Architekturmodells alle relevanten Informationen für die Synthese. Nach der Synthese des Simulationsmodells werden die Simulationen durchgeführt, deren Ergebnisse anhand vordefinierter Metriken bewertet werden, um den Entwurf anschließend iterativ zu optimieren. Alles erfolgt direkt innerhalb von PREEvision, sodass der Nutzer nicht auf andere Softwarewerkzeuge wechseln muss. Dies vereinfacht den Prozess deutlich und spart wertvolle Zeit.

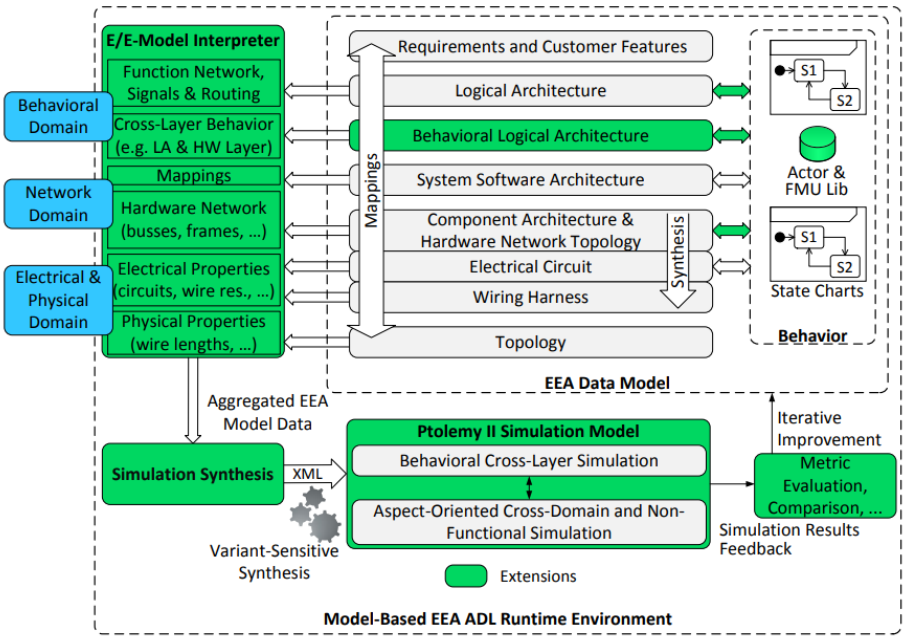

Abbildung 2.1: Prozesseverlauf vom Synthese bis zur Optimierung der E/E-Architektur (Quelle: [3])

Der ursprüngliche Artikel besagt, dass die prototypische Implementierung folgende Anforderungen erfüllt:

- Die Simulation der Netzwerkkommunikation zwischen Steuergeräten.
- Die Durchführung einer Analyse des quasi-statischen Strombedarfs.
- Die Simulation elektrischer Komponenten innerhalb von Steuergeräten, sowie die Analyse des dynamischen Strombedarfs.

2.1.1 Verhaltensmodellierung

Es ist nicht möglich, ein Simulationsmodell zu synthetisieren, ohne eine Verhaltensmodellierung aller beteiligten Komponenten zu berücksichtigen. Daher wurde eine neue Abstraktionsebene, die sogenannte BLA, eingeführt, um die statischen Funktionsblöcke auf der LA-Ebene durch eine detaillierte, ausführbare Verhaltensmodellierung zu verfeinern. Zwei zusätzliche Typen von Mapping werden eingeführt, um Blöcke und deren Ports auf der LA-Ebene mit den Blöcken und Ports der BLA-Ebene miteinander zu verknüpfen:

1. Block Instanz Mappings
2. Port-Prototyp Mappings

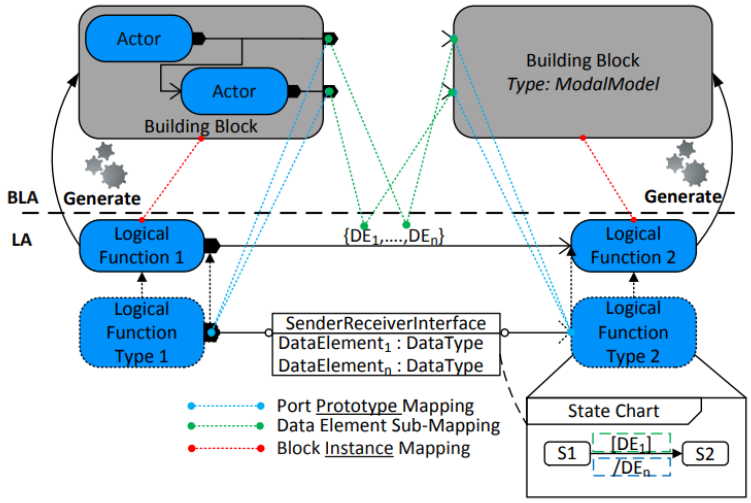

Abbildung 2.2: Verknüpfung der Funktionsblöcke und Statecharts mit den Blöcken auf der BLA-Ebene. (Quelle: [3])

Die Blöcke der BLA-Ebene modellieren das Verhalten der zugeordneten logischen Funktionen durch eine Menge strukturierter und miteinander verbundener Actors aus den Ptolemy II-Bibliotheken. Dabei obliegt es den Nutzern, das Verhalten der entsprechenden logischen Funktionen manuell zu modellieren. Eine logische Funktion wird mit einer oder mehreren Software Component (SWC)s auf der SA-Ebene verknüpft. Jede SWC ist wiederum einer ECU auf der HA-Ebene zugeordnet, was bedeutet, dass die Ausführung der SWC durch die zugeordnete ECU erfolgt. Eine direkte Verknüpfung zwischen einer logischen Funktion und einem Hardware-Artefakt ist ebenfalls möglich.

Ab Version 9.0 bietet PREEvision den Nutzern die Möglichkeit, das Verhalten von Systemen mithilfe von Statecharts zu modellieren. Auf diese Weise lassen sich Hardware-Artefakte, wie beispielsweise Prozessoren, ebenfalls durch Statecharts beschreiben. Um den quasi-statischen Strombedarf präzise zu erfassen, ist eine detaillierte Beschreibung des Stromverbrauchs in den jeweiligen Zuständen notwendig. Der Detailgrad der Strombeschreibung passt sich dabei an die Granularität der Statechart-Beschreibung auf Hardwareebene an. Abbildung 2.3 zeigt die relevanten Metaklassen und deren Attribute in PREEvision.

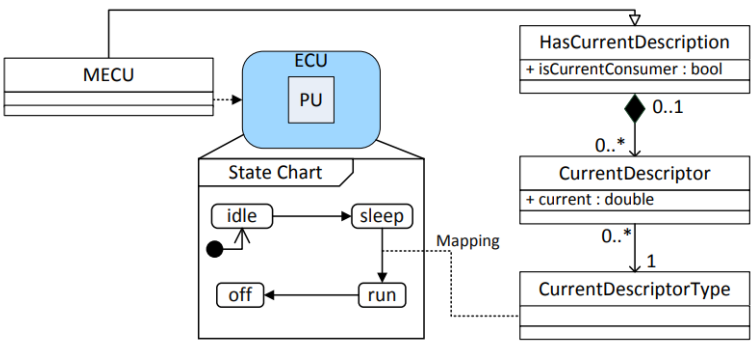

Abbildung 2.3: Hardwaremodellierung mithilfe von Statecharts und Verknüpfung der Zustandswechsel mit der entsprechenden Stromaufnahme. (Quelle: [3])

Bislang erfolgt die Modellierung lediglich auf einer rein logischen Ebene. Um jedoch eine Simulation auf elektrischer Ebene zu realisieren, sind zusätzliche Maßnahmen erforderlich. Logische Ports, die mit Hardware-Pins verknüpft sind, werden als analoge Ports gekennzeichnet. Analoger Port beistzt ein Datenelement mit einem Datentyp ELECTRICAL VOLTAGE.

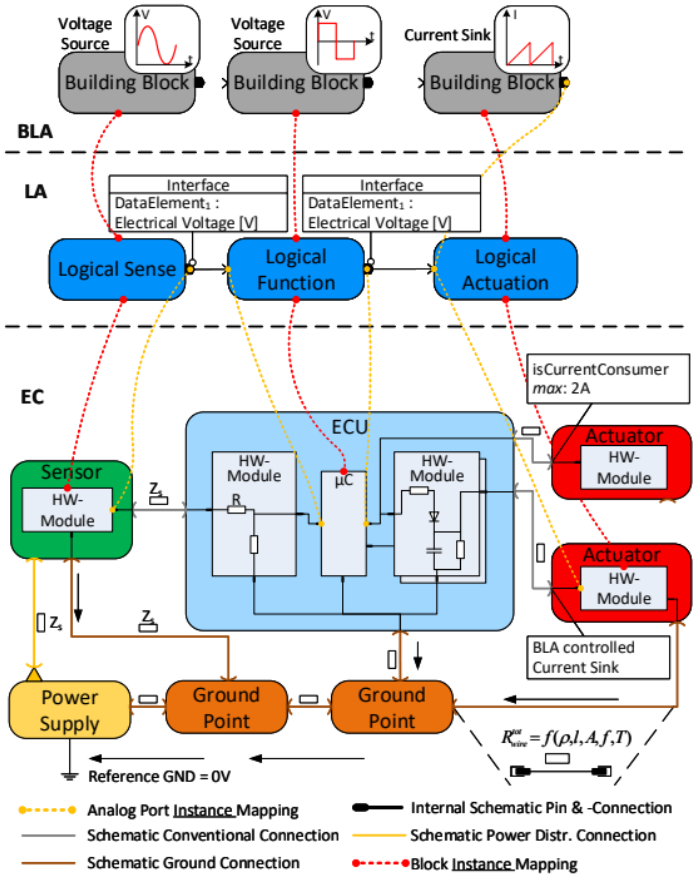

Abbildung 2.4: Modellierungserweiterungen zur elektrischen und leitungssatzsensitiven Verfeinerung von Verhalten. (Quelle: [3])

Ein Hardware-Artefakt kann dabei ein Prozessor, Sensor oder Aktor sein. Der Port, der mit dem Hardware-Pin eines Sensors verbunden ist, wird als Ausgangsport bezeichnet und im zugehörigen Block auf der BLA-Ebene als Spannungsquelle modelliert. Der Signalverlauf dieser Spannungsquelle kann eine beliebige Form annehmen. Der Port, der mit dem Hardware-Pin eines Aktors verbunden ist, wird als Eingangsport bezeichnet und im entsprechenden Block auf der BLA-Ebene als Stromverbraucher dargestellt.

Die Verbindung zwischen analogen Ports wird als konventionelle Leitung in Form von Impedanz berücksichtigt. Dadurch sind dynamische Analysen des Strombedarfs sowie Untersuchungen zum Einfluss elektrischer Komponenten und des Leitungssatzes auf das Gesamtverhalten möglich.

2.1.2 Simulationssynthese

Die Synthese eines ausführbaren Ptolemy II Modells aus den Daten des Architekturmodells sowie die anschließende Durchführung der Simulation erfolgen vollständig innerhalb von PREEvision. Für die Synthese ist eine Abbildungsvorschrift erforderlich, die die Artefakte der BLA den entsprechenden Ptolemy II Artefakten zuordnet.

Tabelle 2.1: Abbildungsvorschriften zwischen Artefakten der BLA und PtII Artefakten (Quelle: [3])

BLA Artefakt	**Beschreibung**	**Ptolemy II Artefakt**
LogicalFunction	Instanzen von atomaren, logischen Funktionstypen; repräsentieren eine Instanz eines konkreten Actors aus der Bibliothek, dieser kann atomar oder zusammengesetzt sein.	Instanz einer abgeleiteten TypedAtomicActor oder TypedCompositeActor Klasse
LogicalFunctionBlockType	Spezifiziert den Typ einer atomaren logischen Funktion; kapselt die konkrete Actor-Klasse, die instanziiert werden soll.	Konkrete Actor-Klasse bzw. ein als MoML-class spezifizierter, zusammengesetzter Actor
LogicalBuildingBlock	Hierarchisiert atomare, logische Funktionen oder wiederum logische Building Blocks; repräsentiert gewöhnliche, zusammengesetzte Actors oder davon abgeleitete.	Instanz einer (konkreten) TypedCompositeActor Klasse
LogicalBuildingBlockType	Spezifiziert den Typ eines Building Blocks.	(Konkrete) TypedCompositeActor Klasse
LogicalProvidedPort	Ausgangsport	TypedIOPort mit Flag isOutput
LogicalRequiredPort	Eingangsport	TypedIOPort mit Flag isInput
LogicalBuildingBlock Custom Attribute	Spezifiziert das Berechnungsmodell bzw. den Director eines ausführbaren, undurchlässigen Composite Actors.	Konkreter Director
LogicalAssemblyConnector	Verbindet logische Ports.	TypedIORelation
GenericAttribute	Parameter eines atomaren oder zusammengesetzten Actors.	Parameter

Statecharts werden in Ptolemy II durch sogenannte Modal Models dargestellt und si-

muliert. Dabei gibt es auch spezifische Abbildungsvorschriften, um die Artefakte von Statecharts auf die entsprechenden Strukturen in Ptolemy II Modal Models zu übertragen. Diese Abbildungsvorschriften regeln, wie die Zustände, Übergänge, Ereignisse und Aktionen aus den Statecharts in die Modal Models integriert werden, sodass eine korrekte Simulation der Zustandsübergänge und des Systemverhaltens in Ptolemy II gewährleistet ist.

2.2 Simulationsmöglichkeiten basierend auf Simulink

K. Neubauer verwendet Simulink als Simulationswerkzeug [9]. Ein entscheidender Vorteil von Simulink gegenüber Ptolemy II liegt in der umfangreichen Bibliothek, die bereits integriert ist, was den Zeitaufwand für die manuelle Modellierung erheblich reduziert. Der Nachteil besteht jedoch darin, dass die Simulation ausschließlich in der Matlab/Simulink-Umgebung durchgeführt werden muss. Im Gegensatz dazu lässt sich Ptolemy II als Plugin nahtlos in PREEvision integrieren, sodass alle Schritte direkt in PREEvision ausgeführt werden können.

Mit Simscape können Anwender physikalische Modellierungen durchführen. Ein wichtiger Aspekt dabei ist, dass die Ports von Simscape-Blöcken nicht ohne Signalumwandlung direkt mit den Ports herkömmlicher Simulink-Blöcke verbunden werden können. Dies liegt daran, dass zwischen Simscape-Blöcken physikalische Signale fließen, während zwischen Simulink-Blöcken logische Signale übertragen werden. Simscape bietet Bibliotheken zur Modellierung elektrischer und mechanischer Komponenten, wobei die Blöcke aus diesen Bibliotheken ebenfalls nicht ohne entsprechende Signalumwandlung untereinander verbunden werden können. K. Neubauer verwendet die Simscape Electrical Specialized Power Systems (SESPS)-Bibliothek für die elektrische Modellierung. Allerdings lassen sich die SESPS-Blöcke nicht direkt mit anderen Blöcken der Simscape-Electrical-Bibliothek verbinden, was zusätzliche Anpassungen erfordert.

In Simscape gibt es verschiedene Bibliotheken, die ähnliche Komponenten anbieten, jedoch mit unterschiedlichen Zielen und Modellierungseigenschaften. Sowohl in der **Simscape > Foundation Library > Electrical > Electrical Elements** als auch in der **Simscape > Electrical > Passive Bibliothek** finden sich jeweils Blöcke wie Widerstand (Resistor), Kondensator (Capacitor) und Induktor (Inductor).

Die Komponenten aus der Foundation Library repräsentieren idealisierte, mathematische Modelle. So stellt der Kondensator dieser Bibliothek beispielsweise einen idealen Kondensator dar, der ausschließlich die Kapazitätswirkung modelliert. Dagegen bieten die entsprechenden Bauteile aus der Simscape Electrical Library eine realistischere Modellierung, die zusätzliche physikalische Effekte wie Leckströme, Serienwiderstände oder Temperaturabhängigkeit berücksichtigen. Diese erweiterte Funktionalität ermöglicht präzisere Simulationen realer elektrischer Schaltungen.

2.2.1 Abbildung von PREEvision nach Simulink

Leitungssatz Zusammengebündelte Kabel werden in der SESPS-Bibliothek durch den Pi Section Line (PSL)-Block dargestellt. Die Wechselwirkungen zwischen den Kabeln

werden durch drei NxN-Matrizen für Widerstand, Kapazität und Induktivität beschrieben, wobei N die Anzahl der Adern repräsentiert. Diese Matrizen erfassen die elektrischen Kopplungen zwischen den einzelnen Adern und basieren auf ihrer geometrischen Anordnung, dem Querschnitt, dem verwendeten Material, der Isolationsdicke und der Länge der Kabel. Diese Parameter können in Simulink mithilfe des Power Line Parameters Tool (PLPT) berechnet werden.

Der Stecker, der die elektrische Komponente mit den Kabeln verbindet, wird dabei vereinfacht durch einen Ersatzwiderstand modelliert.

In der SESPS-Bibliothek steht kein vordefinierter Block für Sicherungen zur Verfügung. Daher wurde ein eigenes Template für die Modellierung von Sicherungen erstellt.

Sensoren und Aktoren In Simulink ist die Modellierung von Sensoren auf physikalischer Ebene nicht möglich, weshalb diese auf logischer Ebene erfolgt. Sensoren erfassen logische Signale aus der Umgebung und wandeln sie in Ausgangssignale um. Eine gesteuerte Spannungsquelle ermöglicht dabei die Umwandlung der logischen Signale in SESPS-Signale. Bei Aktoren hingegen verläuft der Prozess umgekehrt: Die SESPS-Signale werden zunächst in logische Signale transformiert und anschließend an das Template weitergeleitet.

Das zugewiesene Template definiert den Umwandlungsprozess und stellt nichts anderes als ein parametrierbares Simulink-Subsystem dar. Das von K. Neubauer entwickelte Verfahren basiert auf Templates, wodurch sich viel Zeit bei der manuellen Modellierung einsparen lässt.

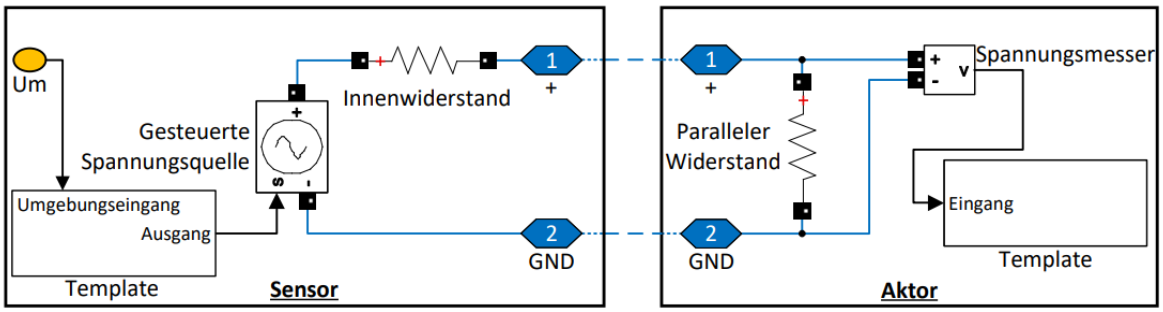

Abbildung 2.5: Prinzip der Signalwandlung in Sensoren und Aktoren. (Quelle: [9, S. 129])

ECUs und Software Zur Speicher-Modellierung in Simulink steht der Data Store Memory (DSM)-Block zur Verfügung. Der Data Store Read (DSR)-Block wird zum Lesen und der Data Store Write (DSW)-Block zum Schreiben von Daten in den DSM verwendet. Ein vom Sensor empfangenes SESPS-Signal muss zunächst in ein logisches Signal umgewandelt werden, bevor es über den DSW-Block in den DSM geschrieben wird. Umgekehrt wird das mithilfe des DSR-Blocks aus dem DSM gelesene Signal zunächst in ein SESPS-Signal konvertiert, bevor es an den Aktor übermittelt wird. Jede Verbindung zwischen SWCs in der PREEvision SA ist ein DSM-Block zugeordnet.

Das SWC wird in Simulink als Function Call Subsystem (FCS) mit einem zugewiesenen Matlab-Skript dargestellt, wobei das Skript die Softwarelogik repräsentiert. Die Ausfüh-

rung des FCS wird durch den Aufruf einer zugewiesenen Simulink-Trigger-Funktion ausgelöst. An den Eingängen der FCS befinden sich die entsprechenden DSR-Blöcke zum Lesen der Daten, während an den Ausgängen die passenden DSW-Blöcke zum Schreiben in den DSM angeschlossen sind.

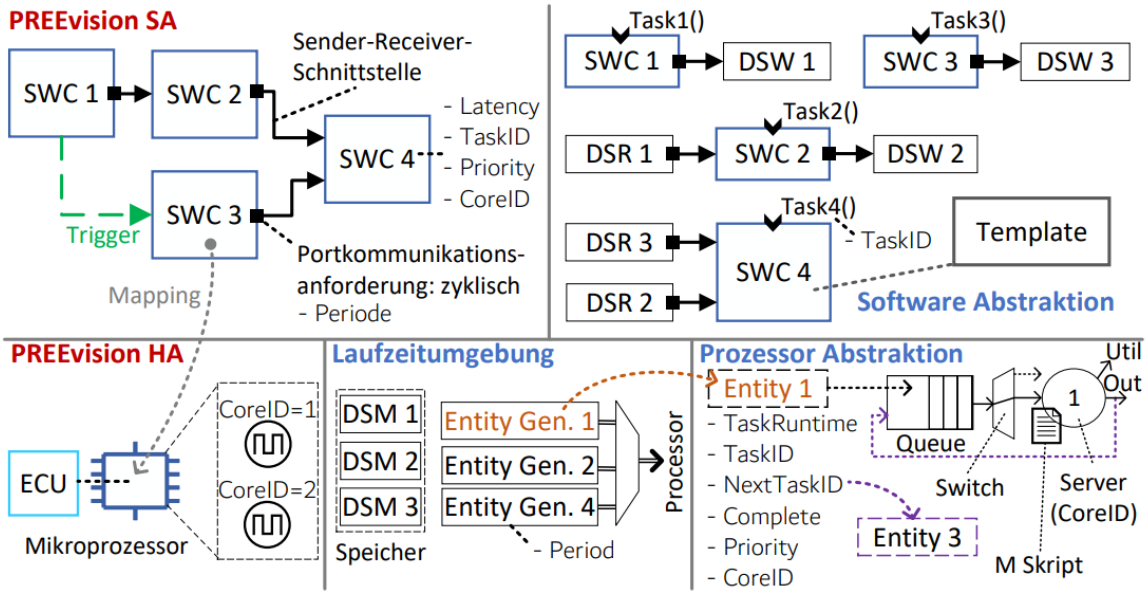

Abbildung 2.6: Modellierung von ECUs und Software in Simulink. (Quelle: [9, S. 136])

Für jede SWC in der PREEvision-SA, die einen Ausgang mit einem zyklischen Sendeverhalten besitzt, wird dann ein Entity-Generator mit der entsprechenden Periode bei der Generierung angelegt. Mit der Periode erzeugen die Entity-Generatoren dann zyklisch Entities, welche hauptsächlich folgende Attribute aufweisen [9]:

- TaskID: Die Verknüpfung des SWC-Subsystems mit seinem individuellen Task erfolgt über diesen Identifikator (ID).
- TaskPriority: Tasks bekommen eine Priorität damit kritische Funktionen schneller auf dem Prozessor ausgeführt werden können.
- TaskRuntime: Abstrakte Laufzeit, die der Prozessor zur Verarbeitung der Komponente benötigen soll.
- CoreID: Ermöglicht die explizite Zuweisung des Tasks zu einem bestimmten Rechenkern.
- NextTaskID: Falls eine SWC eine andere triggert, so erfolgt hier die Referenz auf die nachfolgende SWC.

Entities spielen dabei als Stellvertretern der aufzurudenden FCSs. Die generierte Entities werden weiter an die prioritätenbasierte Warteschlange Entity Queue weitergeleitet. Durch die prioritätenbasierte Entity Queue können Entities mit höherer Priorität zuerst verarbeitet werden. An ihrem Ausgang werden die Entities an die jeweiligen Kerne vom

Typ Entity Server verteilt. In dem Kerne findet die Verarbeitung der Entities statt. Die Ausführung eines Matlab-Skripts wird ausgelöst. Bei der Verarbeitung wird das TaskID-Attribut der Entity ausgelesen und davon abhängig die Trigger-Funktion desjenigen FCSs aufgerufen, welches der SWC entspricht, die mit der Entity verknüpft ist. Als Service Time der Entity Server wird das Attribut TaskRuntime aus der Entity ausgelesen und gesetzt. Die Service Time entspricht der Zeit, in der sich die Entity zur Verarbeitung im Entity Server befindet. Nach einer erfolgreichen Verarbeitung werden anhand NextTaskID sämtliche Attribute der angekommenen Entity mit den Attributen der nachfolgenden Entity überschrieben.

2.2.2 Evaluation

Am Ende wurde ein auf einem Gleichstrommotor basierendes Simulink-Modell simuliert, das aus dem Modell „AC7 - Brushless DC Motor Drive“ der Simscape-Bibliothek stammt. Das Scenario dient hierbei als Stimuli und gibt die Zielgeschwindigkeit (TargetSpeed), die aktuelle Steigung (Grade) sowie das Übersetzungsverhältnis (GearRatio) vor. In der Komponente EgoCar werden mithilfe von Matlab-Funktionen die Zieldrehzahl des Motors (TargetEngineRPM), das Lastmoment (Torque) und die Fahrzeuggeschwindigkeit berechnet.

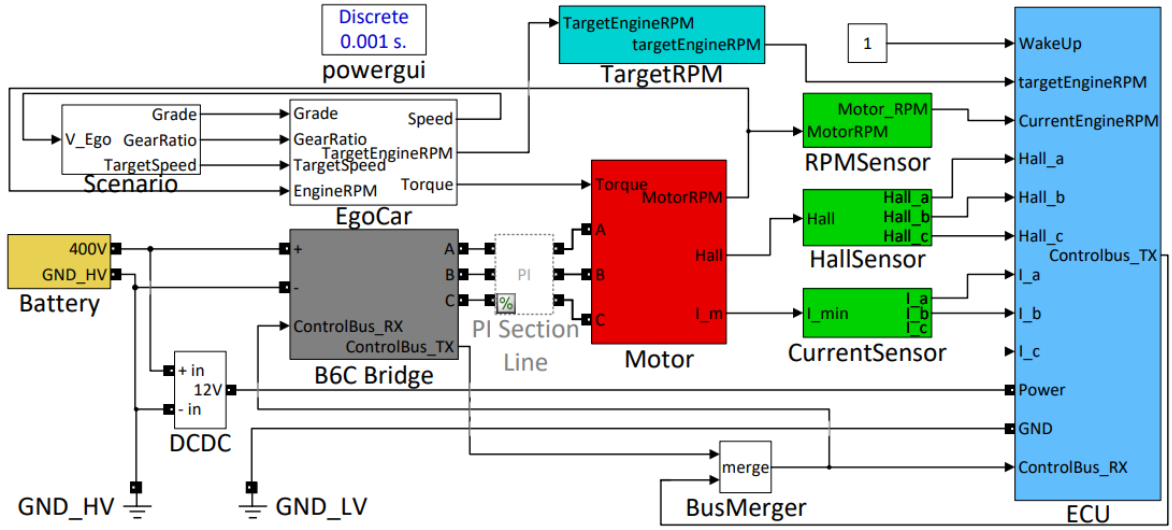

Abbildung 2.7: Generiertes, auf DC-Motor basiertes Simulink-Modell. (Quelle: [9, S. 191])

2.3 Thermische Simulation

In der Praxis ist die Anzahl der Hardware Component (HWC)s, insbesondere der SWCs in PREEvision, deutlich größer als im in Kapitel 2.2.2 simulierten System. Insbesondere die Modellierung bestimmter HWCs und SWCs ist nicht realisierbar in Simulink, wie beispielsweise bei einem Radarsensor und seiner Umgebung.

Denn EDAG besitzt keinen Zugang zu dem Entwicklersplattform für PREEvision, ist das vom K.Neubaruer entwurften Prototyp nicht ins EDAG übertragbar. Denn die Simulation auf elektrische Ebene nicht machbar ist, konzentiert diese Artikel nur auf die termische Simulation. In diesem Kapitel werden zunächst die wesentlichen Elemente der KBL vorgestellt, die als Informationsquelle zur Generierung des Simulink-Modells dienen, um den Temperaturverlauf der Leitungen zu simulieren. Anschließend werden die mathematischen Gleichungen zur Beschreibung des thermischen Modells erläutert.

2.3.1 KBL

Die KBL ist eine in der VDA-Empfehlung 4964 standardisierte Beschreibung von Leitungssträngen im XML-Format. Aktuell wird das XML-Format durch das KBL-Schema in der Version 2.5 definiert [14]. PREEvision unterstützt den Austausch von Kabelbaumdaten im KBL-Datenformat.

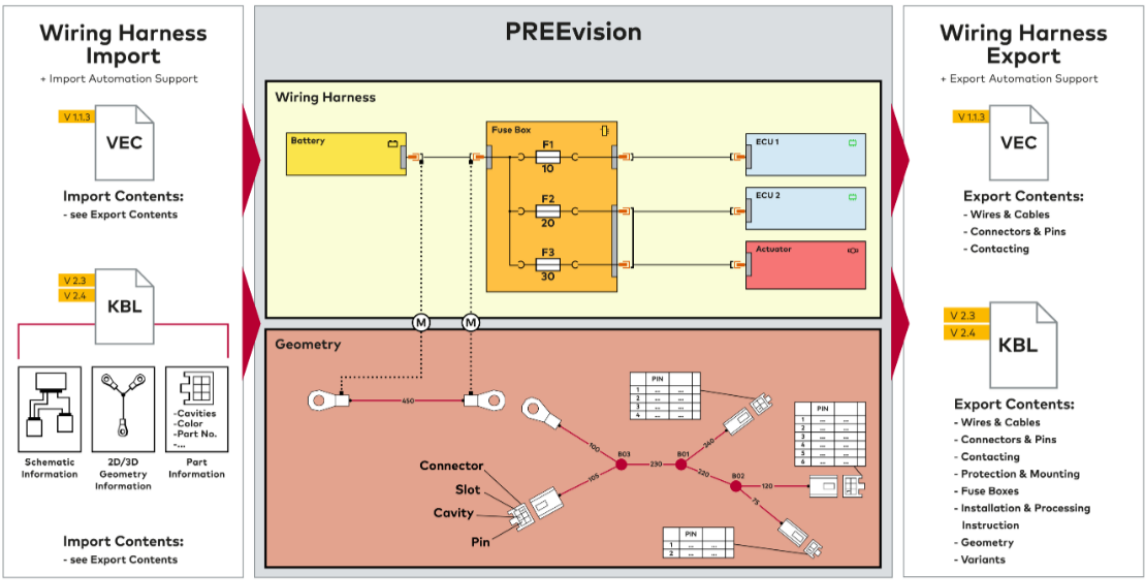

Abbildung 2.8: Ein- und Ausgabeschnittstellen für den Austausch von Kabelbaumdaten. (Quelle: [13])

Die Komplexität der elektrischen Systeme in heutigen Fahrzeugen wächst ständig. Fest organisierte und integrierte unternehmensübergreifende Entwicklungsprozesse sind unerlässlich. Software Component (VEC) ist ein Nachgänger von KBL, ebenfalls im XML-Format. Die VEC-Datenformatspezifikation deckt eine erheblich erweiterte Anzahl von Anwendungsfällen ab und konzentriert sich nicht nur auf einen einzelnen Kabel-

baum im Vergleich zu KBL, sondern auf ein elektrisches System als Ganzes. Die VEC-Datenformatspezifikation unterstützt eine große Vielzahl von Anwendungsfällen zum Datenaustausch entlang des Entwicklungsprozesses des elektrischen Systems. Die folgenden Geschäfts-Anwendungsfälle werden für die Anwendung des VEC-Datenformats überdeckt [13]:

- UC1: Austausch von Komponentendaten
  - UC1.1: Austausch von Komponentendaten mit Fokus auf ihre technischen Merkmale

    Beispiel: Austausch von Komponentendaten zwischen Komponentenlieferant und OEM, um Designingenieuren zu helfen, anwendbare Teile aus technischer Perspektive zu finden und auszuwählen.
  - UC1.2: Austausch von Komponentendaten mit Fokus auf Metadaten

    Beispiel: Um Designingenieuren zu helfen, anwendbare Teile aus organisatorischer Perspektive zu finden und auszuwählen (z. B. unter Berücksichtigung von Genehmigungsinformationen, Vorhandensein bestimmter Nutzungseinschränkungen usw.).
  - UC1.3: Austausch von Komponentendaten mit Fokus auf relationale Daten

    Beispiel: Austausch zwischen OEM und einem Entwicklungspartner zur Definition, welche Hohlräume, Anschlüsse, Hohlraumdichtungen, Hohlraumstecker und Drähte kompatibel bzw. genehmigt sind.
- UC2: Austausch von Verbindungsdaten
  - UC2.1: Austausch von Architekturdaten

    Beispiel: Für nahtlose Rückverfolgbarkeit bis zu einer abstrakten Funktionsebene.
  - UC2.2: Austausch von Schaltplandaten

    Beispiel: Zwischen Tools für Konzeptwerkzeuge (Werkzeuge, die die Evaluierung verschiedener elektrischer Architekturen ermöglichen) und herkömmlichen Schaltplanentwurf-Tools.
  - UC2.3: Austausch von Verdrahtungsdaten

    Beispiel: Unterstützung des Datenaustauschs zwischen OEM und einem Entwicklungspartner, der entweder gebeten werden kann, eine Verdrahtungsdefinition zu vervollständigen oder alternativ eine endgültige Kabelbaumdefinition bereitzustellen.
- UC3: Austausch von Geometriedaten
  - UC3.1: Austausch von Topologie- und dazugehörigen Platzierungsinformationen
  - UC2.2: Austausch von Schaltplandaten
  - UC3.2: Austausch von 2D-Kabelbaumzeichnungen
- UC4: Austausch von Kabelbaumdaten

– UC4.1: Austausch von Stücklistendaten

  Beispiel: Werkzeug zur Kosten- und/oder Gewichtsermittlung mit Daten versorgen.

– UC4.2: Austausch von Verbindungslistendaten

  Beispiel: Werkzeuge zur Testanwendungsentwicklung mit Daten versorgen.

– UC4.3: Austausch vollständiger 150% Kabelbaumdaten

  Beispiel: Unterstützung des Datenaustauschs zwischen OEM und Kabelbaumhersteller im Kontext der Prozesse Kostenkalkulation, Produktionsplanung und Änderungsmanagement.

Trotz der zahlreichen Vorteile des VEC-Datenformats sind die derzeit aus PREEvision exportierten VEC-Daten sehr unvollständig und daher unbrauchbar. Aus diesem Grund können sie momentan nicht als Informationsquelle zur Generierung eines Simulink-Modells verwendet werden.

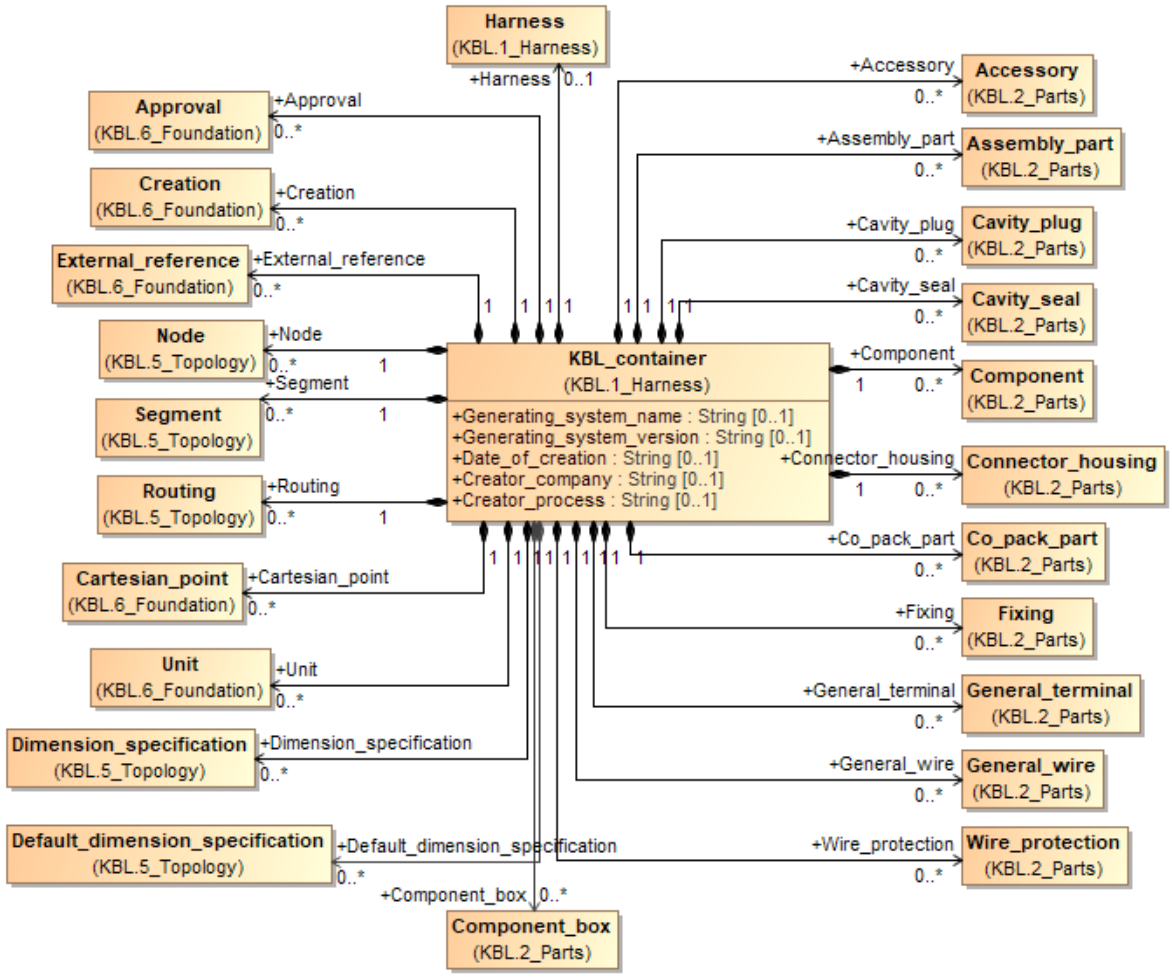

Abbildung 2.9: Klassendiagramm von KBL_Container. (Quelle: [7])

In den folgenden Abschnitten werden die verkabelungsrelevanten Elemente der KBL auf Grundlage von Informationen aus der Website ECAD-Wiki vorgestellt. Auf dieser

Website stehen auch Klassendiagramme zur Verfügung, die die Beziehungen zwischen den KBL-Elementen veranschaulichen.[3] Ein Klick auf die Diagramme führt direkt zu detaillierten Beschreibungen der jeweiligen Elemente.

KBL_container

Das übergeordnete Element aller anderen Elemente.

Ausgehende Beziehungen: Cartesian_point, Connector_housing, Harness, Node, Segment, Routing usw.

Connector_housing

Connector_housing ist ein nicht bestückter Stecker.

Ausgehende Beziehungen: Slot

Eingehende Beziehungen: KBL_container, Connector_occurrence

Ausgehende Beziehungen bezeichnen entweder die in diesem Element enthaltenen Kindelemente oder die über eine ID referenzierten Elemente. Im Umkehrschluss gilt dies für eingehende Beziehungen. Eingehende Beziehungen beziehen sich auf Elemente, die dieses Element als Kindelement enthalten oder die dieses Element über seine ID referenzieren.

```
  </Connector_housing>
  <Connector_housing id="Na5206e3187e73b9ff4527b1XNa5206e3187e73b9ff4527b000">
    <Part_number>DefaultConnector_12pins</Part_number>
    <Company_name/>
    <Version/>
    <Abbreviation/>
    <Description>DefaultConnector_12pins</Description>
    <Slots id="Na5206e3187e73b9ff4527bcXNa5206e3187e73b9ff4527bb00">
      <Id>A</Id>
      <Number_of_cavities>12</Number_of_cavities>
      <Cavities id="Na5206e3187e73b9ff4527bcxNa5206e3187e73b9ff4527c700">
        <Cavity_number>6</Cavity_number>
      </Cavities>
      <Cavities id="Na5206e3187e73b9ff4527bcxNa5206e3187e73b9ff4527d700">
        <Cavity_number>9</Cavity_number>
      </Cavities>
      <Cavities id="Na5206e3187e73b9ff4527bcxNa5206e3187e73b9ff4527e600">
        <Cavity_number>8</Cavity_number>
  </Slots>
</Connector_occurrence>
<Connector_occurrence id="Na66109718b66c108082a5c2XNa66109718b66c108082a50c00">
  <Id>COMP_FT_SPOILER_A</Id>
  <Part>Na5206e3187e73b9ff4527b1XNa5206e3187e73b9ff4527b000</Part>
  <Contact_points id="Na66109718b66c108082b201XNa66109718b66c108082b1fb00">
    <Id></Id>
    <Associated_parts>ID_TermOcc__11923581</Associated_parts>
    <Contacted_cavity>Na66109718b66c108082a5c4xNa66109718b66c108082a57400</Contacted_cavity>
  </Contact_points>
  <Contact_points id="Na66109718b66c108082b2b5XNa66109718b66c108082b2af00">
    <Id></Id>
    <Associated_parts>ID_TermOcc__11923571</Associated_parts>
    <Contacted_cavity>Na66109718b66c108082a5c4xNa66109718b66c108082a58100</Contacted_cavity>
  </Contact_points>
```

Abbildung 2.10: Über eine ID referenzierte Connector_housing.

[3] https://ecad-wiki.prostep.org/specifications/kbl/v25-sr1/classes/

Slots

Slots ist die Gruppierung der Cavity-Objekte eines Steckergehäuses.

Attributen

Number_of_cavities: Entspricht der Anzahl der Hohlräume im geometrischen Modell des Steckverbinders. Diese Zahl ist gleich der Anzahl der Cavity-Objekte unter diesem Slot.

Ausgehende Beziehungen: Cavities

Eingehende Beziehungen: Connector_housing

Cavities

Cavities ist ein definierter Raum in einem Gehäuse zur Platzierung eines elektrischen Kontakts.

Attributen

Cavity_number: Bezeichner der Kontaktkammer.

Eingehende Beziehungen: Slots

Connector_occurrence

Connector_occurrence ist das Vorkommen eines Steckergehäuses (Connector_housing).

Attributen

Part: Referenziert das zugeordnete Connector_housing über dessen ID.

Ausgehende Beziehungen: Contact_point, Slots, Connector_housing

Eingehende Beziehungen: Harness

Contact_point

Definiert die Stellen, an denen elektrische Verbindungen hergestellt werden.

Eingehende Beziehungen: Extremity, Connector_occurrence

General_wire

Ausgehende Beziehungen: Mass_information, Cross_section_area, Outside_diameter, Cover_colour

Eingehende Beziehungen: KBL_container

General_wire_occurrence

General_wire_occurrence ist das Vorkommen eines Allgemeinkabels (General_wire).

Attributen

Part: Referenziert das zugeordnete General_wire über dessen ID.

Ausgehende Beziehungen: Length_information, General_wire

Eingehende Beziehungen: Harness

Connection

Connection ist ein Mechanismus, um die elektrische Verbindung zwischen zwei oder mehr Kontaktpunkten festzulegen. Eine Verbindung beschreibt die Konnektivität eines Kabel-Vorkommens.

Attributen

Signal_name: Gibt logische Informationen zu einer Verbindung an. Beispiel: ein Datenpaket auf einem Bus oder eine analoge Spannung (hoch/niedrig, wellenförmig) auf einem Kabel.

Wire: Referenziert das zugeordnete General_wire_occurrence über dessen ID.

Ausgehende Beziehungen: Extremity , General_wire_occurrence

Eingehende Beziehungen: Harness

Extremity

Gibt den Kontaktpunkt an.

Attributen

Contact_point: Referenziert ein Contact_points in Connector_occurrence über dessen ID.

Position_on_wire: Beschreibt die Stelle, an der die Kontaktierung erfolgt. Ein Wert von 0,0 bezeichnet den Anfang des Kabels, ein Wert von 1,0 das Ende.

Eingehende Beziehungen: Connection

Zusätzlich gibt es weitere Elemente, die das geometrische Routing der Leitungen beschreiben, wie zum Beispiel **Cartesian_point**, **Node**, **Segment** und **Routing**. Diese werden hier jedoch nicht näher erläutert.

2.3.2 Thermisches Modell für konventionelle Leitungen

M. Diebig hat die thermischen Eigenschaften von konventionellen Leitungen modelliert [5, Kap. 3]. In Fahrzeugen kommen bis zu 60 V konventionelle Fahrzeugleitungen zum Einsatz; bei Spannungen über 60 V werden geschirmte Leitungen verwendet.

Der durch die Leitung fließende Strom führt zu Verlusten in Form von Wärme. Diese Wärmeabgabe lässt sich über drei Mechanismen beschreiben: Wärmeleitung, Wärmestrahlung P_s und Konvektion P_k. Wärmeleitung beschreibt den Energietransport in Festkörpern, der auf Temperaturunterschieden basiert. Dabei sind Temperaturgradienten im Material die treibende Kraft. Die Temperatur T ändert sich somit in Abhängigkeit vom Ort und der Zeit.

$$\mathbf{q} = -\lambda \nabla T \tag{1}$$

Hierbei steht $\mathbf{q}$ für die Wärmestromdichte und λ für die materialabhängige Wärmeleitfähigkeit.

Energie kann auch durch strömende Fluide, sei es in Form von Flüssigkeiten oder Gasen, transportiert werden. Der Begriff Konvektion beschreibt den massegebundenen Energietransport. Der konvektive Wärmeübergang P_k wird in Abhängigkeit von der wärmeübertragenden Fläche als Produkt aus dem Umfang u_0, der Länge l und der Temperaturdifferenz zwischen dem Festkörper (Isolierung) und dem Fluid (Luft) folgendermaßen berechnet:

$$P_k = \alpha_k u_0 l \cdot (T_o - T_u) \tag{2}$$

Hierbei steht α_k für den konvektiven Wärmeübergangskoeffizienten. Eine exakte Berechnung des Wärmeübergangskoeffizienten ist meist nicht möglich und muss häufig auf Basis experimenteller Daten bestimmt werden.

Wärmestrahlung beschreibt die Energieabgabe eines Körpers durch elektromagnetische Wellen an die Umgebung, ohne einen materiellen Träger. Daher ist dieser Prozess auch im Vakuum möglich. Um die Leistungsgleichung der Strahlung mit der für die Konvektion vergleichbar zu machen, wird sie in eine äquivalente Form zur Gleichung (2) gebracht:

$$P_S(T_o, T_u) = \alpha_s(T_o, T_u) u_0 l \cdot (T_o - T_u) \tag{3}$$

Hierbei hängt der Wärmeübergangskoeffizient für die Strahlung α_s von den Temperaturen der Oberfläche und der Umgebung ab. Für die Berechnung kann er jedoch näherungsweise als konstant betrachtet werden, um die Komplexität zu reduzieren. Der Gesamtwärmeübergangskoeffizient α_{ges} stellt die Summe der beiden Koeffizienten dar:

$$\alpha_{\text{ges}} = \alpha_S + \alpha_k \tag{4}$$

In Zylinderkoordinaten und mit inneren Wärmequellen lautet die Fourier'sche Wärmeleitungsgleichung:

$$\frac{\partial T}{\partial t} = \frac{\lambda}{c\rho}\left(\frac{\partial^2 T}{\partial r^2} + \frac{1}{r}\frac{\partial T}{\partial r} + \frac{1}{r^2}\frac{\partial^2 T}{\partial \phi^2} + \frac{\partial^2 T}{\partial z^2}\right) + \frac{\dot{W}_0}{c\rho} \tag{5}$$

Eine Möglichkeit, den Temperaturverlauf zu berechnen, besteht darin, die Fourier'sche Wärmeleitungsgleichung direkt zu lösen. Solche partiellen Differentialgleichungen können mithilfe von Anfangs- und Randbedingungen in Matlab gelöst werden. Da Simulink jedoch nur in der Lage ist, gewöhnliche Differentialgleichungen zu lösen, wird dieser Lösungsansatz hier nicht weiter betrachtet.

Ein wärmetechnisches Problem lässt sich auch durch die Nachbildung elektrischer Vorgänge lösen, da beide Prozesse im Wesentlichen auf dieselbe Struktur von Differentialgleichungen zurückzuführen sind. In dieser Analogie entspricht die elektrische Spannung der Temperatur, während der elektrische Strom die Wärmeleistung repräsentiert. Die Beziehung zwischen diesen Größen lässt sich durch folgende Gleichung beschreiben:

$$C\frac{dV_1}{dt} = \frac{V_1 - V_2}{R} + I_0 \tag{6}$$

Bei der stationären Lösung werden nur der thermische Widerstand der Isolierung und die Wärmeabgabe an die Umgebung berücksichtigt, und die Wärmekapazität spielen keine Rolle. Um den Wärmewiderstand der Isolierung zu erhalten, wird eine nichtleitende zylindrische Schicht betrachtet. Mit dem inneren Radius r_1 und dem äußeren Radius r_2 kann die Temperaturverteilung in der Schicht nach [[1] folgendemaße berechnet werden:

$$T(r) = \frac{T_1 - T_2}{\ln(\frac{r_1}{r_2})}\ln\left(\frac{r}{r_2}\right) + T_2 \tag{7}$$

Anhander der Gleichung die Wärmestrom gleich die Ableitung der Temperatur mal die Wärmeleitfähigkeit des betrachteten Materials und die Fläche:

$$P = 2\pi\lambda l \cdot \frac{1}{\ln(\frac{r_1}{r_2})} \cdot (T_1 - T_2) \tag{8}$$

Der thermische Widerstand der Isolierung kann wie folgt berechnet werden:

$$R_{\lambda_i} = \frac{1}{2\pi\lambda_i l} \cdot \ln\left(\frac{r_i}{r_l}\right) \tag{9}$$

wobei r_i den Radius der Isolierung und r_l den Radius des Innenleiters bezeichnet.

Der thermische Widerstand R_{λ_l} für den Leiter kann aufgrund der hohen Wärmeleitfähigkeit von Materialien wie Kupfer oder Aluminium vernachlässigt werden. Der thermische Widerstand für den Wärmeübergangskoeffizienten, welcher sowohl Strahlung als auch Konvektion berücksichtigt, wird durch die Gleichungen (2) und (3) beschrieben. Der Widerstand lautet:

$$R_\alpha = \frac{1}{2\pi r l \alpha_{\text{ges}}} \tag{10}$$

Die stationäre Leitertemperatur ergibt sich schließlich zu:

$$T_l = P \cdot (R_\alpha + R_{\lambda_i}) + T_u \tag{11}$$

Hierbei steht P für die Verlustleistung, T_u für die Umgebungstemperatur.

Für das transiente Verhalten müssen die thermischen Kapazitäten des Leiters und der Isolierung berücksichtigt werden. Die thermische Kapazität C_{th} wird durch die folgende Gleichung definiert:

$$C_{\text{th}} = c \cdot \rho \cdot A \cdot l \tag{12}$$

wobei c die spezifische Wärmekapazität und ρ die Dichte des Materials darstellt.

Der zeitliche Verlauf der Temperatur kann durch die folgende Differenzialgleichung beschrieben werden:

$$\frac{dT_l(t)}{dt} = \frac{1}{C_{\text{con}} + C_{\text{ins}}} \cdot \left(P - \frac{T_l(t) - T_u}{R_{\lambda i} + R_{\alpha \text{ges}}} \right) \tag{13}$$

Die gesamte thermische Kapazität setzt sich aus der Summe der thermischen Kapazitäten der Isolierung und des Leiters zusammen.

Die durch den Strom verursachte Verlustleistung geht zum einen in die Umgebungsluft verloren, während der verbleibende Teil in der Leitung gespeichert wird und zu einer Temperaturerhöhung führt. Der Temperaturverlauf ist unabhängig von der Länge der Leitung, da sich diese auf beiden Seiten der Gleichung ausgleicht.

2.3.3 Simulation

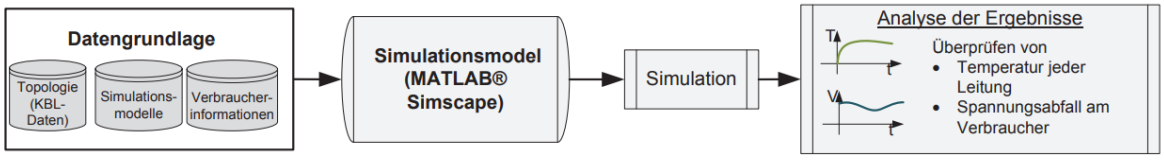

Abbildung 2.11: Prozessablauf zur Simulation von Bordnetzen. (Quelle: [5, S. 64])

Unter Verwendung der KBL als Informationsquelle wird mithilfe von Simscape eine Netzliste generiert [5]. Wie dies genau implementiert ist, wird im ursprünglichen Artikel nicht näher beschrieben. Alle Verbraucher im Bordnetz werden dabei über ihren Stromverlauf abstrahiert. Die Stecker in der Netzliste werden mit den entsprechenden Stromverläufen der zugehörigen Verbraucher verknüpft. Das Ergebnis der Simulation liefert den Temperaturverlauf der einzelnen Leitungen sowie deren Spannungsabfall.

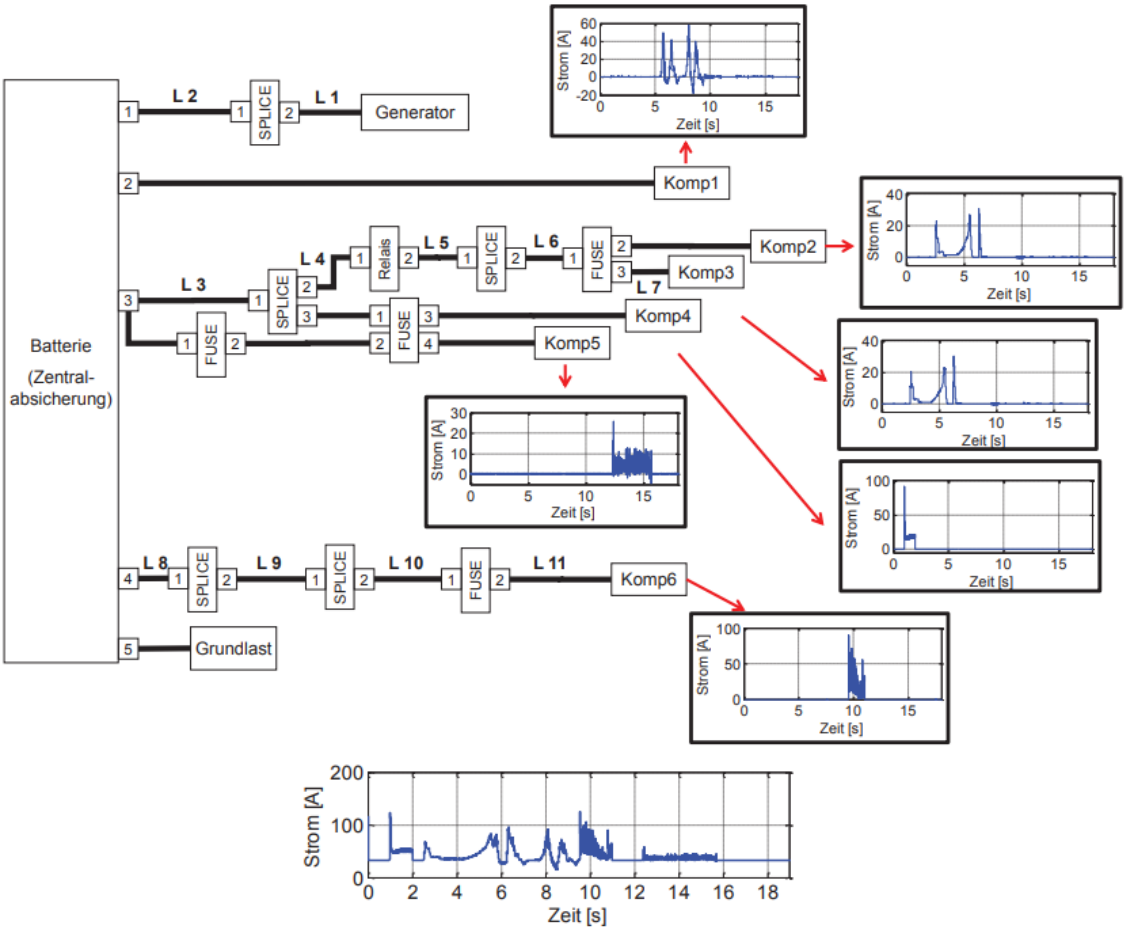

Abbildung 2.12: Simulink-Modell zur Simulation des Temperaturverlaufs und Spannungsabfalls der Leitungen. (Quelle: [5, S. 69])

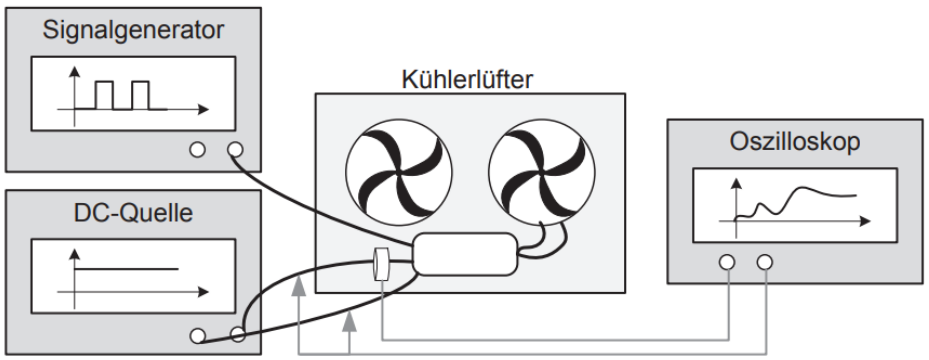

Abbildung 2.13: Messaufbau zur Messung des Stromverlaufs. (Quelle: [5, S. 66])

3 Implementierung

Das Ziel dieses Projekts ist die Entwicklung eines Verfahrens, das auf Basis von Informationen aus XML-Daten automatisch Simulink-Modelle generiert. Bereits existierende Java-Bibliotheken ermöglichen die Übertragung von XML-Daten in Java-Objekte. Der innovative Beitrag dieses Projekts liegt in der Implementierung eines Java-Frameworks, das die Erstellung von Simulink-Modellen auf objektorientierte Weise erlaubt. Der Quellcode ist auf GitHub verfügbar.[4]

3.1 Java-Framework zur objektorientierten Erstellung von Simulink-Modellen

Die Erstellung eines Simulink-Modells erfolgt nicht nur über die grafische Benutzeroberfläche, sondern auch programmatisch mithilfe von MATLAB-Skripten.[5] MATLAB bietet dazu eine Schnittstelle namens Matlab Engine API for Java (ME-API-J), die eine Integration mit Java ermöglicht.[6] Über diese Schnittstelle können MATLAB-Skripte von Java an MATLAB übergeben werden. Das Java-Programm stellt dabei zunächst eine Verbindung zu MATLAB her, woraufhin die Skripte an MATLAB übertragen und dort ausgeführt werden.

Das folgende Java-Programm nutzt die MATLAB Engine API für Java, um ein Simulink-Modell mit dem gegebenen Namen zu erstellen und zu konfigurieren.

```
// Start the MATLAB Engine
MatlabEngine matlab = MatlabEngine.startMatlab();

// Create the Simulink model
matlab.eval("model = new_system('simpleCircuit', 'Model')");

// Add blocks to the model
matlab.eval("add_block('fl_lib/Electrical/Electrical Sources/DC Voltage Source', '
    simpleCircuit/DC')");
matlab.eval("add_block('fl_lib/Electrical/Electrical Elements/Resistor', '
    simpleCircuit/r1')");
matlab.eval("add_block('fl_lib/Electrical/Electrical Elements/Electrical Reference', '
    simpleCircuit/ref1')");
matlab.eval("add_block('nesl_utility/Solver Configuration', 'simpleCircuit/solver1')");


// Set parameters for the blocks
matlab.eval("set_param('simpleCircuit/r1', 'R', '50')");
matlab.eval("set_param('simpleCircuit/DC', 'v0', '10')");

// Connect the blocks in the model
matlab.eval("add_line('simpleCircuit', 'DC/LConn1', 'r1/LConn1', 'autorouting', 'on')"
    );
matlab.eval("add_line('simpleCircuit', 'r1/RConn1', 'ref1/LConn1', 'autorouting', 'on
    ')");
```

[4] https://github.com/songxinrandiyi/preevisiontosimulink

[5] https://de.mathworks.com/help/simulink/programmatic-modeling.html

[6] https://de.mathworks.com/help/matlab/matlab_external/get-started-with-matlab-engine-api-for-java
html

```
matlab.eval("add_line('simpleCircuit', 'DC/RConn1', 'ref1/LConn1', 'autorouting', 'on
    ')");
matlab.eval("add_line('simpleCircuit', 'solver1/RConn1', 'ref1/LConn1', 'autorouting',
     'on')");

// Save and close the model
matlab.eval("save_system('simpleCircuit', 'simpleCircuit.slx')");
matlab.close();
```

- Das Programm beginnt, indem es die MATLAB-Engine startet. Dies ermöglicht die Kommunikation zwischen dem Java-Programm und MATLAB.
- Ein neues Simulink-Modell mit dem Namen **simpleCircuit** wird erstellt. Die Methode new_system erzeugt ein leeres Modell, das anschließend mit Blöcken gefüllt werden kann.
- Es werden verschiedene Blöcke zum Modell hinzugefügt. Die Parameter der hinzugefügten Blöcke werden festgelegt. Die Blöcke werden miteinander verbunden, um elektrische Verbindungen herzustellen.
- Schließlich wird das Modell unter dem Namen **simpleCircuit.slx** gespeichert und die MATLAB-Engine wird geschlossen.

Der Pfad **fl_lib/Electrical/Electrical Sources/DC Voltage Source** verweist auf einen Block in der Simulink-Bibliothek, der eine Gleichstromspannungsquelle darstellt. Um den Pfad eines bestimmten Blocks in Simulink herauszufinden, kann man einfach den Bibliotheksbrowser öffnen, den gewünschten Block auswählen und die Maus darüber halten – der vollständige Pfad wird dann direkt angezeigt.

simpleCircuit/DC gibt die Position und den Namen des eingefügten Blocks im Simulink-Modell an. **simpleCircuit** ist der Name des Modells, in das der Block eingefügt wird, während **DC** der Name ist, der dem Block im Modell zugewiesen wird. Dieser Name wird verwendet, um später auf den Block innerhalb des Modells zuzugreifen.

Die Funktion **set_param** in MATLAB dient dazu, Parameter eines Simulink-Blocks zu ändern oder zu konfigurieren. Der Block wird dabei über seinen Pfad im Modell referenziert. Der Parameter **v0** steht in diesem Fall für die Ausgangsspannung des **DC Voltage Source**-Blocks, und der anschließende Wert gibt die neue Spannung an, die dem Parameter **v0** zugewiesen wird.

add_line ist eine MATLAB-Funktion, die in Simulink verwendet wird, um zwei Blöcke im Modell durch eine Signalverbindung miteinander zu verknüpfen. **simpleCircuit** bezeichnet das Simulink-Modell, in dem diese Verbindungen hergestellt werden. **LConn1** und **RConn1** sind spezifische Anschlüsse der Blöcke, die für diese Verbindungen genutzt werden.

Um Informationen über die Anschlüsse und Parameter eines Blocks zu erhalten, werden folgende MATLAB-Skripte verwendet: **getSimulinkBlockHandle** ist eine MATLAB-Funktion, die den Handle (eine Art Referenz) des angegebenen Blocks im Simulink-Modell zurückgibt. Der Befehl **disp(get_param(h, 'PortHandles'))** ruft die Informationen über die Portanschlüsse des **DC Voltage Source**-Blocks ab und gibt diese in

der MATLAB-Konsole aus, sodass der Benutzer sehen kann, welche Anschlüsse zur Verfügung stehen. Zudem ermöglicht **get_param(h, 'ObjectParameters')** das Abrufen der Parameter, die für diesen Blocktyp konfigurierbar sind.

```
path = 'simpleCircuit/DC';
h = getSimulinkBlockHandle(path,true);
disp(get_param(h,'PortHandles'));
disp(get_param(h,'ObjectParameters'));
```

Da der **DC Voltage Source**-Block zu Simscape gehört, besitzt er keine **Inport**- oder **Outport**-Anschlüsse. Diese Anschlüsse dienen in Simulink der Übertragung logischer Signale zwischen Blöcken. In Simscape hingegen steht die physikalische Modellierung im Vordergrund, weshalb hier die Anschlüsse **LConn** und **RConn** zur Übertragung physikalischer Signale zwischen Simscape-Blöcken verwendet werden. Die Zahl dahinter gibt die Position des jeweiligen Ports im Modell an. Ein **Gain**-Block in Simulink hingegen besitzt keine **LConn**- und **RConn**-Anschlüsse, sondern verwendet stattdessen **Inport** und **Outport**, um logische Signale innerhalb eines Simulink-Modells zu übertragen.

Abbildung 3.1: Portanschlüsse und Parameter des DC Voltage Source-Blocks

Das Vertauschen der Positionen von **DC/LConn1** und **r1/LConn1** hat keine Auswirkungen auf die Verbindung. Dies gilt jedoch nicht für Verbindungen zwischen Simulink-Blöcken. Im folgenden Code wird beispielsweise eine Verbindung zwischen dem Ausgang des **Product**-Blocks und dem zweiten Eingang des **Sum**-Blocks hergestellt. Würde man die beiden Anschlüsse vertauschen, würde dies bedeuten, dass der zweite Ausgang des **Sum**-Blocks mit dem Eingang des **Product**-Blocks verbunden wird, was zu einem Fehler führen würde, da der **Sum**-Block nur einen Ausgang besitzt.

```
%add_line('simpleCircuit', 'DC/LConn1', 'r1/LConn1', 'autorouting', 'on')
%is the same as
%add_line('simpleCircuit', 'r1/LConn1', 'DC/LConn1', 'autorouting', 'on')
new_system('MyModel');
add_block('simulink/Math Operations/Sum', 'MyModel/Sum');
add_block('simulink/Math Operations/Product', 'MyModel/Product');
add_line('MyModel', 'Product/1', 'Sum/2', 'autorouting', 'on');
```

Wenn ein Block oder eine Verbindung in einem Subsystem erstellt werden soll, ist es wichtig, den vollständigen Pfad des Blocks anzugeben. Dieser Pfad muss sowohl den Namen des Modells als auch den des Subsystems enthalten. Da Subsysteme hierarchisch geschachtelt werden können, müssen alle Subsystemnamen in der richtigen Reihenfolge angegeben werden. Obwohl die Tiefe eines Subsystems theoretisch unbegrenzt ist, sind sehr tief geschachtelte Subsysteme in der Praxis selten sinnvoll.

```
add_block('simulink/Commonly Used Blocks/Subsystem', 'MyModel/MySubsystem');
add_block('simulink/Math Operations/Gain', 'MyModel/MySubsystem/Gain1');
add_block('simulink/Commonly Used Blocks/In1', 'MyModel/MySubsystem/Input1');
add_line('MyModel/MySubsystem', 'Input1/1', 'Gain1/1', 'autorouting', 'on');
```

Die Skripte zur Erstellung von Simulink-Modellen festzuschreiben ist jedoch nicht besonders übersichtlich und anfällig für Fehler. Daher implementiert K. Neubauer in [9, S. 147-156] ein Framework als Java-Stellvertreter für Simulink-Modelle. Basierend auf diesen Überlegungen wird ein eigenes Framework gemäß den Anforderungen dieses Projekts entwickelt (siehe Abbildung 3.2), um Simulink-Modell in der objekorientierten Weise aufzubauen.

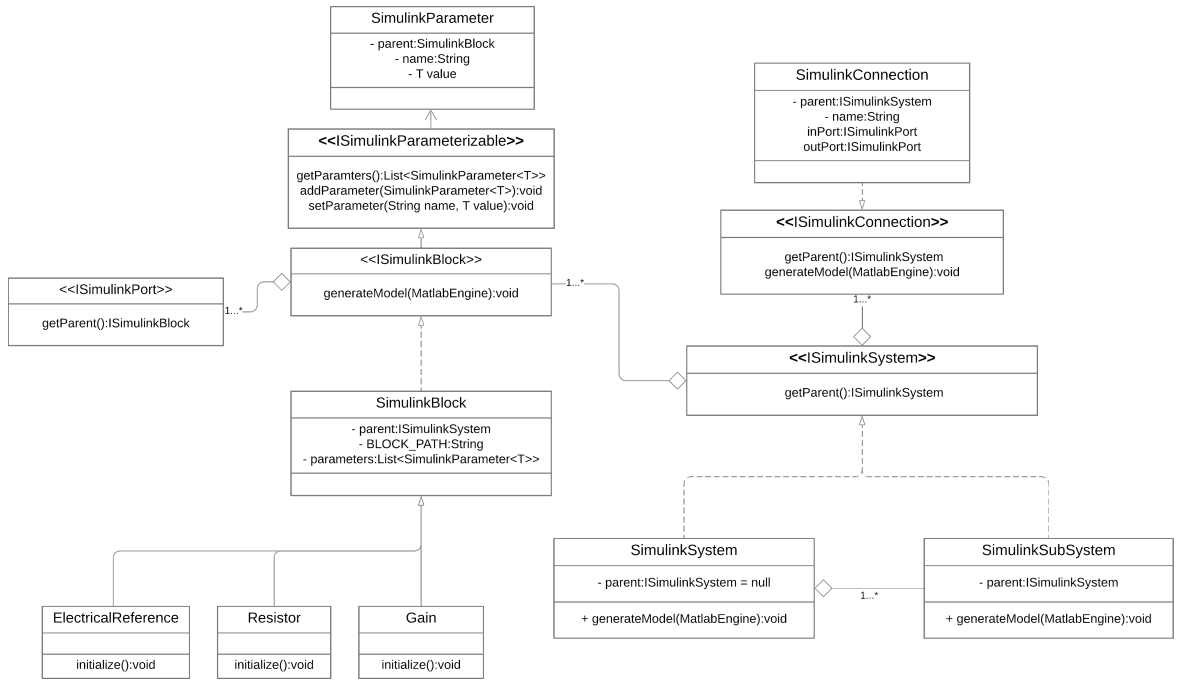

Abbildung 3.2: Klassendiagramm von Java-Stellvertretern für Simulink

Mit diesem Framework muss der Programmierer keine Skripte mehr manuell zum Erstellen eines Blocks schreiben. Stattdessen werden solche Skripte flexibel in der Methode **generateModel** der Klasse **ISimulinkBlock** generiert. Der Konstruktor von **ISimulinkBlock** erwartet zwei Argumente: **parent**, das übergeordnete System oder Subsystem (vom Typ **ISimulinkSystem**), dem der Block angehört, sowie den Namen des Simulink-Blocks. Zusätzlich werden die Listen **inPorts**, **outPorts** und **parameters** verwendet, um die Ein- und Ausgangsports sowie die Parameter des Blocks zu verwalten.

Die Methode **generateModel** akzeptiert einen Parameter vom Typ **MatlabEngine**, welcher die Schnittstelle zur Ausführung von MATLAB-Befehlen aus Java heraus darstellt. Über diese Instanz erfolgt die Kommunikation zwischen Java und der MATLAB-Umgebung. Der Ablauf der Methode ist wie folgt:

- Zunächst wird mithilfe der Methode **generateCombinedPath** der vollständige Pfad des Blocks im Simulink-Modell generiert, basierend auf der Hierarchie der übergeordneten Subsysteme.
- Anschließend wird der MATLAB-Befehl **add_block** ausgeführt, um den neuen Block in das Simulink-Modell einzufügen. Der Blocktyp wird durch den Pfad **BLOCK_PATH** angegeben, während **combinedPath** den genauen Ort im Modell definiert, an dem der Block erstellt wird.
- Danach werden alle Parameter des Simulink-Blocks durchlaufen und deren Werte mit dem Befehl **set_param** in MATLAB gesetzt.

```
public SimulinkBlock(ISimulinkSystem parent, String name) {
    this.name = name;
    this.parent = parent;
    this.inPorts = new ArrayList<>();
    this.outPorts = new ArrayList<>();
    this.parameters = new ArrayList<>();
    this.initialize();
}

@Override
public void generateModel(MatlabEngine matlab) {
    try {
        String combinedPath = generateCombinedPath();
        matlab.eval("add_block('" + BLOCK_PATH + "', '" + combinedPath + "')");

        for (SimulinkParameter<?> param : getParameters()) {
            if (param.getValue() != null) {
                matlab.eval("set_param('" + combinedPath + "', '"
                        + param.getName() + "', '"
                        + param.getValue().toString() + "')");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Override
public String generateCombinedPath() {
    StringBuilder pathBuilder = new StringBuilder(name);
    ISimulinkSystem currentParent = parent;
    while (currentParent != null) {
        pathBuilder.insert(0, currentParent.getName() + "/");
        currentParent = currentParent.getParent();
    }
    return pathBuilder.toString();
}
```

Die folgende Klasse, die von **SimulinkBlock** abgeleitet ist, repräsentiert konkrete Blöcke in Simulink. In dieser Klasse werden der Pfad zu dem Block, die Eingangs- und Ausgangsports sowie die Parameter des Blocks festgelegt. Dabei sind **LConn**, **RConn** und **SimulinkPort** allesamt Klassen, die von **ISimulinkPort** abgeleitet sind.

```
//class DCVoltageSource extends SimulinkBlock
@Override
public void initialize() {
    this.BLOCK_PATH = "fl_lib/Electrical/Electrical Sources/DC Voltage Source";

    this.inPorts.add(new LConn(1, this));
    this.outPorts.add(new RConn(1, this));
    this.parameters.add(new SimulinkParameter<Double>("v0", this));
}

//class Gain extends SimulinkBlock
@Override
public void initialize() {
    this.BLOCK_PATH = "simulink/Math Operations/Gain";

    this.inPorts.add(new SimulinkPort(1, this));
    this.outPorts.add(new SimulinkPort(1, this));
    this.parameters.add(new SimulinkParameter<Double>("Gain", this));
}
```

Die MATLAB-Befehle um zwei Blöck zu verbinden ist ebenfalls flexibel konstrukiert. **outPort** repräsentiert den Ausgangsport, von dem die Verbindung ausgeht, während **inPort** den Eingangsport darstellt, zu dem die Verbindung führt. **parent** ist das übergeordnete System oder Subsystem, in dem sich die Verbindung befindet.

In der Methode **generateModel** wird zunächst der Pfad des Ausgangsports generiert, der sich aus dem Namen des übergeordneten Blocks und dem Namen des Ausgangsports zusammensetzt. Analog dazu wird der Pfad des Eingangsports aufgebaut. Anschließend wird der vollständige Pfad des übergeordneten Systems erstellt, indem die Namen aller übergeordneten Systeme hinzugefügt werden, bis kein weiteres übergeordnetes System mehr vorhanden ist.

Am Ende dieser Schritte wird der MATLAB-Befehl **add_line** ausgeführt, um eine Linie zwischen dem Quell- und dem Zielport zu zeichnen. Dadurch wird sichergestellt, dass die Verbindung korrekt im Simulink-Modell dargestellt wird.

```
public class SimulinkConnection implements ISimulinkConnection {
  private ISimulinkPort inPort;
  private ISimulinkPort outPort;
  private ISimulinkSystem parent;

  public SimulinkConnection(ISimulinkPort outPort, ISimulinkPort inPort,
    ISimulinkSystem parent) {
    this.inPort = inPort;
    this.outPort = outPort;
    this.parent = parent;
  }

  @Override
  public ISimulinkSystem getParent() {
```

```
    return parent;
  }

  @Override
  public void generateModel(MatlabEngine matlab) {
    // Implementation for generating the connection
    String sourceBlockPath = outPort.getParent().getName() + "/"
        + outPort.getName();
    String destinationBlockPath = inPort.getParent().getName() + "/"
        + inPort.getName();

    String parentPath = parent.getName();
    ISimulinkSystem currentParent = parent.getParent();
    while (currentParent != null) {
      parentPath = currentParent.getName() + "/" + parentPath;
      currentParent = currentParent.getParent();
    }

    try {
      matlab.eval("add_line('" + parentPath + "', '" + sourceBlockPath + "', '"
          + destinationBlockPath + "', 'autorouting', 'on')");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Die Einführung von Subsystemen (**SimulinkSubsystem**) erhöht die Übersichtlichkeit des Modells erheblich. Ein **SimulinkSubsystem** fasst eine Gruppe von Blöcken zu einem einzigen Block zusammen, wodurch das Modell einfacher und übersichtlicher wird. Ein Subsystem kann auch andere Subsysteme enthalten und als übergeordnetes Subsystem fungieren. Obwohl Subsysteme theoretisch unendlich tief verschachtelt werden können, ist dies in der Praxis selten sinnvoll.

Das Einfügen von Subsystemen hilft, komplexe Modelle in kleinere, handhabbare Einheiten zu zerlegen, was die Modellentwicklung, -wartung und -verständlichkeit erleichtert. Durch die Strukturierung des Modells in Subsysteme können Entwickler spezifische Funktionalitäten isolieren und sich auf einzelne Aspekte des Modells konzentrieren, ohne von der Gesamtheit des Systems überwältigt zu werden.

Blöcke vom Typ **Inport** und **Outport** repräsentieren die Ein- und Ausgänge in Simulink-Subsystemen und dienen zur Übertragung logischer Signale zwischen Simulink-Blöcken. Jedes Simulink-Subsystem wird standardmäßig mit einem Paar verbundener **Inport**- und **Outport**-Blöcke initialisiert. Ein Block vom Typ **Connection Port** steht für Anschlüsse in einem Simulink-Subsystem, die physikalische Signale von Simscape-Blöcken übertragen. Ist dieser Block nach rechts ausgerichtet, fungiert er als Eingang; bei einer Ausrichtung nach links fungiert er als Ausgang.

In der Methode **generateModel** der Klasse **SimulinkSubsystem** wird zunächst der Code zum Erstellen des Subsystems generiert, da ein Subsystem in Simulink ebenfalls als Block behandelt wird. Danach werden die bei der Initialisierung erstellten **Inport**- und **Outport**-Blöcke sowie deren Verbindung entfernt. Anschließend werden die Listen durchlaufen,

um die zugehörigen Blöcke (**ISimulinkBlock**), Verbindungen (**ISimulinkConnection**), weiteren Subsysteme und Anschlüsse zu generieren, indem die jeweilige **generateModel**-Methode aufgerufen wird.

```
public class SimulinkSubsystem implements ISimulinkSystem {
  private static final String BLOCK_PATH = "simulink/Ports & Subsystems/Subsystem";
  private List<LConnection> inConnections = new ArrayList<>();
  private List<RConnection> outConnections = new ArrayList<>();
  private List<InPort> inPorts = new ArrayList<>();
  private List<OutPort> outPorts = new ArrayList<>();
  private List<ISimulinkBlock> blockList = new ArrayList<>();
  private List<ISimulinkConnection> connectionList = new ArrayList<>();
  private List<SimulinkSubsystem> subsystemList = new ArrayList<>();

  public void generateModel(MatlabEngine matlab) {
    try {
      String combinedPath = generateCombinedPath(parent, name);

      matlab.eval("add_block('" + BLOCK_PATH + "', '" + combinedPath + "')");
      matlab.eval("delete_line('" + combinedPath + "','In1/1','Out1/1')");
      matlab.eval("delete_block('" + combinedPath + "/In1')");
      matlab.eval("delete_block('" + combinedPath + "/Out1')");

      for (SimulinkSubsystem subsystem : subsystemList) {
        subsystem.generateModel(matlab);
      }
      for (ISimulinkBlock block : blockList) {
        block.generateModel(matlab);
      }
      // Additional for-loop for the remaining list
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Der Singleton **SimulinkSystem** entspricht dem zu generierenden Simulink-Modell. Zu **SimulinkSystem** sind eine Menge von Blöcken (**ISimulinkBlock**) sowie Verbindungen zwischen Blöcken (**ISimulinkRelation**) und Subsystemen (**SimulinkSubsystem**) zugeordnet. Die Vorgehensweise um das gesamte Simulink-Modell zu konstruieren is ähnlich. Die **generateModel**-Methode in **SimulinkSystem** ruft einfach die **generateModel**-Methode der zugehörigen Blöcke, Verbindungen und Subsysteme auf.

Mithilfe von diesem Framework ist es dann Möglich, Simulink Modell in objektorientiertweise, ordentlich, fehlerfrei zu generieren. Der untere Code zeigt ein einfaches Beispiel.

```
SimulinkSystem system = new SimulinkSystem(modelName);

// Add blocks
system.addBlock(new DCVoltageSource(system, "DC"));
system.addBlock(new Resistor(system, "R1"));
system.addBlock(new ElectricalReference(system, "Ref1"));
system.addBlock(new SolverConfiguration(system, "Solver1"));
```

```
// Set parameters
system.getBlock("DC").setParameter("v0", 10);
system.getBlock("R1").setParameter("R", 50);

// Bind blocks
system.addConnection(new SimulinkConnection(system.getBlock("DC").getInPort(0),
        system.getBlock("R1").getInPort(0), system));
system.addConnection(new SimulinkConnection(system.getBlock("R1").getOutPort(0),
        system.getBlock("Ref1").getInPort(0), system));
system.addConnection(new SimulinkConnection(system.getBlock("DC").getOutPort(0),
        system.getBlock("Ref1").getInPort(0), system));
system.addConnection(new SimulinkConnection(system.getBlock("Solver1").getInPort(0),
        system.getBlock("Ref1").getInPort(0), system));

try {
    MatlabEngine matlab = MatlabEngine.startMatlab();
    system.generateModel(matlab);
    matlab.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

3.2 Java Bibliotheken für XML

KBL ist ein XML-basiertes Datenformat, das, wie der Name bereits andeutet, hauptsächlich Informationen über den Kfz-Kabelbaum speichert (siehe Kapitel 2.3.1).

Es stehen zahlreiche Java-Bibliotheken zur Verfügung, die die Arbeit mit XML erleichtern. Eine der bekanntesten ist Java Architecture for XML Binding (JAXB), ein Framework, das die Konvertierung von Java-Instanzen in XML-Daten und umgekehrt ermöglicht [8]. JAXB vereinfacht die Verarbeitung von XML-Daten in Java-Anwendungen erheblich, indem es eine klare Verknüpfung zwischen XML-Dokumenten und Java-Klassen herstellt.

Die beiden Hauptprozesse, die dabei eine Rolle spielen, sind:

- **Marshalling**: Dieser Prozess beschreibt die Umwandlung von Java-Instanzen in XML-Daten. Mit JAXB können Entwickler Java-Objekte in ein XML-Format transformieren, das leicht gelesen, gespeichert oder über Netzwerke übertragen werden kann. Durch spezielle Annotations in den Java-Klassen wird festgelegt, wie diese Instanzen in XML-Elemente und -Attribute umgewandelt werden.
- **Unmarshalling**: Hierbei handelt es sich um den umgekehrten Prozess, bei dem XML-Daten in Java-Instanzen konvertiert werden. Dieser Vorgang ermöglicht es Anwendungen, XML-Daten zu lesen und in ein Format zu bringen, das direkt in Java verwendet werden kann. JAXB gewährleistet dabei, dass die aus dem XML-Dokument extrahierten Daten korrekt in die entsprechenden Java-Instanzen eingefügt werden, wobei die Struktur und Typintegrität der Objekte gewahrt bleibt.

Soll das Ziel hingegen die direkte Modifizierung der XML-Daten sein, ist JAXB nicht mehr die geeignete Wahl, da die Daten, die aus dem Marshalling-Prozess resultieren,

möglicherweise nicht identisch mit den Originaldaten sind. Hier kommt Java Document Object Model for XML (JDOM) ins Spiel, eine beliebte Bibliothek zur Verarbeitung von XML in Java, die eine intuitive API zum Erstellen, Bearbeiten und Verarbeiten von XML-Dokumenten bereitstellt. Für die direkte Bearbeitung von XML ist JDOM daher besser geeignet als JAXB.

Um die KBL-Daten zu verarbeiten, müssen zunächst Java-Klassen erstellt werden, die der Struktur der KBL-Daten entsprechen. Diese Klassen sind entscheidend, um die XML-Daten später mithilfe von JAXB in Java-Objekte umzuwandeln. Ein Beispiel hierfür ist das `Connection`-Element aus den KBL-Daten, das eine Verbindung zwischen zwei Punkten beschreibt. Für jedes in den KBL-Daten vorkommende `Connection`-Element wird eine entsprechende Instanz der Java-Klasse erzeugt, die ihre Attribute speichert und auf verschachtelte Elemente verweist.

Abbildung 3.3: XML-Element und dessen Java-Stellvertreter

Der folgende Code dient dazu, KBL-Daten zu parsen und die KBL-Elemente in Java-Objekte zu überführen. Die entsprechenden KBL-Elemente werden im Kapitel 2.3.1 näher vorgestellt. Sobald diese Daten in Form von Java-Objekten vorliegen, können sie einfach durchlaufen werden, um programmgesteuert ein Simulink-Modell zu erstellen.

```
public class KBLParser {
    private List<File> kblFiles = new ArrayList<>();

    private KBLContainer kblContainer;
    private List<ConnectorHousing> connectorHousings = new ArrayList<>();
    private List<GeneralWire> generalWires = new ArrayList<>();
    private List<Harness> harnesses = new ArrayList<>();
    private List<Connection> connections = new ArrayList<>();
    private List<ConnectorOccurrence> connectorOccurrences = new ArrayList<>();
    private List<GeneralWireOccurrence> generalWireOccurrences = new ArrayList<>();

    public KBLParser(List<String> paths) {
        for (String path : paths) {
            File file = new File(path);
            if (path.endsWith(".kbl")) {
                kblFiles.add(file);
```

```
            }
        }
        init();
    }

    private void init() {
        try {
            JAXBContext jaxbContext = JAXBContext.newInstance(KBLContainer.class);
            Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();

            for (File kblFile : kblFiles) {
                JAXBElement<KBLContainer> rootElement = jaxbUnmarshaller.
                        unmarshal(new StreamSource(kblFile), KBLContainer.class);
                kblContainer = rootElement.getValue();
                connectorHousings.addAll(kblContainer.getConnectorHousings());
                generalWires.addAll(kblContainer.getGeneralWires());
                harnesses.add(kblContainer.getHarness());
            }

            for (Harness harness : harnesses) {
                connections.addAll(harness.getConnections());
                connectorOccurrences.addAll(harness.getConnectorOccurrences());
                generalWireOccurrences.addAll(harness.getGeneralWireOccurrences());
            }
        } catch (JAXBException e) {
            e.printStackTrace();
        }
    }

    public void generateModel() {
        for (ConnectorOccurrence connectorOccurrence : connectorOccurrences) {
            // Utilizing the framework described in Chapter 3.1
            // to generate Simulink elements
            // based on the information from ConnectorOccurrences.
        }
        for (Connection connection : connections) {
            // Utilizing the framework described in Chapter 3.1
            // to generate Simulink elements
            // based on the information from Connection.
        }
    }
}
```

3.3 Evaluation

Jede KBL-Datei deckt nur einen Teil des gesamten Kabelbaums im Bordnetz ab. Dabei werden die über Stecker mit dem Bordnetz verbundenen Komponenten, wie ECUs, Sensoren und Aktoren, als Simulink-Subsysteme abstrahiert. Die Leitungen werden einfach als Widerstände modelliert. Um die Übersichtlichkeit zu verbessern, sind diese Widerstände ebenfalls in Simulink-Subsystemen untergebracht. Informationen zur Querschnittsfläche und Länge der Leitung in mm^2 und mm folgen direkt auf den Signalnamen und dienen als Basis zur Berechnung des Leitungswiderstands.

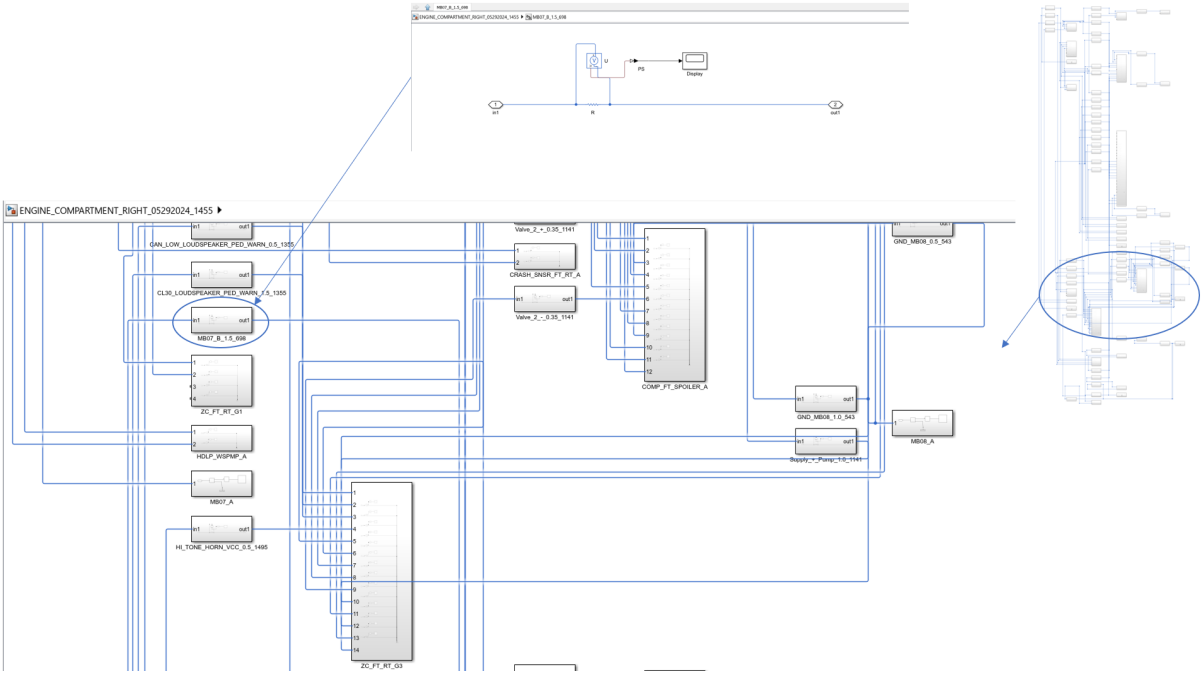

Abbildung 3.4: Übertragung des Kabelbaums aus KBL-Daten nach Simscape

Da im KBL keine vollständigen Komponentendaten vorhanden sind, ist es nicht möglich, ausschließlich mit den KBL-Daten als Informationsquelle ein ausführbares Simulationsmodell für elektrische Simulationen zu generieren. Der Architekt weiß zwar, dass es sich bei einer Komponente beispielsweise um einen Radarsensor oder einen Motor handelt, aber das Java-Programm kann dies nicht automatisch erkennen. Zudem ist es in der Praxis schwierig, den Stromverlauf jeder einzelnen Komponente zu messen und diesen dann dem entsprechenden Simulink-Subsystem zuzuordnen (siehe Kapitel 2.3.3).

Auch für die Erstellung eines thermischen Simulink-Modells fehlen in den KBL-Daten wesentliche Informationen. Der Anwender der Software muss diese fehlenden Daten, wie den maximal dauerhaft durch die Leitung fließenden Strom, die Dicke der Isolierung und das Material des Leiters, über eine grafische Benutzeroberfläche ergänzen.

Die erste Spalte entspricht dem **General_wire** in der KBL, einem Allgemeinkabel mit definierter Querschnittsfläche und Isolationsfarbe. Die zweite Spalte repräsentiert die **Connection**, also die elektrische Verbindung zwischen zwei Kontaktpunkten (siehe Kapitel 2.3.1). Jedes Allgemeinkabel ist einer Menge konkreter Verbindungen zugeordnet. Der Benutzer muss dabei eine Vorstellung davon haben, wie groß der maximal dauerhaft

fließende Strom durch diese Verbindungen ist.

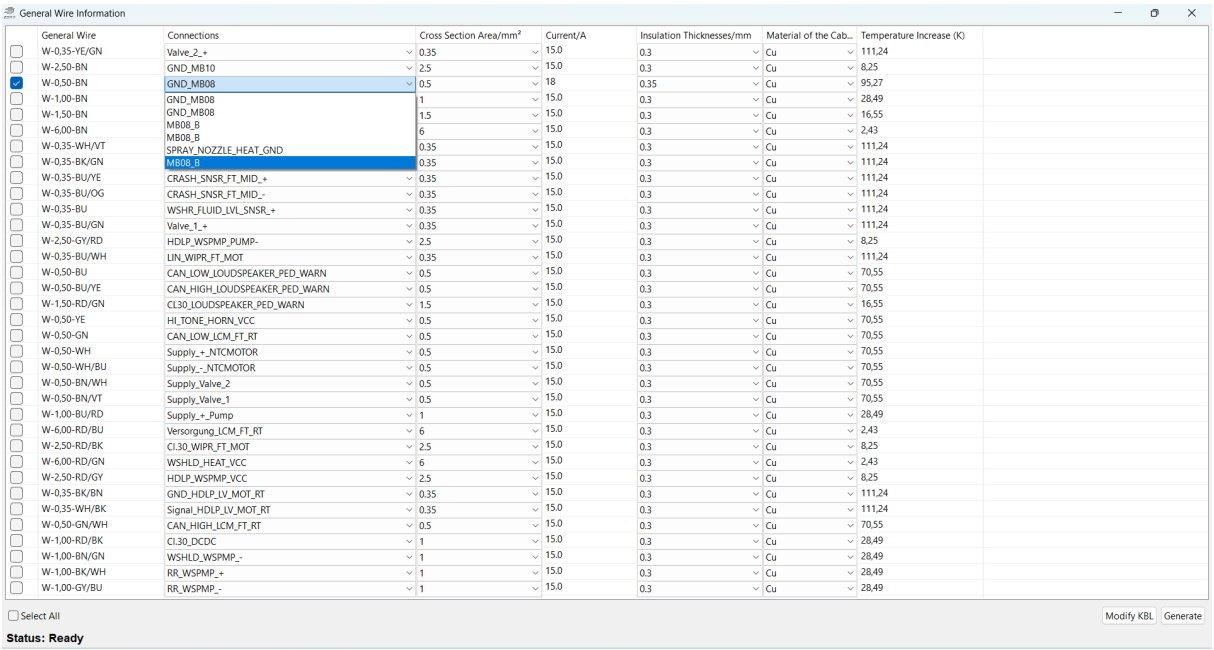

Abbildung 3.5: Benutzeroberfläche zur Ausfüllen der fehlenden Informationen

Anhand der Gleichung (11) lässt sich die Endtemperatur des Leiters direkt in Java berechnen, ohne zu MATLAB wechseln zu müssen. Um einen dynamischen Stromverlauf zu erhalten, wählt der Anwender zunächst die zu optimierenden Allgemeinkabel aus. Auf Basis der Informationen aus der Tabelle werden anschließend die Simulink-Modelle generiert.

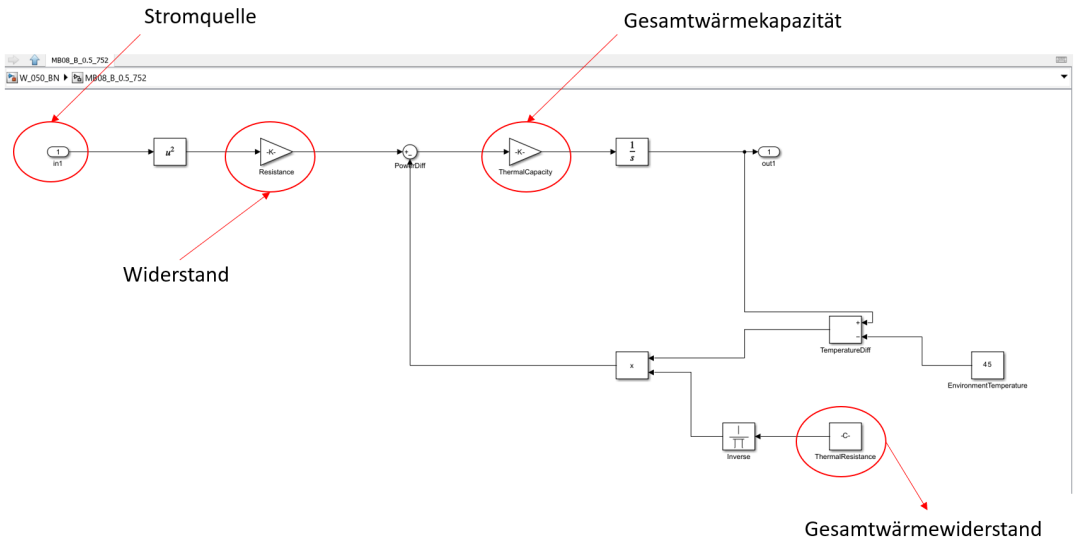

Abbildung 3.6: Simulink-Modell zur Simulation des dynamischen Temperaturverlaufs eines Leiters, siehe Gleichung (13)

Hierbei wird der Temperatursverlauf eines Kuperleiter mit Querschnittfläche $0.5mm^2$ simuliert. Die Dicke der Isolierung beträgt $0.35mm$. Der Leiter wird mit dem Strom in Höhe 18A belastet. Der Gesamtwärmeübergangskoeffizient α_{ges} ist nicht herleitbar, und wird anhand des Experimentergebnis bestimmt (siehe Kapitel 2.3.2).

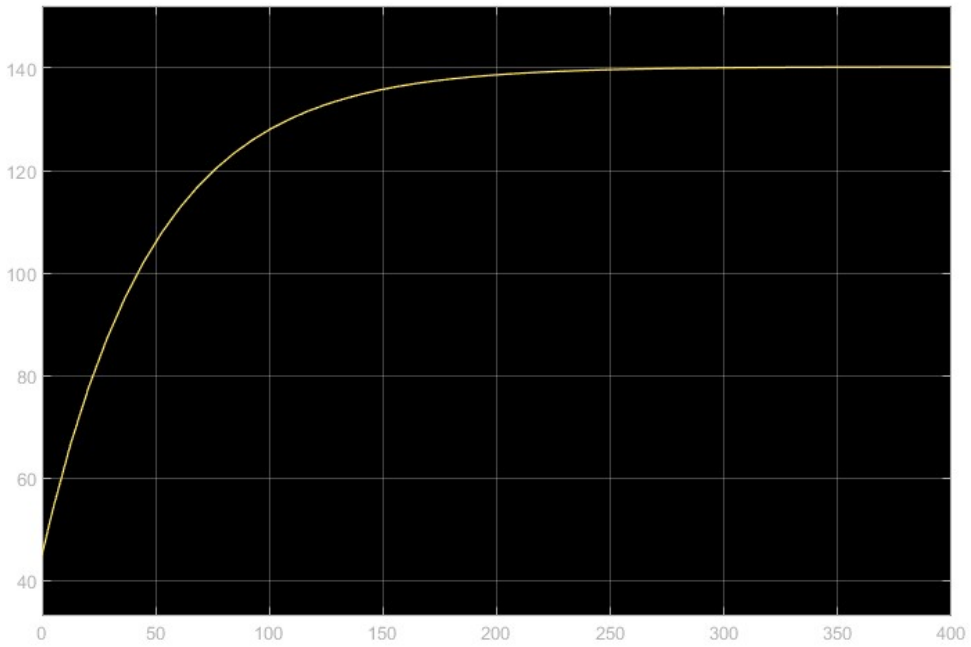

Abbildung 3.7: Simulationsergebnis

Der Strom, der zur Übertragung des Signals durch die Leitung fließt, führt kaum zu einer merklichen Temperaturerhöhung. In diesem Fall spielt hauptsächlich die mechanische Belastung eine Rolle bei der Bestimmung der Querschnittsfläche. Kritischer sind jedoch die Leitungen, die für die Energieversorgung zuständig sind. Eine zu starke Temperaturerhöhung kann zur Verschmelzung oder schnellen Alterung der Isolierung führen. In der Praxis sind die Querschnittsflächen aus Sicherheitsgründen oft überdimensioniert, sodass der Strom nur geringe Temperaturerhöhungen verursacht. Hier besteht die Möglichkeit zur Optimierung durch Simulation, um das Gewicht der Leitungen zu reduzieren.

4 Schlussfolgerung und Ausblick

Dieser Artikel beschreibt ein Verfahren zur programmatischen Übertragung von XML-Daten nach Simulink. Da eine Simulation auf elektrischer Ebene nicht möglich ist, wird die thermische Simulation zur Bestimmung des Temperaturverlaufs des Leiters als Anwendungsfall herangezogen.

Das VEC als neues standardisiertes XML-Datenformat bietet deutlich mehr nützliche Informationen als KBL (siehe Kapitel 2.3.1). Nachdem PREEvision den Export von VEC-Daten unterstützt, stehen neben der thermischen Simulation auch weitere Simulationsmöglichkeiten zur Verfügung. Die Vorgehensweise bleibt jedoch dieselbe, da sowohl VEC als auch KBL auf dem XML-Format basieren.

Literaturverzeichnis

[1] G. Anders. *Rating of Electric Power Cables - Ampacity Computations for Transmission, Distribution, and Industrial Applications*. New York: IEEE Press, 1997.

[2] J. Atheis. „Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262“. Doktorarbeit. Diss. Universität Karlsruhe (TH), 2010.

[3] Harald. Bucher. „Integrierte modell- und simulationsbasierte Entwicklung zur dynamischen Bewertung automobiler Elektrik/Elektronik-Architekturen“. Doktorarbeit. Diss. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 2020.

[4] O. Burkacky, J. Deichmann und J. P. Stein. *Automotive software and electronics 2030*. Technical Report. McKinsey Center for Future Mobility, Juli 2019.

[5] Maja. Diebig. „Entwicklung einer Methodik zur simulationsbasierten Dimensionierung von Kfz-Bordnetzen“. Doktorarbeit. Diss. Dortmund, Germany: Technischen Universität Dortmund, 2016.

[6] Daniel J. Gebauer. „Ein modellbasiertes, graphisch notiertes, integriertes Verfahren zur Bewertung und zum Vergleich von Elektrik/Elektronik-Architekturen“. Doktorarbeit. Diss. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 2016.

[7] ProSTEP iViP. *KBL Specification*. `https://ecad-wiki.prostep.org/specifications/kbl/v25-sr1/`. Accessed: 2024-10-08.

[8] Java Community. *JAXB Documentation*. `https://javaee.github.io/jaxb-v2/`. Zugriff: 2024-06-25.

[9] Kevin. Neubauer. „Skalierbarkeit einer Szenarien- und Template-basierten Simulation von Elektrik/Elektronik-Architekturen in reaktiven Umgebungen“. Doktorarbeit. Diss. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 2022.

[10] Project Group “Car Electric” of VDA Working Group “PLM”. *VDA Harness Description List (KBL) 4964*. 2nd edition (KBL 2.4). Verband der Automobilindustrie. Berlin, Germany: Verband der Automobilindustrie, Nov. 2014. URL: `http://www.vda.de`.

[11] M. Traub, A. Maier und K. L. Barbehön. „Future Automotive Architecture and the Impact of IT Trends“. In: *IEEE Software* 34.3 (Mai 2017), S. 27–32.

[12] VECTOR INFORMATIK GMBH. *PREEvision® Manual Version 8.5*. Manual. Germany, 2017.

[13] Vector Informatik GmbH. *PREEVISION*. Zugriff am: 2024-10-06. 2024. URL: `https://www.vector.com/int/en/products/products-a-z/software/preevision/`.

[14] Volkswagen AG. *KBL (VDA 4964) VOBES spezifische Erweiterungen zu KBL2.4 SR-1*. Techn. Ber. Volkswagen AG, März 2019.

[15] H. Winner u. a. *Handbuch Fahrerassistenzsysteme*. 3. Aufl. Springer, 2015.