Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Modellierung und Deployment von cyber-physischen Systemen basierend auf TOSCA

Alexander Kicherer

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Kálmán Képes, M.Sc.
Beginn am:	27. September 2018
Beendet am:	27. Februar 2019

Kurzfassung

Durch den wachsenden Trend, physische Systeme zu digitalisieren und zu automatisieren ist das Paradigma der cyber-physischen Systeme entstanden. Durch die Digitalisierung und Automatisierung selbst alltäglicher Objekte sowie deren Vernetzung dieser und weiterer digitaler Systeme untereinander und mit Cloud-Anwendungen, wurden weitere Systeme entwickelt. Dadurch entstand ein neues Modell, das Internet of Things. Da sämtliche dieser Systeme aus vielen Komponenten bestehen, ist es sinnvoll, deren Entwicklung und Deployment durch entsprechende Modellierung zu unterstützen. Hierfür existieren einige Modellierungsstandards, jedoch sind diese Ansätze dafür gedacht, entweder das Verhalten, die Softwarekomponenten oder die Hardwarekomponenten eines cyber-physischen oder Cloud-Systems darzustellen. In dieser Arbeit wird eine mögliche Erweiterung des für die Modellierung von Cloud-Anwendungen entwickelten Standards TOSCA vorgestellt, um mit diesem cyber-physische Systeme darstellen und bereitstellen zu können. Hierbei werden die Hardwarekomponenten eines Systems großteils wie dessen Softwarekomponenten behandelt und dargestellt, jedoch werden zusätzliche Einschränkungen und Informationen bei diesen angegeben. Ebenfalls wird darauf basierend das Auffinden von Möglichkeiten für Datenverbindungen sowie für ein Deployment eines auf diese Weise dargestellten Systems behandelt. Dass die vorgestellte Erweiterung des TOSCA-Standards für die Darstellung von cyber-physischen Systemen geeignet ist, wird anhand eines autonomen Fahrzeugs gezeigt. Als Modellierungswerkzeug wurde hierbei OpenTOSCA genutzt. Ebenfalls werden einige Vorschläge zur nötigen Unterstützung von Modellierern durch die grafische Anzeige von generierten Informationen oder das gezielte Ausblenden davon genannt. Teile dieser Vorschläge wurden für diese Arbeit in OpenTOSCA umgesetzt.

Inhaltsverzeichnis

1 Einleitung **11**
1.1 Motivation 11
1.2 Gliederung der Arbeit 12

2 Grundlagen **13**
2.1 Cloud Computing 13
2.2 TOSCA 13
2.3 OpenTOSCA 16
2.4 Hardware 17
2.5 Cyber-physische Systeme 22
2.6 Internet of Things 22

3 Related Work **23**
3.1 Bigraph 23
3.2 Komponentenbasierte Darstellung von Hardware 23
3.3 Schaltplan 25
3.4 Blockdiagramm 26
3.5 MechatronicUML 30

4 Fallstudie autonomes Fahrzeug **31**
4.1 Hardwarekomponenten 33
4.2 Softwarekomponenten 34

5 Modellieren von cyber-physischen Systemen in TOSCA **37**
5.1 Problemstellung 37
5.2 Anforderungen 37
5.3 Modellierung von Hardwarekomponenten in TOSCA 38
5.4 Modellierung von CPS-Softwarekomponenten in TOSCA 47
5.5 Modellierung von Relationen mit Hardwarekomponenten in TOSCA 48

6 Prototypische Implementierung und Analyse **57**
6.1 Modell des autonomen Fahrzeugs 57
6.2 Analyse für Modellierung, Deployment und Management 66
6.3 Reduktion von Komplexität durch Modellierungssichten 71
6.4 Lektionen 74

7 Zusammenfassung und Ausblick **79**

Literaturverzeichnis **81**

Abbildungsverzeichnis

2.1 Struktur und Inhalt eines Service Templates in TOSCA [OASIS13a] 15
2.2 Das OpenTOSCA Ökosystem [OT18] 16
2.3 Die Rückseite eines Desktopcomputers zeigt verschiedene Hardwareports. 18
2.4 Kombiniertes Kabel für den Anschluss von Tastatur, Bildschirm und Maus. Ausführung für die Verwendung zwischen Computer und KVM-Umschalter. 19
2.5 Adapterkabel, welches die Audiosignale der Anschlüsse für Stereo-Audio und Mikrofon zu einem kombinierten Anschluss für Audio und Mikrofon zusammenführt. 19
2.6 Hardwarefunktionen einer Computingkomponente, deren Hardwareports und ihrer möglichen Verbindungen zu Anschlüssen der Komponente schematisch dargestellt. 20
2.7 Ausschnitt aus dem grafischen Datenblatt des Arduino Pro Mini [SF14]. 21

3.1 Beispielhafter Bigraph (oben) und dessen Auflösung in einen Place Graph (links unten) und einen Link Graph (rechts unten). Nach Milner [Mil06] 24
3.2 Darstellung zweier Server und eines Switches in einem Serverschrank (Rack) und einer externen Ressource, verbunden durch Netzwerkkabel, nach [KG14] 25
3.3 Modul mit Mikrocontroller ESP8266 und Speicherchip für Firmware, die Schirmung ist abgenommen, die einzelnen Komponenten sind sichtbar [AI15] 26
3.4 Schaltplan des in Abbildung 3.3 gezeigten Moduls [ITEAD16] 27
3.5 Schaltsymbol des Moduls aus Abbildung 3.3 in einem Schaltplan [ESPA18] . . . 27
3.6 Modul aus Abbildung 3.3 verbaut auf weiterer Schaltung [Handson17] 28
3.7 Ausschnitt aus dem Blockdiagramm des Mikrocontrollers ATMega8. Aus [MC13] 29
3.8 Blockdiagramm des in Abbildungen 3.4 und 3.3 gezeigten Moduls. Ausschnitt aus dem Datenblatt [ESP18] 29
3.9 Beispielhafte Darstellung eines in MechatronicUML modellierten Systems [BDG+14] 30

4.1 Skizzenhafte Darstellung des autonomen Fahrzeugs und seines primären Verhaltens. 31
4.2 Draufsicht des Fahrzeugs, die meisten Komponenten sind sichtbar. 32
4.3 Draufsicht des Fahrzeugs als schematische Darstellung der Komponenten mit Bezeichnung. 33

5.1 Grafische Darstellung einer in TOSCA beschriebenen Node mit Capabilities und Requirements sowie zu diesen zugehöriger Constraints. 39
5.2 Grafische Darstellung einer von Hardwarekomponenten nutzbaren Capability USB-Host und eines Requirements einer Node am Beispiel eines RaspberryPi. In Constraints sind zugehörige Hardwareports mit ihren Anschlüssen angegeben. . . 41
5.3 Grafische Darstellung einer mehrfach vorhandenen und von Softwarekomponenten nutzbaren Capability. 42

5.4 Grafische Darstellung einer von Softwarekomponenten nutzbaren Capability um auf Hardwareports eines Typs zuzugreifen. Das Constraint definiert Referenzen zur zugehörigen Capability für Hardwarekomponenten und den entsprechenden Ports sowie die von Softwarekomponenten nutzbaren IDs. 43
5.5 Grafische Darstellung eines Kabels als Hardwarekomponente mit einer generischen Capability und einem generischen Requirement. Interne Verbindungen zwischen Anschlüssen sind angegeben. 45
5.6 Grafische Darstellung des in Abbildung 2.5 gezeigten Kabels zum Aufteilen beziehungsweise Zusammenführen von Audiosignalen. 46
5.7 Grafische Darstellung eines Treibers als Softwarekomponente mit einer Capability mit virtuellen Ports sowie einem Requirement, welches durch eine Fähigkeit einer Hardwarekomponente erfüllt werden kann. 48
5.8 Grafische Darstellung eines Arduino Pro Mini und eines USB zu Seriell Adapters sowie eines sie verbindenden Kabels. Zwei Relationen definieren, welche Capabilities und Requirements der beteiligten Komponenten miteinander verbunden werden sollen. 51
5.9 Mögliche Relationen zwischen Hardwarekomponenten und darauf gespeicherten und ausgeführten Softwarekomponenten am Beispiel eines Betriebssystems. . . . 53
5.10 Zugriff einer Softwarekomponente auf einen Hardwareport einer Hardwarekomponente, dargestellt durch eine Relation. 54

6.1 Gesamtansicht aller Komponenten und Beziehungen im Modell des autonomen Fahrzeugs in TOSCA. Ansicht mit dem Topologymodeler von Winery generiert. . 58
6.2 Ausschnitt des in TOSCA erstellten Modells des autonomen Fahrzeugs mit angezeigten Capabilities und Requirements der Batterien und des RaspberryPis. . . . 60
6.3 Detail des in TOSCA erstellten Modells des autonomen Fahrzeugs, zu sehen eine Softwarekomponente zum Ansteuern des Motortreibermoduls sowie zugehörige Hardwarekomponenten welche diese Verbindung und das Ausführen der Softwarekomponente ermöglichen. 63
6.4 Ausschnitt des in TOSCA erstellten Modells des autonomen Fahrzeugs, zu sehen eine Softwarekomponente zum Ansteuern des Motortreibermoduls sowie zugehörige Hardwarekomponenten welche diese Verbindung und das Ausführen der Software ermöglichen. 65
6.5 Grafische Darstellung der in 6.1.5 genannten Komponenten mit Capabilities und Requirements sowie deren Relationen untereinander. 67
6.6 Ansicht des Modells des autonomen Fahrzeugs mit ausgeblendeten HardwareNodes. 72
6.7 Detail der in Abbildung 6.1 gezeigten Darstellung des als Fallstudie genutzten autonomen Fahrzeugs. Gezeigt sind unter anderem die Komponenten des Antriebsstrangs sowie die Darstellung mit substituierten Komponenten. 73
6.8 Ansicht des Modells des autonomen Fahrzeugs mit abstrahierten Kabelverbindungen. 75

Verzeichnis der Listings

6.1 Anschluss des Hardwareports einer Batterie in einem Constraint dargestellt. . . . 60
6.2 Die Anschlüsse der Hardwareports der ausgehenden Stromversorgung eines RaspberryPi, dargestellt in einem Constraint. 61
6.3 Anschlüsse eines der Hardwareports von Y-Kabeln in einem Constraint dargestellt. 62

1 Einleitung

Mit der wachsenden Digitalisierung und Automatisierung des Alltags bei dem physische und virtuelle Prozesse vereint cyber-physische Systeme ergeben, steigt auch die Komplexität dieser Systeme. So werden Fahrzeuge, Fertigungsprozesse und Alltagsgegenstände wie Haushaltsgeräte zunehmend mit virtuellen Ressourcen, wie etwa der Cloud verknüpft. Zusätzlich wird bei einer wachsenden Zahl an Systemen Cloud-Technologie zur Umsetzung des Systems oder Teilen davon verwendet, insbesondere ist dies bei Systemen nach dem Paradigma des Internet of Things der Fall, da hierbei cyber-physische Systeme mit Cloud-Anwendungen zu einem Gesamtsystem verschmelzen. Da sämtliche solcher komplexer Systeme erstellt und verwaltet werden müssen, ist es notwendig, sie in einem für Menschen verständlichen Modell darzustellen. Da jedoch auch das Erstellen und die Verwaltung dieser Systeme aufgrund ihrer Größe sowie aufgrund von Geschwindigkeitsvorteilen automatisch erfolgen muss, müssen diese Modelle zusätzlich für Programme mit dieser Funktion nutzbar sein und sämtliche dafür relevanten Daten beinhalten. Die Modellierung hybrider Systeme aus physischen sowie virtuellen Komponenten stellt eine große Herausforderung dar, da keine ganzheitliche Sprache existiert, welche die Modellierung der Hardware von cyber-physischen Systemen und die Modellierung von Software in sich vereint. Da der Modellierungsstandard TOSCA sehr generisch ist, bietet es sich an, dessen bestehende Modellierungskonzepte für die Modellierung von cyber-physischen Systeme zu erweitern.

1.1 Motivation

Cyber-physische Systeme entstehen, wenn virtuelle und physische Ressourcen miteinander zu einem System verknüpft werden. Insbesondere im Bereich des Internet of Things (Iot) bestehen viele moderne cyber-physische Systeme dabei nicht nur aus den physischen Ressourcen des Systems sowie darauf ausgeführten Softwarekomponenten, sondern zusätzlich aus in einer Cloud-Umgebung ausgeführten Softwarekomponenten. Da sowohl reine cyber-physische Systeme als auch reine Cloud-Anwendungen bereits komplexe Systeme darstellen, sind solche hybriden Systeme nochmals weitaus komplexer. Mit der stetig wachsenden Anzahl und Komplexität von Cloud-basierten Systemen wächst auch der Bedarf, solche Systeme automatisch zu verwalten. Um die Verwaltung von Cloud-Systemen zu standardisieren, wurde mit TOSCA [OASIS13a] ein Standard entwickelt, welcher die Modellierung und das Management solcher Systeme ermöglicht. Dabei ist der Fokus von TOSCA die Beschreibung von Anwendungen, welche aus Softwarekomponenten bestehen die in einer Cloud-Umgebung ausgeführt werden. Durch die steigende Verknüpfung von physischen und virtuellen Ressourcen stellt sich entsprechend die Frage, in wie weit Modellierungssprachen für Cloud-Anwendungen wie TOSCA auch die Beschreibung von cyber-physischen Systemen ermöglichen. Hierdurch stellt sich die Frage, wie die Komponenten von solchen Systemen, wie beispielsweise Rechner und Mikrocontroller und weitere elektronische Komponenten, in TOSCA abgebildet werden können. Ebenso stellt sich die Frage, auf welche Art die Verbindungen zwischen diesen

Komponenten in TOSCA abgebildet werden können. Zusätzlich ist zu erörtern, wie Verbindungen zwischen Softwarekomponenten und Hardwarekomponenten in TOSCA abgebildet werden können. Da TOSCA nicht nur zur Darstellung von Komponenten und deren Verbindungen untereinander dient, sondern ebenso Angaben zum Deployment dieser Komponenten anbietet, folgt darauf die Frage, wie Deployment von Hardwarekomponenten in TOSCA dargestellt werden kann. Dazu gehört auch die Frage, wie dabei das Erstellen der Verbindungen zwischen diesen Komponenten ermöglicht werden kann.

Ziel dieser Arbeit ist es, aufzuzeigen, dass mit nur geringen Erweiterungen des TOSCA-Standards auch Hardwarekomponenten und darauf vorhandene Softwarekomponenten modelliert und verwaltet werden können. Zusätzlich soll aufgezeigt werden, dass auch Relationen zwischen Hardwarekomponenten sowie zwischen Hardware- und Softwarekomponenten dargestellt werden können. Dabei liegt das Hauptaugenmerk auf elektronischen Hardwarekomponenten. Diese Arbeit soll sich dabei vorwiegend auf heterogene und statische cyber-physische Systeme beziehen.

1.2 Gliederung der Arbeit

Diese Arbeit ist in folgender Weise gegliedert: In Kapitel 2 werden die Grundlagen dieser Arbeit zu TOSCA, Hardwarekomponenten und cyber-physischen Systemen beschrieben. Kapitel 3 nennt relevante existierende Ansätze um Hardware und cyber-physische Systeme zu modellieren. Daraufhin wird ein zu Demonstrations- und Forschungszwecken für diese Arbeit entwickeltes einfaches autonomes Fahrzeug in Kapitel 4 vorgestellt, welches als Fallstudie für die ermittelten Konzepte gilt. In Kapitel 5 werden die Anforderungen für die Modellierung von Hardwarekomponenten in TOSCA und eine Lösung dafür vorgestellt. Ebenfalls werden an dieser Stelle einige der daraus ableitbaren Informationen genannt, welche für Systementwickler und im darauf folgenden Kapitel relevant sind. In Kapitel 6 wird das Modell des in Kapitel 4 vorgestellten Fahrzeugs in TOSCA gezeigt sowie Details zur Modellierung einzelner Komponenten, Relationen und Systemteile erklärt. Ebenfalls dient es der Aufzählung von Hürden und Lösungen für Deployment und Management von Hardwarekomponenten mittels TOSCA, Lektionen aus der Modellierung dieses Fahrzeugs werden ebenfalls genannt. Schließlich fasst Kapitel 7 die Ergebnisse dieser Arbeit zusammen und stellt Anknüpfungspunkte für weitere Arbeiten vor.

2 Grundlagen

Dieses Kapitel dient der Übersicht über die dieser Arbeit zugrunde liegenden Werke. Anfänglich wird in Abschnitt 2.1 das Prinzip des Cloud Computing vorgestellt. In Abschnitt 2.2 wird der TOSCA-Standard zur Modellierung von Cloud-Anwendungen erklärt. Abschnitt 2.3 stellt daraufhin mit OpenTOSCA eine Opensource-Implementierung eines Editors und einer Laufzeitumgebung für diesen Standard vor. Daraufhin werden in Abschnitt 2.4 verschiedene Arten an Hardwarekomponenten und ihre Eigenschaften vorgestellt. Auf Cyber-physische Systeme und das damit verwandte Internet of Things wird in Abschnitt 2.5 und Abschnitt 2.6 eingegangen.

2.1 Cloud Computing

Unter Cloud Computing wird ein Paradigma verstanden, in welchem die Verwaltung eines Pools an Hardwareressourcen durch einen Cloud-Anbieter erfolgt. Diese Ressourcen sind dabei mit einem Netzwerk wie dem Internet verbunden damit darüber mit geringem Aufwand und entsprechend schnell Hardwareressourcen aus diesem Pool konfiguriert und bereitgestellt werden können [MG+11]. Diese Ressourcen werden dabei mit vielen Nutzern geteilt, wodurch von einem Nutzer nicht benötigte Ressourcen von anderen Nutzern verwendet werden können. Da diese Ressourcen über das Netzwerk geordert und angesprochen werden können ist es möglich, darauf basierende Softwaresysteme automatisiert zu erstellen und zu verwalten. Hierdurch ermöglicht dieses Paradigma ein Skalieren eines solchen Systems durch kurzfristiges Hinzufügen weiterer Ressourcen und darauf ausgebrachter Softwarekomponenten oder durch Entfernen unbenötigter Ressourcen und Komponenten. Oftmals werden diese Ressourcen in Form von virtuellen Maschinen angeboten, auf welchen daraufhin vom Nutzer oder von einem Programm Softwarekomponenten installiert und eingerichtet werden. Jedoch entwickeln sich auch weitere Angebote, durch welche auch ausschließlich Speichermöglichkeiten für Dateien wie AWS Buckets von Cloud-Anbietern bezogen werden können. Ebenfalls existieren von Cloud-Anbietern neben Angeboten für Hardwareressourcen auch Angebote für Softwareressourcen wie Datenbanken und Cloud-Speicher.

2.2 TOSCA

Die Topology and Orchestration Specification for Cloud Applications (TOSCA) [OASIS13a] ist ein OASIS-Standard zur Modellierung von Cloud-Anwendungen. Das Ziel dieses Standards ist Cloud-Anwendungen technologie- und anbieterunabhängig in XML [OASIS13b] oder YAML [PRS16] zu beschreiben. Damit kann eine Abhängigkeit von einem Cloud-Anbieter und dessen möglicherweise proprietären Technologien, ein sogenannter Vendor Lock-In, verhindert werden. Jedes nach diesem Standard implementierte System kann eine in TOSCA dargestellte Cloud-Anwendung instanziieren und verwalten, sprich seine Komponenten nach den Vorgaben des Modells

bereitstellen und verändern. Eine Cloud-Anwendung wird in TOSCA durch ein sogenanntes *Service Template* definiert, der prinzipielle Aufbau davon ist in Abbildung 2.2 skizziert. Ein Service Template beinhaltet mit einem *Topology Template* eine Topologie der Cloud-Anwendung, in welcher ihre einzelnen Komponenten und die Relationen zwischen diesen beschrieben werden. Die Komponenten einer Cloud-Anwendung werden in TOSCA als *Node Templates* bezeichnet, Relationen als *Relationship Templates*. Zusätzlich kann bei Node Templates angegeben werden, dass sie bestimmte Fähigkeiten anbieten oder für ihre Funktion benötigen, sogenannte *Capabilities* oder *Requirements*. Diese Requirements können über eine Relation zu einer anderen Komponente mit entsprechender Capability erfüllt werden. Ebenfalls können Node Templates und Relationship Templates Eigenschaften besitzen, sogenannte *Properties*. Diese Properties stellen beispielsweise die initiale Konfiguration einer Komponente dar, welche zur Verwaltung einer Instanz dieses Node Templates nötig sind. Hierzu zählen unter anderem Logindaten und IP dieses Systems sowie der Port, auf welchem ein Login erfolgen kann. Optional können bei Relationship Templates sogenannte *RelationshipConstraints* angegeben werden.

Um die Wiederverwendbarkeit von Node bzw. Relationship Templates zu ermöglichen, besitzt jedes Node bzw. Relationship Template auf einen Node- bzw. Relationship Type, der den Typ des Node Templates bzw. des Relationship Templates spezifiziert. Diese Typen werden in einem *Definitions*-Dokument portabel gespeichert und definieren in sogenannten Properties die initiale Konfiguration von Komponenten und Relationen bzw. Node Templates und Relationship Templates. Ebenfalls verweisen Capabilities und Requirements auf sie spezifizierende Capability Types beziehungsweise Requirement Types. Dabei geben Requirement Types an, durch welchen Capability Type sie erfüllt werden können. Sämtliche dieser Typen können sich von weiteren Typen ableiten und dabei deren Eigenschaften übernehmen. Ebenfalls können in diesen Requirements und Capabilities sogenannte *Constraints* angegeben werden. Der Inhalt dieser Constraints ist nicht weiter spezifiziert, muss jedoch im Format von XML beziehungsweise YAML sein.

Um die modellierten Node Templates bereitzustellen und während der Laufzeit zu verwalten, definieren Node Types zusätzlich so genannte Management Interfaces mit Management Operationen. Diese ermöglichen das Erstellen, Entfernen und Verwalten von Instanzen von Node Templates des entsprechenden Node Types. Zusätzlich werden für alle Node Types die beim Deployment eines entsprechenden Node Templates benötigten Dateien in zugehörigen Implementation Artifacts vorgehalten. Ebenfalls werden beim Deploymentvorgang auszuführende Skripte und Programme in zugehörigen Deployment Artifacts vorgehalten.

In TOSCA werden mit Relationen zwischen zwei Softwarekomponenten je nach Typ dieser Relation verschiedene Beziehungen dargestellt. Damit können Kommunikationsverbindungen dargestellt werden oder ein Verweis, auf welcher weiteren Komponente eine Komponente gehostet ist. Ebenfalls kann eine Abhängigkeit einer Komponente von einer weiteren dargestellt werden. Diese Relationen dienen verschiedenen Zwecken, unter anderem für sie Modellierer zum Verständnis der dargestellten Cloud-Anwendung. Stellt eine Relation dar, dass eine Komponente auf einer weiteren gehostet ist, so ist daraus zum einen zu erkennen, dass erst die Zielkomponente vorhanden sein muss, bevor die Komponente an der Quelle der Relation bereitgestellt werden kann. Zum anderen zeigt diese Relation, dass das Deployment durch Management Operationen der Zielkomponente erfolgen muss. Eine typischer Relationstyp ist der Relationship Type *HostedOn*. Relationen dieses Typs geben an, dass ihre Quellkomponente auf ihrer Zielkomponente bereitgestellt werden muss. Stellt eine Relation dagegen eine Kommunikationsverbindung zwischen zwei Softwarekomponenten dar, so dient sie beim Deployment der Cloud-Anwendung als Hinweis, dass in der Zielkomponente und den sie

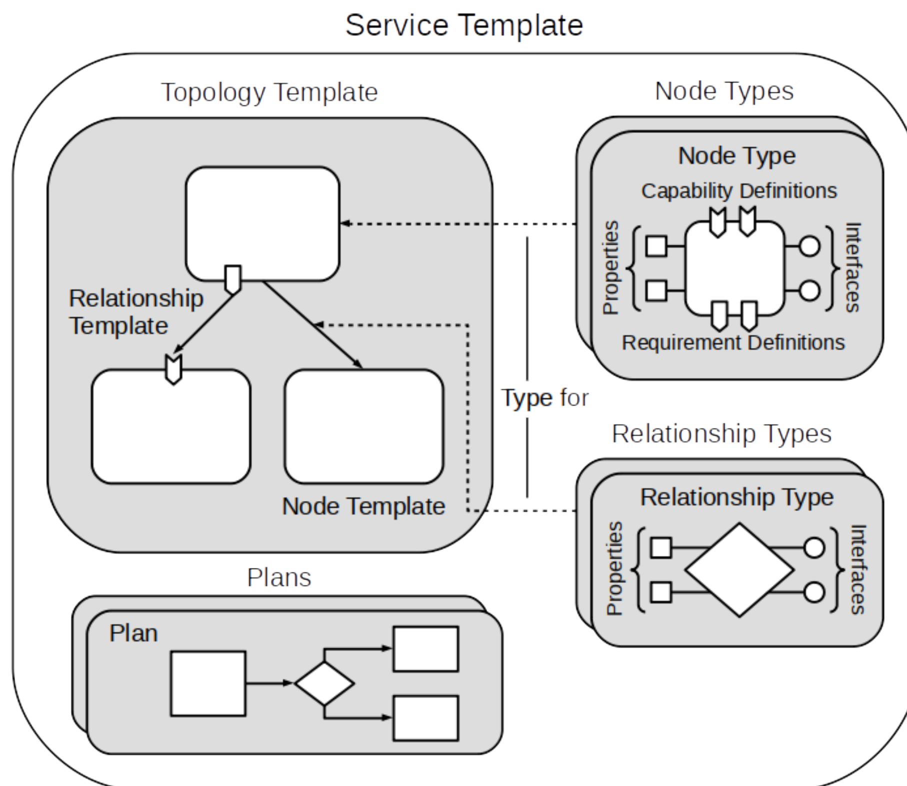


Abbildung 2.1: Struktur und Inhalt eines Service Templates in TOSCA [OASIS13a]

hostenden Komponenten Informationen zum Aufbau dieser Kommunikationsverbindung zu finden sind. Ebenfalls zeigt eine solche Relation, dass eine Management Operation der Quellkomponente genutzt werden muss um diese Kommunikationsverbindung einzurichten. Typischerweise wird dies durch eine Relation vom Typ *ConnectsTo* dargestellt. Eine weitere mögliche Relation stellt die Abhängigkeit einer Komponente von einer weiteren dar, ein typisches Beispiel hierfür ist die Abhängigkeit einer Softwarekomponente von einer Bibliothek. Dies bedeutet schlicht, dass beim Deployment erst alle Komponenten, von denen eine Komponente abhängig ist, instanziiert sein müssen, bevor diese Komponente instanziiert werden kann.

Da zum Bereitstellen einer Cloud-Anwendung nicht nur ihre Komponenten und deren Relationen benötigt werden, sondern auch die Schritte zum Bereitstellen und Konfigurieren dieser, muss dies durch entsprechende Prozesse dargestellt werden. In TOSCA können für das Erstellen, Beenden und Verändern der im Service Template dargestellten Cloud-Anwendung Prozessmodelle als sogenannte *Plans* im Service Template hinterlegt werden. Diese Pläne können in beliebigen Sprachen zur Definition von Prozessmodellen definiert werden. Dabei stellen sie vorwiegend das Erstellen und Terminieren von Komponenten sowie den Aufbau von Verbindungen zwischen diesen dar. Ebenfalls kann bei bestimmten Ereignissen eine Cloud-Anwendung durch das Ausführen entsprechender Pläne verändert werden. Hierbei kann beispielsweise auf eine hohe Auslastung von Datenbankservern reagiert werden, indem weitere virtuelle Maschinen mit Datenbankservern bereitgestellt und mit den bisher im System existierenden Komponenten verbunden werden. Bei geringer Auslastung können dagegen gezielt bestehende Komponenten aus dem System entfernt und terminiert werden. Hierdurch kann das Verhalten einer skalierenden Cloud-Anwendung definiert werden kann.

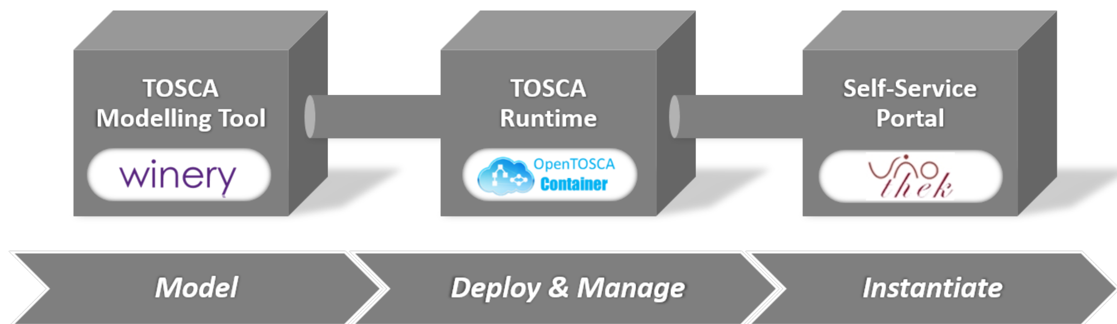


Abbildung 2.2: Das OpenTOSCA Ökosystem [OT18]

2.3 OpenTOSCA

Das OpenTOSCA Ökosystem ist eine Open Source Implementierung zum Modellieren, Bereitstellen und Verwalten von Cloud-Anwendungen nach dem TOSCA Standard. Dabei besteht das OpenTOSCA-Ökosystem aus einer Runtime-Implementierung, dem Modellierungswerkzeug Winery [KBBL13] und dem Self-Service Portal Vinothek [BBKL14]. Winery ist ein Webeditor, welcher das Modellieren von Cloud-Anwendungen unterstützt indem er das Erstellen und Verändern von Node Types und Relationship Types anbietet. Ebenfalls können in diesem Webeditor die möglichen Capabilities und Requirements der Node Types sowie die zugehörigen Capability und Requirement Types verwaltet werden. Ein weiterer Teil dieses Webeditors unterstützt die Erstellung von Topologien. In ihm werden Topology Templates als Graph dargestellt, wobei einzelne Node Templates als Knoten und deren Relationship Templates als Kanten dargestellt werden. In diesem Teil des Webeditors können zusätzlich für jedes Node Template die im zugehörigen Node Type definierten Capabilities und Requirements sowie weitere Eigenschaften, die Properties, angegeben werden. Ist zum Beispiel mittels Winery eine Cloud-Anwendung modelliert worden, so kann dieses Modell als CSAR-Archiv exportiert werden und in die OpenTOSCA Runtime geladen werden. Damit wird diese Cloud-Anwendung über das Self-Service Portal Vinothek angeboten und Nutzer dieses Portals können darüber eine Instanz dieser Anwendung erstellen lassen. Mit OpenTOSCA wurde bereits das Deployment von IoT-Anwendungen auf entsprechenden Geräten im wissenschaftlichen Kontext erforscht [SBK+16], jedoch wurde hierbei von schon vorhandenen und eingerichteten Geräten ausgegangen und der Fokus auf das Deployment und Management von Softwarekomponenten auf den Betriebssystemen dieser Geräte gelegt. Zusätzlich wurden bei der Darstellung des genutzten IoT-Systems in TOSCA als einzige Hardwarekomponenten die verwendeten Kleinstrechner dargestellt, allerdings keine weiteren der genutzten Komponenten wie Aktoren und Sensoren. Ebenfalls wurden keine physischen Verbindungen zwischen den Komponenten oder deren Stromversorgung dargestellt. Da OpenTOSCA als Forschungsprototyp entwickelt wurde und der Quellcode dazu offen liegt eignet es sich für die Implementierung neuer Funktionen zu Testzwecken.

2.4 Hardware

Als Hardware werden all jene Komponenten eines Systems bezeichnet, welche im Gegensatz zu Software in Form von physischen Objekten existieren. Hardwarekomponenten können verschiedenen Zwecken dienen und entsprechend unterschiedliche Funktionen und Eigenschaften haben. Zum einen gibt es *elektronische Hardwarekomponenten*, welche bestimmte Signale empfangen und verarbeiten, oder auch aussenden oder weiterleiten. Zum anderen existieren rein passive *mechanische Hardwarekomponenten*, welche als Montagebasis oder Schutz für weitere Komponenten oder der Kraftübertragung auf mechanischem Wege dienen. Im informationstechnischen Bereich werden vorwiegend elektronische Hardwarekomponenten als Hardware bezeichnet. Im Folgenden sollen die typischen Eigenschaften und Fähigkeiten von Hardwarekomponenten aufgeführt werden.

Elektronische Hardware

Elektronische Hardwarekomponenten zeichnen sich dadurch aus, dass ihre Funktion durch elektrischen Strom ermöglicht wird oder sie diesen Strom zur Verfügung stellen. In den meisten Fällen verarbeiten diese Hardwarekomponenten elektronische Signale, teilweise jedoch auch elektromagnetische oder optische Signale, und sind auch meist in der Lage, solche Signale selbst zu senden. Zusätzlich steuern einige elektronische Hardwarekomponenten das Verhalten weiterer Hardwarekomponenten, indem sie deren Stromversorgung stellen und kontrollieren. Hierunter fallen insbesondere Steuerungskomponenten für Motoren, Leuchten und Lautsprecher. Zur Vereinfachung wird in dieser Arbeit solch eine gesteuerte Stromversorgung als Signal betrachtet. Das Verarbeiten von Signalen bedeutet, dass diese Signale für weitere elektronische Hardwarekomponenten um einen bestimmten Faktor verstärkt werden, zu weiteren Signalen nach einem bestimmten Algorithmus umgesetzt werden oder dass die durch die Signale dargestellten Daten gespeichert oder abgerufen werden. Dabei stellen manche internen Teile einer Hardwarekomponente bestimmte Funktionen zur Verfügung, welche beispielsweise über Anschlüsse angesprochen werden können. Ebenfalls benötigen manche internen Teile einer Hardwarekomponente eine Verbindung zu einer bestimmten Hardwarefähigkeit einer anderen Komponente. Viele Hardwarefähigkeiten können nur über einen Verbund an Signal- beziehungsweise Datenleitungen bezogen werden, ein solcher Verbund wird häufig als *Port* bezeichnet. Kann die Funktion einer anderen Hardwarekomponente über einen solchen Verbund bezogen werden, so wird auch dieser Verbund als Port bezeichnet. Da Verbinder wie Stecker oder Steckbuchsen meist einen solchen Verbund darstellen, werden auch sie häufig als Port bezeichnet. Diese einheitliche Bezeichnung wird in dieser Arbeit übernommen, allerdings werden die Ports von Hardwarekomponenten in dieser Arbeit als *Hardwareport* bezeichnet um eine Verwechslung mit Ports im Kontext von Netzwerkkommunikation zu vermeiden. Auf vielen elektronischen Hardwarekomponenten sind solche Hardwarefähigkeiten mehrfach vorhanden, wodurch diese Fähigkeit mehrfach angeboten werden kann, zum Beispiel über mehrere Anschlüsse. Auch existieren in manchen Hardwarekomponenten mehrere solcher Anschlüsse, über die eine benötigte Fähigkeit von anderen Hardwarekomponenten bezogen werden kann. Dabei kann es jedoch durchaus sein, dass der Hardwareport jeder dieser Fähigkeiten durch einen anderen Schnittstellentyp zur Verfügung gestellt wird.

So bezieht der in Abbildung 2.3 gezeigte Desktopcomputer beispielsweise Strom über den Kaltgerätestecker unten links während er unter anderem über sechs USB-Anschlüsse die Fähigkeit als USB Host anbietet (rot und schwarz gefärbt) sowie über drei Anschlüsse jeweils eine serielle Schnittstelle

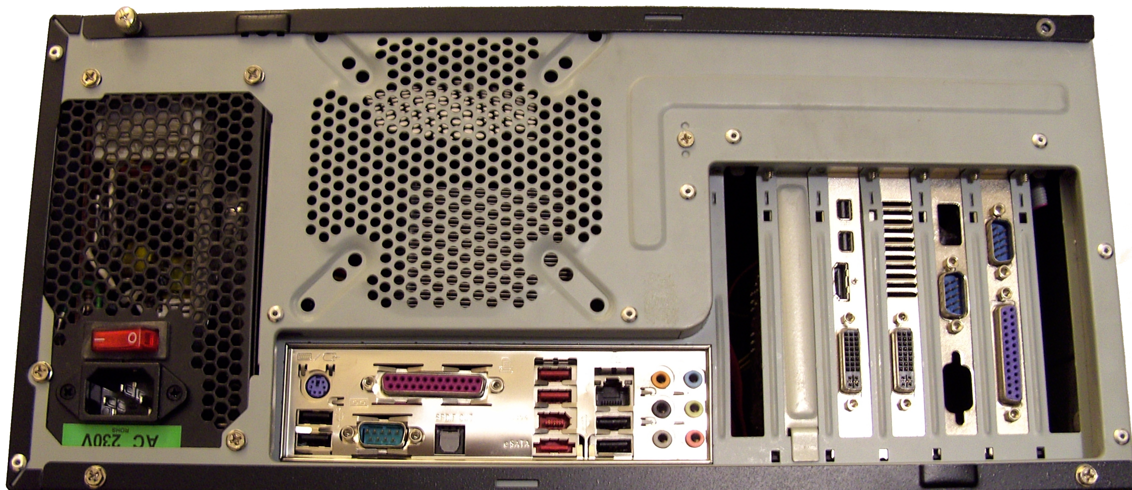


Abbildung 2.3: Die Rückseite eines Desktopcomputers zeigt verschiedene Hardwareports.

nach RS232-Standard (Blau und Türkis gefärbt). Ebenfalls kann ein Computer die Funktion einen Monitor zu unterstützen mehrfach anbieten, wobei diese Funktion von verschiedenen Hardwareports bezogen werden kann. So ist in Abbildung 2.3 zu sehen, dass diese Fähigkeit über verschiedene Arten an Anschlüssen und Übertragungsprotokollen ermöglicht wird. In diesem Beispiel wird die Fähigkeit einen Monitor anzusprechen bei den Anschlüssen nach den Standards HDMI und DisplayPort über digitale Signalübertragung angeboten. Bei den Anschlüssen nach DVI-I Standard wird jedoch zusätzlich eine analoge Signalübertragung angeboten. Dies ermöglicht beispielsweise die Verbindung mit einem Anschluss nach VGA-Standard über ein entsprechendes Kabel. Ebenfalls können mehr Monitoranschlüsse vorhanden sein als über interne Fähigkeiten unterstützt werden. Auch besteht die Möglichkeit, dass ein Anschluss eines Hardwareports über mehrere einzelne Stecker oder Buchsen verteilt ist oder eine kontaktlose Verbindung eingehen kann wie beispielsweise über Funk oder Infrarotsignale.

Zusätzlich stellen Kabel eine Spezialgruppe dar, da ihre einzige Fähigkeit darin besteht, Signale zwischen ihren Anschlüssen weiterzuleiten. Mit Standards wie USB-C [Fra17] existieren mittlerweile auch aktive Kabel auf dem Markt, jedoch dienen diese aktiven Funktionen lediglich dazu, dass angeschlossene Endgeräte untereinander ein Übertragungsprotokoll aushandeln und Eigenschaften des Kabels erfragen können. Eine Umwandlung der Daten zu einem anderen Protokoll, physikalischer Darstellung oder Ähnliches findet in einem Kabel nach USB-C-Standard jedoch nicht statt. Somit sind auch diese Kabel bezüglich der Datenübertragung zwischen Geräten passiv. Spezielle Kabel verbinden jedoch nicht alle ihrer Anschlüsse untereinander sondern können verschiedene andere Kabel in sich vereinen. Ein Beispiel hierfür sind kombinierte ”Keyboard-Video-Mouse” (KVM) Kabel wie in Abbildung 2.4 gezeigt, welche jeweils ein Kabel zum Anschluss einer Tastatur, einer Maus und eines Bildschirms an einem Computer in sich vereinen. Ebenfalls existieren Kabel, welche die Signale mehrerer Hardwareports in einem weiteren vereinen beziehungsweise die Signale eines Hardwareports separat auf weitere Hardwareports verteilen. Ein Beispiel für ein solches Kabel ist in Abbildung 2.5 gezeigt. Ebenfalls existieren aktive Adapter, welche Signale zwischen verschiedenen Protokollen oder physikalischen Formaten umwandeln, beispielsweise Videosignale nach dem digitalen Standard HDMI zu Videosignalen nach dem analogen Standard VGA.

Abbildung 2.4: Kombiniertes Kabel für den Anschluss von Tastatur, Bildschirm und Maus. Ausführung für die Verwendung zwischen Computer und KVM-Umschalter.

Abbildung 2.5: Adapterkabel, welches die Audiosignale der Anschlüsse für Stereo-Audio und Mikrofon zu einem kombinierten Anschluss für Audio und Mikrofon zusammenführt.

Computinghardware

Eine besondere Art der elektronischen Hardware stellt die Möglichkeit zur Verfügung, Software auf sich auszuführen. Solche elektronische Hardware wird in dieser Arbeit als *Computinghardware* oder *Computingkomponenten* bezeichnet.

Da konventionelle Computersysteme eine extrem hohe Anzahl an internen und externen Schnittstellen besitzen, bieten kleinere Computersysteme wie System-on-a-Chip (SoC) oder Mikrocontroller eine übersichtlichere aber dennoch umfangreiche Auswahl an Hardwarefunktionen. SoCs und Mikrocontroller können wie konventionelle PCs für sie entwickelte Softwarekomponenten oder schlicht mit einem passenden Compiler übersetzte Software für konventionelle PCs ausführen, da sie als stromsparende, langsamere und günstigere Variante zu Computersystemen entwickelt wurden. Da in vielen cyber-physischen Systemen massengefertigte general-purpose Computinghardware verwendet wird und selbst in Spezialgebieten wie Luft- und Raumfahrt häufig solche oder angepasste, aber funktional ähnliche bis gleiche Komponenten verwendet werden [Mue06] [Lep17], wird in dieser Arbeit davon ausgegangen, dass SoCs und Mikrocontroller als repräsentativ für die meiste Computinghardware in cyber-physischen Systemen angesehen werden können.

Wird ausschließlich die interne Hardware eines Mikrocontrollers oder SoC wie beispielsweise aus den verbreiteten Chipfamilien ARM und AVR, zu finden in RaspberryPi und Smartphones beziehungsweise in vielen Arduino-Modulen [Edw13], oder PIC betrachtet, so ist zu sehen, dass diese verschiedene Funktionen von internen Hardwareelementen über Hardwareports anbietet. Die

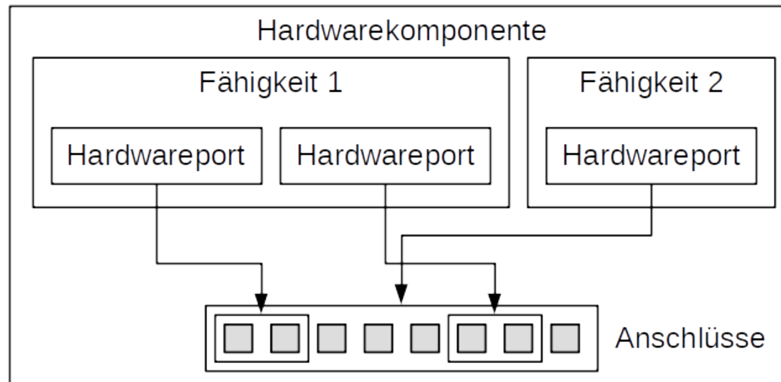


Abbildung 2.6: Hardwarefunktionen einer Computingkomponente, deren Hardwareports und ihrer möglichen Verbindungen zu Anschlüssen der Komponente schematisch dargestellt.

Hardwareports dieser Hardwareelemente sind oft direkt mit Anschlüssen des Gehäuses verbunden, können jedoch auch häufig von einer auf der Hardware ausgeführten Software konfiguriert und intern mit bestimmten Anschlüssen des Gehäuses verbunden oder von diesen getrennt werden. So können beispielsweise zwei bestimmte einzelne Anschlüsse eines Mikrocontrollers entweder gemeinsam für eine serielle Schnittstelle oder für zwei unabhängige digitale Ausgänge verwendet werden. Jeder Anschluss sollte dabei jedoch intern nur mit maximal einer Hardwarefähigkeit verbunden werden. Dadurch können manche Anschlüsse nicht konfiguriert beziehungsweise in Standardkonfiguration verbleiben oder für jeweils eine bestimmte Hardwarefähigkeit genutzt werden. Solche für verschiedene Hardwareports nutzbare Anschlüsse werden "General Purpose In/Out" (GPIO) Anschlüsse genannt. Schematisch sind solche GPIO Anschlüsse in Abbildung 2.6 dargestellt. Die dargestellte Hardwarefähigkeit "Fähigkeit 1" kann über zwei Hardwareports über je zwei bestimmte Anschlüsse bereitgestellt werden. Die dargestellten Hardwarefähigkeit "Fähigkeit 2' kann dagegen nur über einen Hardwareport bereitgestellt werden. Wird dies getan, so werden von dem Hardwareport von Fähigkeit 2 alle Anschlüsse belegt, daher kann in diesem Fall die Fähigkeit 1 nicht mehr bezogen werden.

Diese mehrfach mögliche Belegung ist gut in Abbildung 2.7 zu erkennen. Während bestimmte Anschlüsse rein für die Stromversorgung genutzt werden (GND, VCC und RAW), können sämtliche anderen Anschlüsse mit verschiedenen internen Funktionen verbunden werden. Blau hinterlegt sind dabei die digitalen Ein- und Ausgabefähigkeiten als einzelne Anschlüsse oder bis zu 8 Anschlüsse zur parallelen Datenübertragung, orange sind Interruptfunktionen, blassgelb sind Anschlüsse an welchen Signale mit Pulsweitenmodulation ausgegeben werden können. Über violett hinterlegte Anschlüsse können analoge Spannungen in digitale Werte konvertiert werden oder mit der Spannung eines anderen Anschlusses verglichen werden. Ebenfalls wird über grau hinterlegte Anschlüsse eine Datenübertragung über verschiedene serielle Protokolle unterstützt, beispielsweise nach dem RS232-Standard welcher meist als *serielle Schnittstelle* bezeichnet wird.

Neben Fähigkeiten für andere Hardwarekomponenten bietet manche Computinghardware den auf ihr ausgeführten Softwarekomponenten zusätzliche interne Fähigkeiten wie beispielsweise zur schnellen Dekomprimierung von Daten nach einem bestimmten Standard oder bestimmte Berechnungen für 3D-Darstellung an. Zusätzlich existieren Hardwarekomponenten, welche an Computinghardware angeschlossen werden können und daraufhin manche ihrer Fähigkeiten den

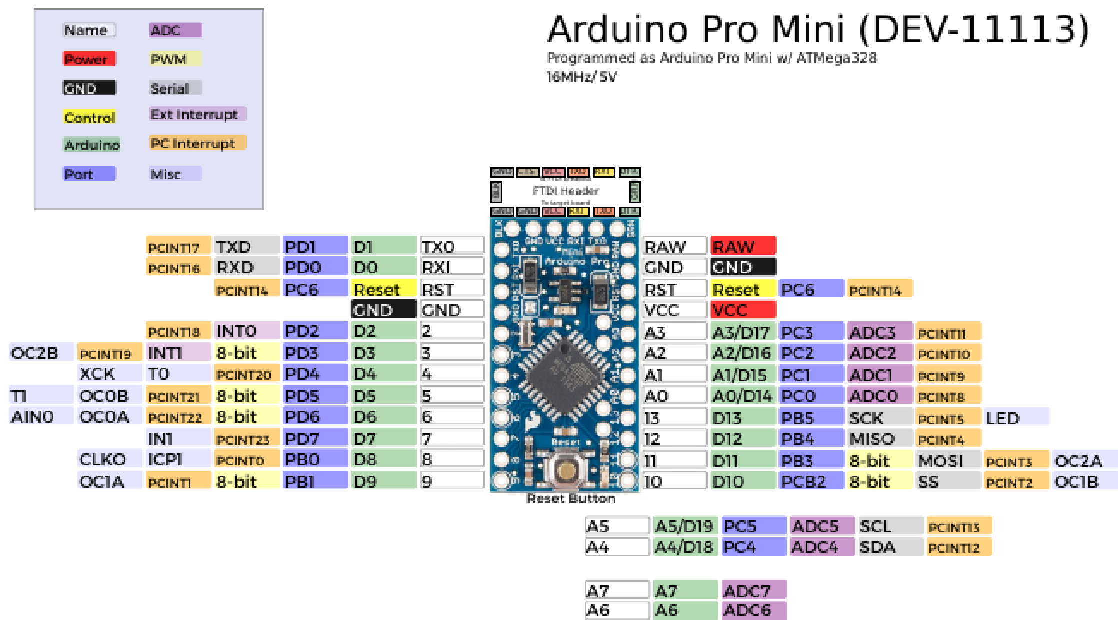


Abbildung 2.7: Ausschnitt aus dem grafischen Datenblatt des Arduino Pro Mini [SF14].

auf der Computinghardware ausgeführten Softwarekomponenten zur Verfügung stellen. Dabei kann es sich ebenfalls sowohl um weitere Hardwareanschlüsse als auch interne Fähigkeiten für Softwarekomponenten handeln.

Um Softwarekomponenten diese Hardwarefähigkeiten durch einheitliche Softwareschnittstellen zur Verfügung zu stellen, wird oftmals eine weitere Softwarekomponente benötigt, ein sogenannter *Treiber*. Zusätzlich können diese Treiber die entsprechenden Hardwarefähigkeiten verwalten und mehreren Softwarekomponenten die Nutzung einer Hardwarefähigkeit ermöglichen. Ist eine Hardwarefähigkeit nur von einer Softwarekomponente nutzbar, so können Treiber dies garantieren und jeder weiteren Softwarekomponente den Zugriff auf eine bereits verwendete Hardwarekomponente verweigern um Fehlverhalten zu vermeiden. Treibern muss im Normalfall keine Information über angeschlossene Adapter gegeben werden, da sie selbstständig nach angeschlossenen Adaptern suchen und bei Erfolg den Zugriff auf deren Funktionen anderen Softwarekomponenten zur Verfügung stellen. Manche Softwareprodukte werden als Firmare bezeichnet , dies sind Softwarekomponenten mit bestimmten Eigenschaften. Sie werden in dieser Arbeit jedoch nicht gesondert behandelt und daher auch als Softwarekomponenten bezeichnet.

Mechanische Hardware

Rein mechanische Hardwarekomponenten dienen oft als Montagemöglichkeit für oder dem Schutz von weiteren Hardwarekomponenten wie beispielsweise Gehäuse oder Chassis. Dabei sind sie auch teilweise in sich oder zueinander beweglich und können mittels angebrachter Aktoren bewegt werden. Andere mechanische Hardwarekomponenten dienen der Kraftübertragung zwischen Hardwarekomponenten, hierzu gehören unter anderem Räder, Wellen und Seilzüge.

2.5 Cyber-physische Systeme

Unter cyber-physischen Systemen versteht man Systeme welche Aktoren oder Sensoren sowie eine Komponente zum Ausführen von Software beinhalten. Dabei wird das Verhalten dieser Systeme durch auf ihnen ausgeführte Software gesteuert, überwacht und koordiniert [RLSS10]. Oftmals bestehen diese Systeme aus vielen einzelnen Komponenten, welche jeweils bestimmte Aufgaben übernehmen und miteinander kommunizieren, interagieren oder von anderen Komponenten dieses Systems gesteuert werden. In einigen Fällen kommunizieren diese Geräte auch über das Internet mit Servern, über welche sie Steuersignale bekommen oder welche rechenintensive Aufgaben übernehmen. Dies ist häufig im Bereich der Heimautomation zu sehen, da hierbei eine Kontrolle der Komponenten auch von außerhalb des nicht vom Internet aus zugänglichen Heimnetzes erwünscht ist. Umfang, Einsatzort und funktionale Ziele von Cyber-physischen Systemen können insgesamt sehr verschieden sein. Dies kann von kleinsten Messstationen für Umweltdaten über Fahrzeuge mit Assistenzsystemen oder autonomen Fähigkeiten bis zu industriellen Fertigungsstraßen oder vernetzter und automatisierter Gebäudetechnik reichen. Systeme wie moderne Stromnetze fallen jedoch auch in diese Kategorie System [MQM11].

2.6 Internet of Things

Unter dem Internet of Things (IoT) versteht man vernetzte und durch eindeutige Addressierung ansprechbare Gegenstände wie Sensoren, Aktoren und mit Radio-Frequency Identification (RFID) Tags versehene Objekte, jedoch auch netzwerkfähige Fernseher und Smartphones [AIM10] [GBMP13]. Durch diese Vernetzung können daraus Systeme entstehen, welche weitere Fähigkeiten aufweisen. Sehr viele IoT-Geräte stellen selbst cyber-physische Systeme dar, da sie durch Aktoren in ihr physisches Umfeld eingreifen können oder durch Sensoren bestimmte Eigenschaften ihres Umfelds messen können. Häufig sind diese Geräte mit Softwaresystemem im Internet verbunden und können darüber gesteuert werden und Informationen austauschen. Diese Softwaresysteme sind oftmals selbst Cloud-Anwendungen, in diesem Fall vereint ein solches IoT-System die Prinzipien von Cloud-Anwendungen und cyber-physischen Systemen [WBX14]. Ebenso vereinen diese IoT-Systeme jedoch auch die Komplexität von Cloud-Anwendungen und cyber-physischen Systemen.

3 Related Work

In diesem Kapitel werden relevante Techniken vorgestellt, welche zur Modellierung von Hardware und cyber-physischen Systemen verwendet werden können. Bei der Entwicklung von cyber-physischen Systemen werden Modelle zum Verständnis und zur Verifizierung des Verhaltens einzelner Komponenten oder des Gesamtsystems genutzt. Oft werden daher die einzelnen Komponenten eines Systems dargestellt, teils jedoch auch Abläufe und das Verhalten eines Systems modelliert. In dieser Arbeit liegt der Fokus jedoch auf der Struktur von cyber-physischen Systemen. In manchen Fällen werden auch Komponenten und zusätzlich selektiv deren Verhalten dargestellt. Im Folgenden werden Arbeiten zu diesem Thema sowie relevante Aspekte kurz vorgestellt.

3.1 Bigraph

Bigraph ist ein formales Modell, welches entwickelt wurde mit dem Ziel, sowohl direkten Lokalitätsbezug von Komponenten zueinander, als auch anderweitige Verbindungen zwischen diesen Komponenten in einem Modell darzustellen. Bei der Darstellung eines Systems mittels eines Bigraphs werden Komponenten als Knoten in zwei Graphen angegeben [Mil06]. Einer der Graphen stellt die Lokalität der Komponenten relativ zueinander dar, die gerichteten Verbindungen dieses Graphen zeigen auf, welche Komponente in welcher weiteren Komponente lokalisiert ist. Je nach Kontext ist es auch möglich, dass die Angabe der Lokalisierung auch auf oder an einer weiteren Komponente bedeutet. Dies kann beispielsweise bedeuten, dass eine Komponente an einer weiteren montiert ist oder auf dieser platziert ist. Da dieser Graph die Lokalität von Komponenten relativ zueinander angibt, wird er *Placement Graph* genannt. Der zweite Graph stellt Verbindungen zwischen den Komponenten dar, der sogenannte *Link Graph*. Dies ermöglicht die separate Darstellung von Ortsbezug und Interaktionsmöglichkeiten beziehungsweise Kommunikationsverbindungen verschiedener Komponenten untereinander. Abbildung 3.1 zeigt ein Beispiel für diese Darstellung.

3.2 Komponentenbasierte Darstellung von Hardware

Takayuki Kuroda und Aniruddha Gokhale stellen in [KG14] eine Komponentenbasierte Darstellung von Hardwarekomponenten im Bereich der IT-Infrastruktur sowie deren physikalischen Relationen untereinander vor. Hierbei werden Hardwarekomponenten wie Serverschränke, Server, Erweiterungskarten und Kabel als einzelne Komponenten dargestellt. Weiterhin wird jeder Anschluss für sowie jede Montageposition an einer weiteren Komponente als *wiringPorts* angegeben. Jeder wiringPort besitzt wiederum ein *wiringInterface*, welches vom Typ *accepts* sein kann und damit eine Montage- oder Steckposition darstellt. Alternativ kann der Typ des wiringInterface *consumes* sein. In diesem Fall stellt dies einen Stecker oder eine Montagestelle dar, welche einen wiringPort mit entsprechendem wiringInterface vom Typ accepts benötigen. Verbindungen zwischen diesen

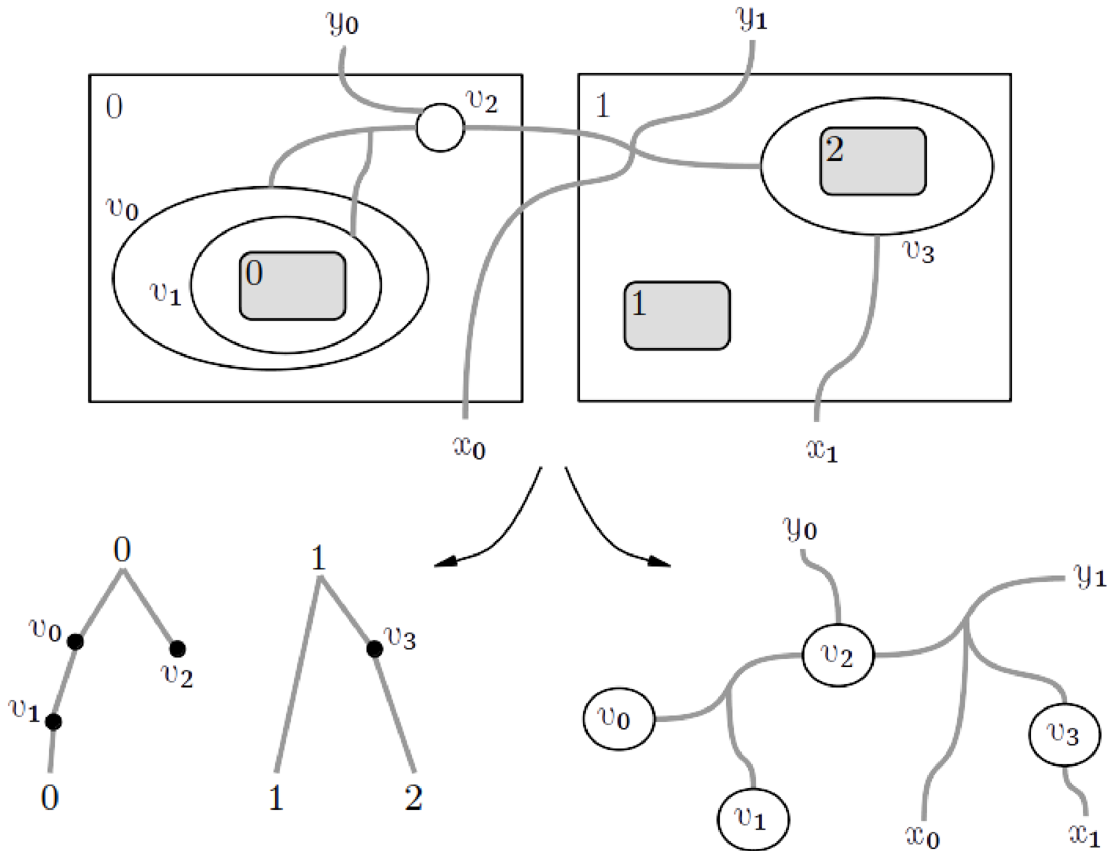


Abbildung 3.1: Beispielhafter Bigraph (oben) und dessen Auflösung in einen Place Graph (links unten) und einen Link Graph (rechts unten). Nach Milner [Mil06]

wiringPorts werden durch ein sogenanntes *wiring* dargestellt, eine Relation welche ebenfalls ein *wiringInterface* besitzt, welches den Typ der Verbindung angibt. Zusätzlich wird in dieser Arbeit ein Ansatz für das Generieren der Aufgaben für Menschen vorgestellt, sogenannte *Tasks*. Dies wird benötigt um Arbeiter beim Aufbau des entsprechenden Systems anzuleiten und um sicherzustellen dass dieses System entsprechend dem Modell aufgebaut wird. Ebenso werden in diesem Modellierungsansatz die möglichen Stadien der einzelnen Komponenten angegeben, so zum Beispiel ob sie noch verpackt sind oder ob Gehäuse geschlossen sind oder geöffnet. Der Status jeder einzelnen Komponente kann damit auch beim Ausführen von Tasks verfolgt werden. Zusätzlich können für sämtliche Komponenten *Properties* als Paare aus Schlüsselwort und Wert definiert werden. Auch bei dieser Darstellung schlagen die Verfasser vor, Vorlagen für einzelne Komponenten in einem Repository vorzuhalten um eine einfache Wiederverwendbarkeit zu ermöglichen.

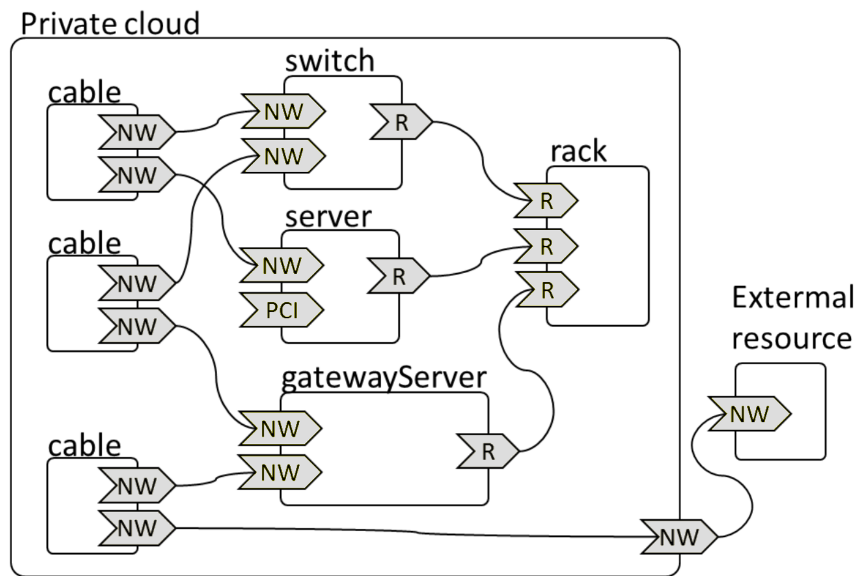


Abbildung 3.2: Darstellung zweier Server und eines Switches in einem Serverschrank (Rack) und einer externen Ressource, verbunden durch Netzwerkkabel, nach [KG14]

3.3 Schaltplan

Mit Hilfe von Schaltplänen nach DIN EN 60617 beziehungsweise IEC 60617 [DIN97] oder IEEE Standard 315-1975 [IEEE93] können sehr detailliert alle Bauteile und Komponenten eines elektronischen Systems (Schaltung) sowie deren elektrische Verbindungen untereinander grafisch dargestellt werden. Zur Darstellung von elektronischen Komponenten sowie deren Verbindungen existieren verschiedene Standards, welche jedoch sich nur leicht in der grafischen Darstellung unterscheiden und auf denselben Prinzipien aufbauen. In Schaltplänen werden sämtliche in einer Schaltung vorhandenen elektronischen Komponenten angegeben sowie deren elektrische Verbindungen untereinander. Jede elektronische Komponente ist dabei von einem bestimmten Typ und wird durch einen diesen Typ darstellendes Symbol repräsentiert. Ebenfalls besitzt jede elektronische Komponente Anschlüsse für elektrische Verbindungen. Diese Verbindungen können sich verzweigen und damit beliebig viele Anschlüssen miteinander verbinden. Die Verbindungen zusammengehörender Datenleitungen können in diesem Modell gebündelt als genannten Busse abstrahiert dargestellt werden. Ebenfalls können Teile einer Schaltung zu einer neuen Komponente zusammengefasst werden. Hierbei kann es sich durchaus um mehrere einzelne Komponenten in einem physikalischen Gehäuse handeln, welches als eigenständige Komponente in weiteren Schaltplänen verwendet wird. Dies ist beispielsweise bei integrierten Schaltungen, oft Chip oder IC genannt, der Fall. In manchen Fällen werden auch ganze Schaltungen als eine Komponente angegeben, obwohl diese nicht als integrierte Schaltungen in einem Chip, sondern aus einzelnen Bauteilen auf einer separaten Platine oder in einem separaten Gehäuse aufgebaut sind. Ebenfalls können beliebige Teile einer Schaltung als eine Komponente abstrahiert dargestellt werden um die Komplexität einer Schaltung zu reduzieren. Wird eine Schaltung oder ein Teil davon auf diese Art zu einer einzelnen Komponente abstrahiert, so werden auch in diesem Fall sämtliche zu anderen Schaltungsteilen führende Verbindungen als Anschlüsse dieser Komponente angegeben. Zusätzlich können zur besseren Übersicht elektrische Verbindungen

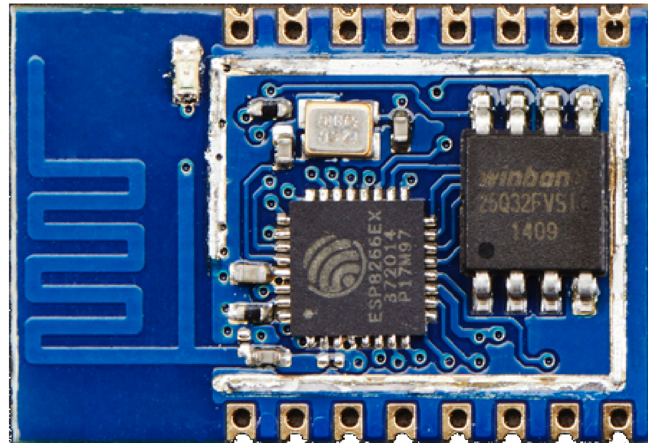


Abbildung 3.3: Modul mit Mikrocontroller ESP8266 und Speicherchip für Firmware, die Schirmung ist abgenommen, die einzelnen Komponenten sind sichtbar [AI15]

einen sogenannten Signalnamen und ein entsprechendes Schaltzeichen als virtuellen Anschluss zugeordnet bekommen. Damit können elektrische Verbindungen übersichtlicher dargestellt werden, wenn sie viele Komponenten miteinander verbinden. Statt alle Anschlüsse an dieser Signalleitung grafisch miteinander zu verbinden können diese jeweils mit dem entsprechenden Schaltzeichen verbunden werden. Zur Herstellung von elektronischen Schaltungen müssen allerdings zu jedem als eigenständige Komponente abstrahierten Schaltungsteil die Maße dessen Gehäuse beziehungsweise dessen Platine und die Platzierung und Art der Anschlüsse daran bekannt sein.

Ein typisches Beispiel für eine Schaltung, welche in weiteren Schaltungen als Komponente verwendet wird, ist das in Abbildung 3.3 abgebildete Modul mit Mikrocontroller ESP8266 sowie einem Speicherchip für Software und Daten. Dieses Modul kann durch den in Abbildung 3.4 gezeigten Schaltplan dargestellt werden. Sowohl auf dem Foto als auch auf dem Schaltplan sind die drei größten Bauteile SoC (mittig), Speicher (rechts) und Quarz (oben) sowie viele elektrische Verbindungen gut zu erkennen. Auf dem Schaltplan ist ebenfalls die Abstraktion von Verbindungen durch Schaltzeichen zu sehen, beispielsweise werden die Leitungen für die Spannungsversorgung, Masse ("GND") und 3,3V ("VDD33"), auf diese Weise abstrahiert. Wird dieses Modul nun in weiteren Modulen als einzelne Komponente verbaut, wie in Abbildung 3.6 zu sehen, so bietet es sich an in dem Schaltplan dieses größeren Moduls das kleinere als eine Komponente darzustellen und seine internen Komponenten damit nicht darzustellen. Ein einfaches Beispiel hierfür ist in Abbildung 3.5 zu sehen, dieser Schaltplan stellt die minimal nötige Konfiguration einer Schaltung dar um das Modul betreiben zu können.

3.4 Blockdiagramm

Blockdiagramme [Blu02] sind eine einfache Darstellungsmöglichkeit von Abläufen und Zusammenhängen zwischen Komponenten und werden häufig genutzt, um elektronische Komponenten nur anhand ihrer Funktion als abstrakte Blöcke darzustellen. Sämtliche Informationen darüber, wie sie

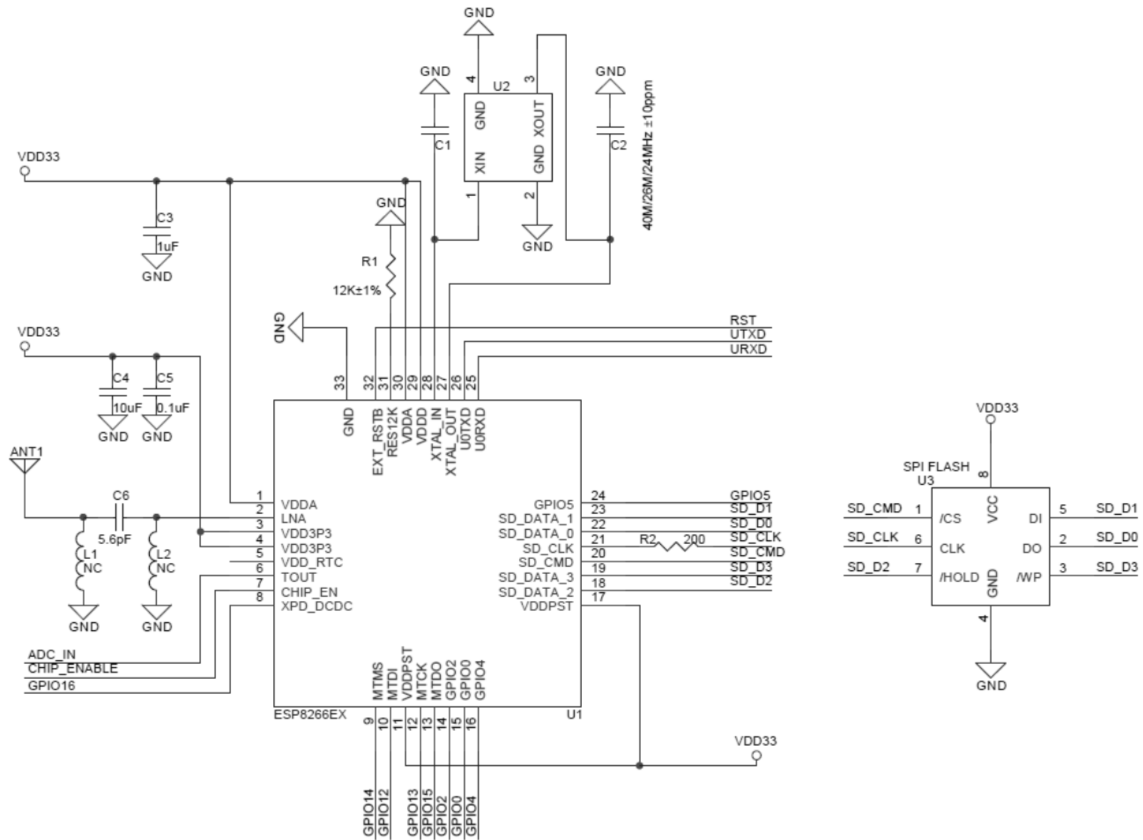


Abbildung 3.4: Schaltplan des in Abbildung 3.3 gezeigten Moduls [ITEAD16]

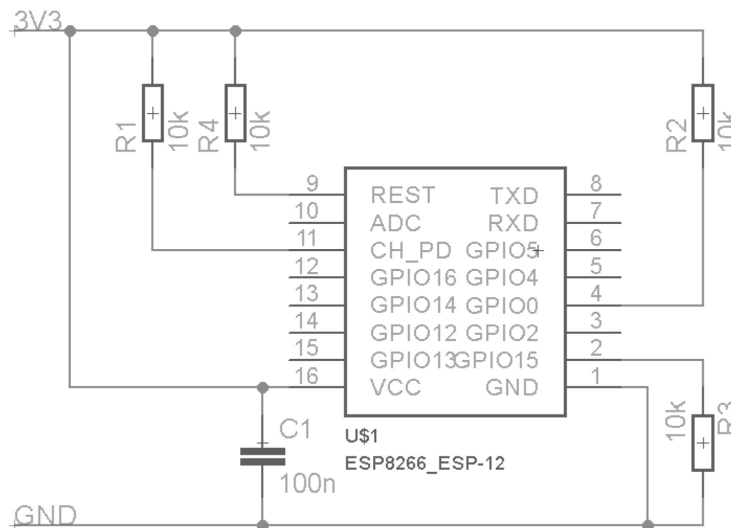


Abbildung 3.5: Schaltsymbol des Moduls aus Abbildung 3.3 in einem Schaltplan [ESPA18]

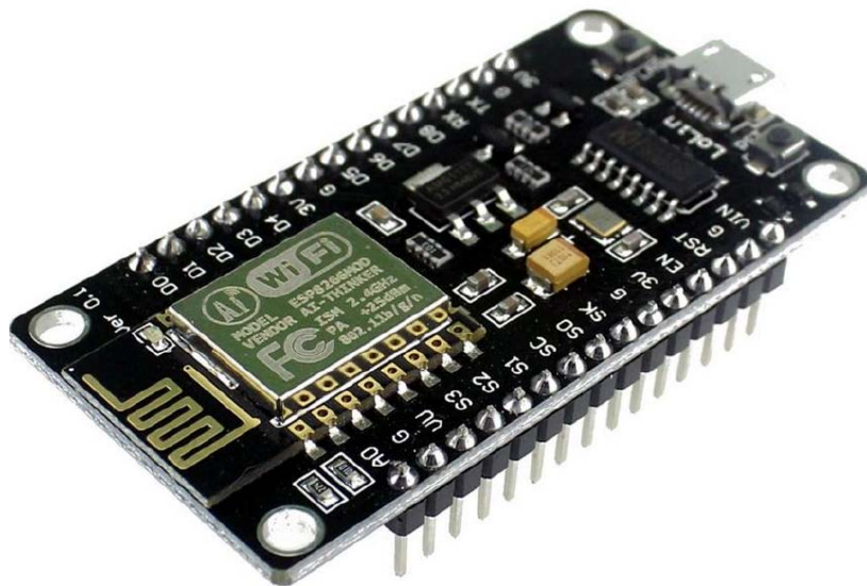


Abbildung 3.6: Modul aus Abbildung 3.3 verbaut auf weiterer Schaltung [Handson17]

diese Funktion erfüllen, werden dabei ausgeblendet. Da diese Art der Darstellung von elektronischen Systemen nicht genormt ist, haben Entwickler hierbei viel Freiheit in der Darstellung von Systemen. Zusätzlich zu den elektrischen Komponenten werden verbindende Datenleitungen zwischen diesen Komponenten angezeigt, im Normalfall allerdings ebenfalls nur abstrahiert, Details wie die Anzahl an Daten- und Signalleitungen bei oder das genutzte Protokoll werden nicht angegeben. Ebenfalls werden hierbei oft nur für das Verständnis der Funktion des Systems relevante Komponenten und Verbindungen angezeigt. So werden zum Beispiel bei der Darstellung von Mikrocontrollern oder SoCs die Stromversorgung und ihre Verbindungen nicht dargestellt. Diese Art der Darstellung wird meist als abstrakter Überblick über die Funktionen integrierter Schaltungen oder elektronischer Systeme genutzt, bedingt auch, um ihre Abhängigkeiten beziehungsweise Zusammenhänge darzustellen. Ein detailliertes Beispiel einer solchen Darstellung ist in Abbildung 3.7 zu sehen, hier werden lediglich die Stromversorgung und ihre Verbindungen zu den einzelnen Komponenten nicht angezeigt. An dieser Stelle ist auch gut zu sehen, dass verschiedene interne Komponenten auf dieselben Anschlüsse zugreifen können. Unter welchen Umständen diese Zugriffsmöglichkeiten genutzt werden und wie genau dies technisch umgesetzt wird, ist jedoch nicht dargestellt. So greifen beispielsweise sowohl die Komponente des Analog-Digital-Wandlers (ADC) als auch die Komponente mit Treiber- und Pufferbausteinen für den parallelen Port C auf die Anschlüsse PC0 bis PC6 zu. Auch werden teilweise nur ausgewählte Daten- oder Signalleitungen angegeben, beispielsweise wenn der Fokus auf den Komponenten oder Teilen eines Systems liegt und nicht auf deren Interaktion, siehe beispielsweise Abbildung 3.8 mit dem Fokus auf der Funkschnittstelle für WLAN. Auch werden teilweise Komponenten nach Funktionstypen in übergeordnete Komponenten gruppiert, auch dies ist in Abbildung 3.8 zu sehen.

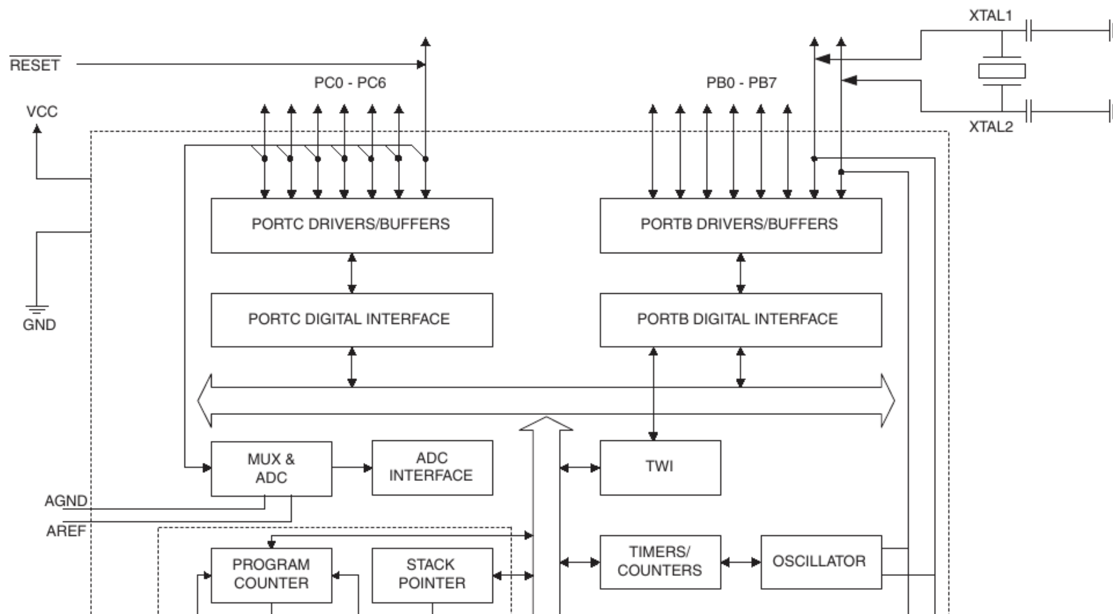


Abbildung 3.7: Ausschnitt aus dem Blockdiagramm des Mikrocontrollers ATMega8. Aus [MC13]

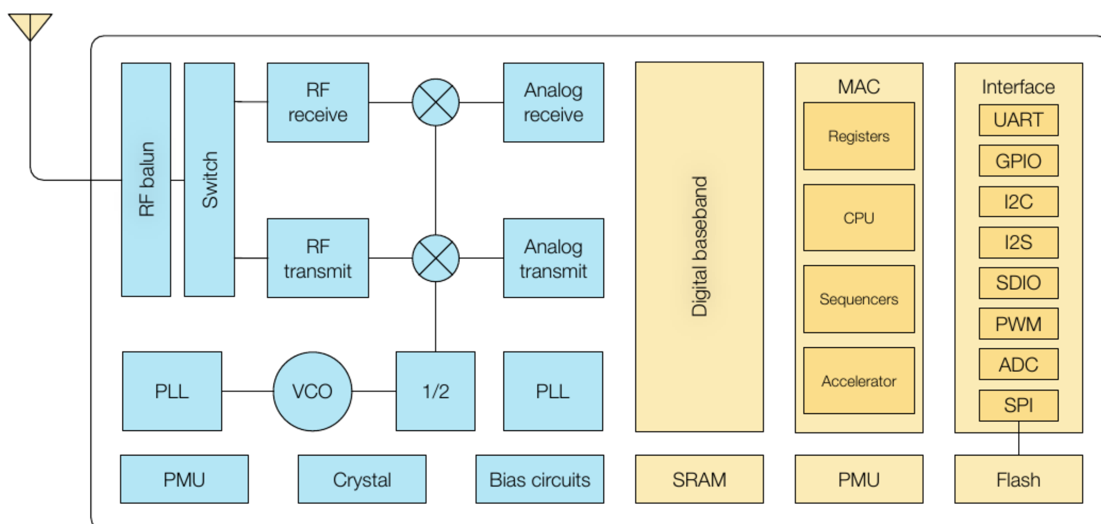


Abbildung 3.8: Blockdiagramm des in Abbildungen 3.4 und 3.3 gezeigten Moduls. Ausschnitt aus dem Datenblatt [ESP18]

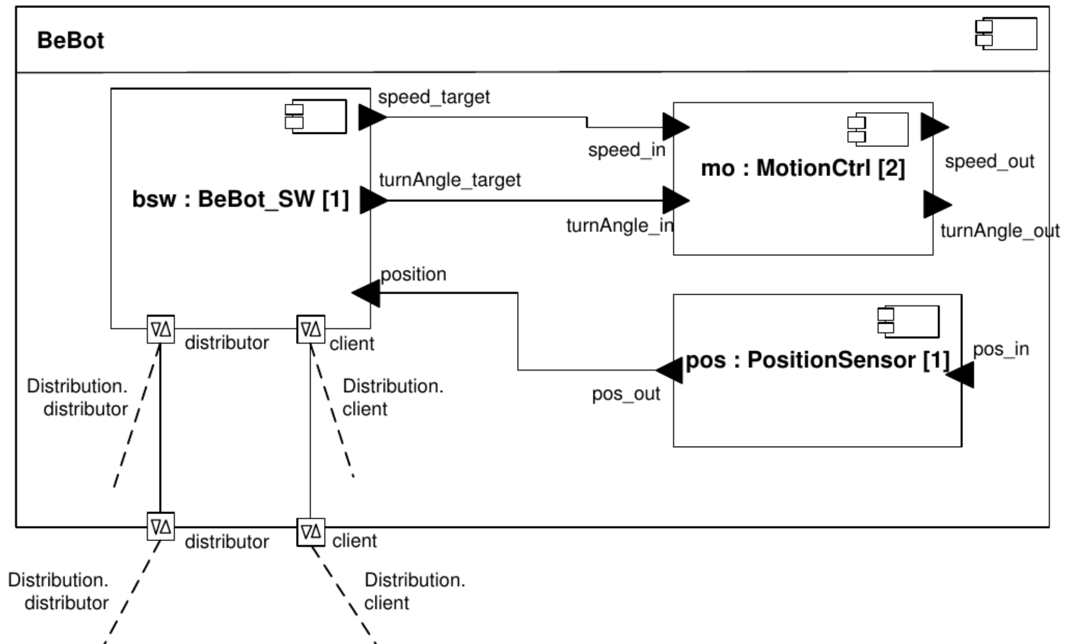


Abbildung 3.9: Beispielhafte Darstellung eines in MechatronicUML modellierten Systems [BDG+14]

3.5 MechatronicUML

MechatronicUML ist ein modellgetriebener Ansatz um Software für mechatronische Systeme zu entwickeln. Dabei ist eine Modellierungssprache für Softwarekomponenten von cyber-physischen Systemen Teil von MechatronicUML. Zusätzlich ermöglicht MechatronicUML die formale Verifikation des Verhaltens eines darin modellierten Systems. In dieser Modellierungssprache werden basierend auf den Konzepten der Unified Modelling Language (UML) und einer komponentenbasierten Darstellungsweise Softwarekomponenten und Kommunikationskanäle eines solchen Systems modelliert. Hierbei besitzt jede Komponente sogenannte Ports, welche je nach Typ eingehende, ausgehende oder unidirektionale Kommunikations- beziehungsweise Signalverbindungen ermöglichen. Hierbei wird bei MechatronicUML zusätzlich unterschieden, ob dieses Signal dauerhaft gesendet wird beziehungsweise empfangen werden muss oder ob lediglich diskrete Signale gesendet beziehungsweise erwartet werden. Im ersten Fall wird ein solcher Port als *continuous Port* bezeichnet, im zweiten Fall als *discrete Port*. Außerdem können Ports als zwingend (*mandatory*) oder optional zu nutzende Ports dargestellt werden. Ein Beispiel für die Darstellung eines solchen Systems zeigt Abbildung 3.9. In dieser Abbildung ist ein Robotersystem dargestellt, dessen zentrale Komponente einer Komponente zum Bewegen des Roboters dauerhaft Signale sendet. Zusätzlich empfängt diese zentrale Komponente dauerhaft ein Signal von einem Positionssensor. Ebenfalls stellt sie zwei optionale bidirektionale Ports, über welche sie mit weiteren solchen Komponenten in anderen Robotern kommunizieren kann. Hardwarekomponenten sowie die für Kommunikationsverbindungen nötigen physikalischen Verbindungen zwischen diesen werden in MechatronicUML nicht dargestellt.

4 Fallstudie autonomes Fahrzeug

Um die Grundlagen und Herausforderungen bei der Modellierung eines cyber-physischen Systems besser zu verstehen, wurde im Rahmen dieser Arbeit ein einfaches autonomes Fahrzeug entwickelt. Zusätzlich dient dieses dazu, anhand eines einfachen und dennoch repräsentativen Beispiels die Praktikabilität des in dieser Arbeit vorgeschlagenen Modellierungskonzeptes zu überprüfen. Das funktionale Ziel des autonomen Fahrzeugs ist das Suchen, Erkennen und sich Nähern an eine bestimmte Kombination geometrischer Figuren, im Detail ein Quadrat in einem Kreis. Dieses Szenario ist zur Veranschaulichung in Abbildung 4.1 grob skizziert. Zusätzlich soll das Fahrzeug den Bereich vor und hinter sich rudimentär auf Hindernisse überwachen und selbstständig seine Fahrt unterbrechen, auch im Falle anderweitig lautender Befehle der steuernden Softwarekomponente. Um möglichst einfach zu sein und dennoch als repräsentativ gelten zu können, besteht das Fahrzeug aus konventionellen Komponenten, welche in IoT-Geräten genutzt werden oder in leistungsfähigerer Form in cyber-physischen Systemen wie autonomen oder semiautonomen Fahrzeugen eingesetzt werden. Im folgenden Absatz werden die Hardwarekomponenten dieses Fahrzeugs sowie ihre elektrischen Verbindungen untereinander vorgestellt. Danach werden in Abschnitt 4.2 die darauf installierten Softwarekomponenten sowie deren Funktion und Interaktion beschrieben.

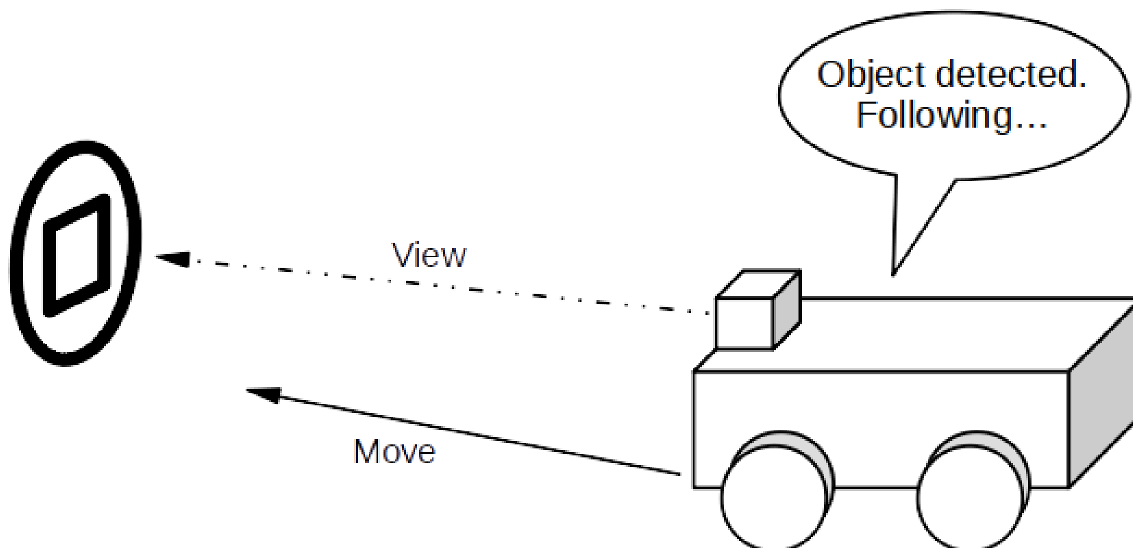


Abbildung 4.1: Skizzenhafte Darstellung des autonomen Fahrzeugs und seines primären Verhaltens.

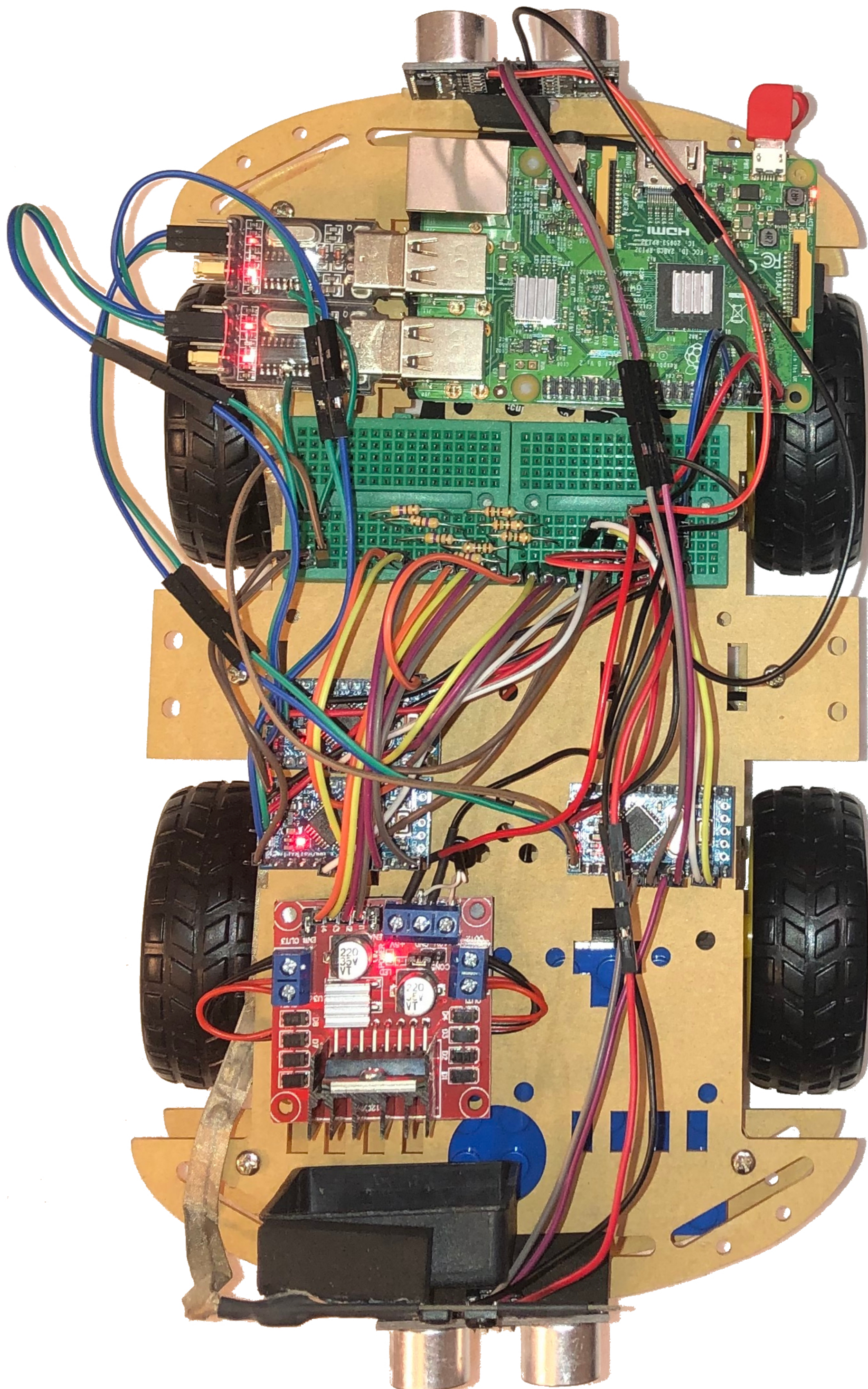


Abbildung 4.2: Draufsicht des Fahrzeugs, die meisten Komponenten sind sichtbar.

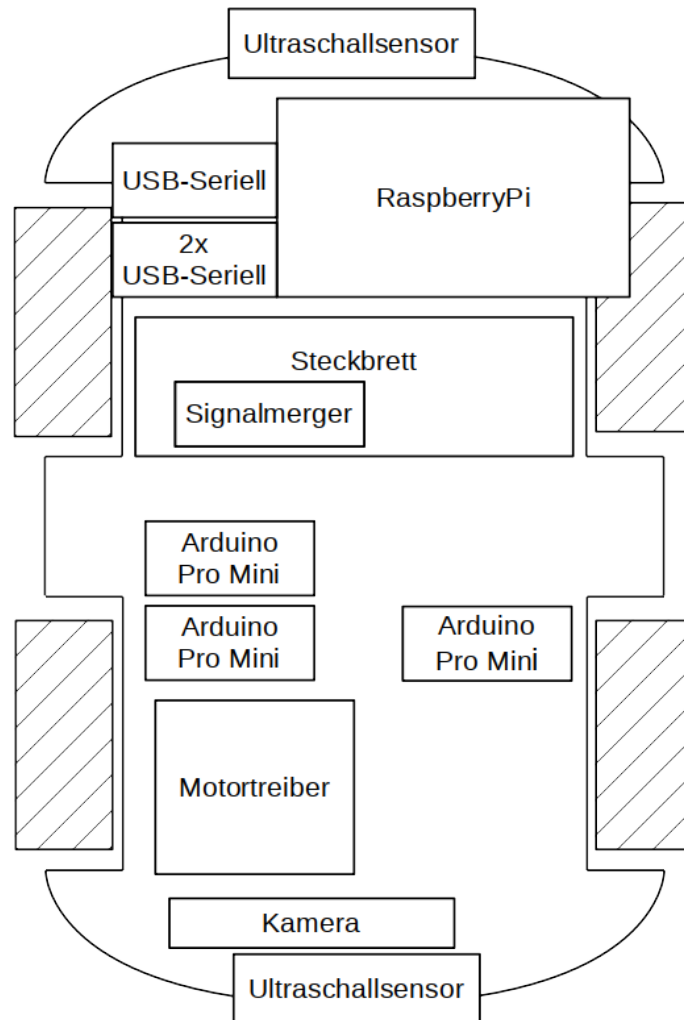


Abbildung 4.3: Draufsicht des Fahrzeugs als schematische Darstellung der Komponenten mit Bezeichnung.

4.1 Hardwarekomponenten

Um die Funktion zum Detektieren und Verfolgen eines Objektes zu ermöglichen, müssen auf der Hardware des Fahrzeugs Softwarekomponenten für diese Funktion ausgeführt werden. Daher muss die Hardware neben der typischen Hardware eines Fahrzeuges auch Computinghardware anbieten, welche es ermöglicht, auf ihr Software für die Objekterkennung und Steuerung des Fahrzeugs bereitzustellen. Um Kollisionen zu vermeiden müssen die Bereiche vor und hinter dem Fahrzeug durch Abstandssensoren überwacht werden und eine entsprechende Weiterfahrt trotz anderslautender Befehle der Steuersoftware unterbunden werden.

Als physische Plattform dient ein offenes Chassis mit vier nicht lenkbaren Rädern, welche jeweils von einem Gleichstrommotor angetrieben werden. Die Motoren jeder Seite sind jeweils über ein Y-Kabel miteinander Verbunden.

Als Computingplattform für die Software, welche die zentralen Funktionen übernimmt, dient ein konventioneller Kleinstrechner vom Typ RaspberryPi Version 3B (in Abbildung 4.2 oben rechts zu sehen). Dieser ist mit einem 64Bit ARM-Prozessor mit vier Rechenkernen ausgestattet und besitzt viele verbreitete Anschlussmöglichkeiten wie USB, HDMI und 3,5mm Stereo Soundausgang. Zusätzlich bietet dieser Kleinstrechner verschiedene GPIO Anschlüsse, welche einzeln angesteuert werden können oder bestimmte Protokolle unterstützen. An der Vorderseite des Fahrzeugs ist eine Kamera befestigt wie in Abbildung 4.2 unten mittig zu sehen ist. Diese Kamera ist über einen USB-Anschluss an das RaspberryPi angeschlossen.

Die Motorsteuerung wird über zwei redundante Arduino Pro Mini realisiert, um mögliche Herausforderungen beim Modellieren von redundanten Systemteilen aufzuzeigen. Diese beiden Arduino-Module sind über Adapter von USB zu serieller Schnittstelle mit dem RaspberryPi verbunden, über diese Verbindungen können auch Softwarekomponenten durch sogenanntes Flashen auf die Arduino-Module installiert werden. Ebenso kann Software auf dem RaspberryPi über diese seriellen Schnittstellen mit der Firmware der Arduino-Module kommunizieren. Die beiden Arduino-Module sind wiederum über einen primitiven Signalmixer mit einer einfachen Treiberschaltung verbunden, welche über zwei Ausgänge jeweils die beiden Motoren einer Seite des Fahrzeugs betreibt. Da der Chip der Arduino-Module keine Ausgabe von Analogsignalen unterstützt, müssen die Signale an die Treiberschaltung durch Pulsweitenmodulation (PWM) erfolgen. Dabei wird der Ausgangsanschluss mit einer festen Frequenz an und wieder aus geschaltet, durch Verändern des Verhältnisses der Zeit zwischen an (”High’, 5V) und aus (”Low”, 0V) kann hierbei der Strom durch die Motoren gesteuert werden. Ebenso kann die Drehrichtung der Motoren durch Wechseln des durch PWM angesteuerten Anschluss geändert werden. Mit der passenden Firmware auf mindestens einem der Arduino-Module kann das Fahrzeug dadurch ähnlich wie ein Kettenfahrzeug gesteuert werden.

Um eine sicherheitskritische Komponente einzubauen, ist das Fahrzeug mit je einem auf Ultraschall basierenden Distanzsensor an der Vorderseite sowie Rückseite ausgestattet. Ein weiteres Arduino-Modul ist mit diesen Sensoren verbunden, damit darauf ausgeführte Software die Daten der Sensoren auslesen kann. Dieses Arduino-Modul ist ebenfalls über einen Adapter von USB zu serieller Schnittstelle mit dem RaspberryPi verbunden. Zusätzlich ist dieses Arduino-Modul über zwei Signalleitungen mit jeweils einem als Interrupt-Eingang nutzbaren Anschluss der für die Ansteuerung der Motoren genutzten Arduino-Module verbunden.

Als Stromversorgung dienen zwei Batterien mit USB-Anschluss, wovon eine das RaspberryPi, die Arduino-Module und die Treiberschaltung der Motoren mit Strom versorgt. Die zweite Batterie wird zum Betrieb der Motoren genutzt und ist dazu an einen separaten Anschluss der Treiberschaltung der Motoren angeschlossen. Eine getrennte Stromversorgung ist nötig, da beim Betrieb der Motoren Spannungsschwankungen auftreten, welche einen zuverlässigen Betrieb des RaspberryPi und der Arduino-Module unmöglich machen.

4.2 Softwarekomponenten

Auf dem RaspberryPi ist das Betriebssystem Raspbian installiert, darauf wiederum ein Interpreter für die Skriptsprache Python sowie mit OpenCV eine in Bibliothek mit Funktionen zur Bilderkennung für in Python und C geschriebene Programme. Zur Steuerung des Fahrzeugs wird ein Python-Skript verwendet, welches die Bilddaten der Kamera mit Hilfe von OpenCV verarbeitet und entsprechende

Steuerbefehle für die Motoren an die Firmware eines der zur Motorsteuerung genutzten Arduino-Module sendet. Bei diesen Befehlen wird jeweils die gewünschte Seite der anzusteuernden Motoren sowie der zu setzende Wert des PWM-Signals angegeben. Ist der im Steuerbefehl gesendete Wert negativ, so sind die Motoren der angegebenen Seite so anzusteuern, dass sie das Fahrzeug rückwärts bewegen. Dabei ist der Betrag dieses Wertes der Wert, welcher der Hardwarefunktion für das PWM-Signal am entsprechenden Anschluss übergeben werden soll.

Auf den beiden zur Motorsteuerung genutzten Arduino-Modulen ist eine Firmware installiert, welche über die serielle Schnittstelle Befehle zum Ansteuern der Motoren erwartet und entsprechend an den Ausgängen zur Treiberschaltung der Motoren ein PWM-Signal ausgibt. Weiterhin lösen Signale an zwei Anschlüssen bei dieser Firmware Interrupts aus, wodurch jeweils eine Vorwärts- beziehungsweise Rückwärtsbewegung der Motoren über die Dauer des Signals unterbunden wird.

Auf dem dritten Arduino Pro Mini werden von einer Firmware die Signale der Ultraschallsensoren ausgewertet. Dabei wird in regelmäßigen Abständen ein Signal an den entsprechenden Sensor gesendet. Dieser sendet etwas später ein Signal zurück, dessen Dauer dabei dem Abstand bis zum nächsten detektierten Objekt entspricht. Wurde kein Objekt detektiert entspricht die Dauer des Signals dem maximalen Abstand, in welchem der Sensor Objekte detektieren kann. Detektiert ein Sensor ein Objekt näher als in einer vorgegebenen Distanz, so sendet die Firmware ein Signal über die entsprechende Signalleitung an die Interrupt-Eingänge der beiden für die Motorsteuerung zuständigen Arduino-Module. Damit soll eine weitere Fahrt in Richtung des detektierten Objekts trotz möglicherweise anderweitig lautender Befehle des Python-Skripts zur Steuerung verhindert werden. Dennoch kann damit eine Fahrt in die entgegengesetzte Richtung noch möglich bleiben, da für jede Fahrtrichtung ein separates Signal an die anderen Arduino-Module gesendet wird.

5 Modellieren von cyber-physischen Systemen in TOSCA

TOSCA wurde mit der Zielsetzung entwickelt, Anwendungen in einer Cloudumgebung zu modellieren, welche daraufhin bereitgestellt und zur Laufzeit verwaltet werden können. Da die Prinzipien von Cloud-Systemen mit dem Ziel entwickelt wurden, das Management von Hardwareressourcen an Cloud-Anbieter auszulagern, bestand bei der Erstellung des TOSCA-Standards keine Notwendigkeit Hardwarekomponenten zu modellieren. Durch fortschreitende Automation in Produktion und Gebäudesteuerung steigt die Verbreitung cyber-physischer Systeme immer weiter. Diese Systeme kommunizieren häufig mit Cloud-Systemen oder werden direkt von diesen gesteuert, wodurch ein hybrides Gesamtsystem entsteht. Dadurch stellt sich die Frage ob und wie solche hybriden Systeme in TOSCA modelliert werden können. Darauf aufbauend stellt sich auch die Frage nach Möglichkeiten für Modellierung, Deployment und Management solcher Systeme in TOSCA.

5.1 Problemstellung

Hardwarekomponenten können durch ihre physischen Eigenschaften nicht gleich behandelt werden wie Softwarekomponenten. Ob dies auch gilt, wenn ein cyber-physisches System auf Basis von TOSCA beschrieben werden soll mit dem Ziel, es anhand dieser Beschreibung bereitstellen zu können, ist zu klären. Daher muss evaluiert werden, auf welche Art und Weise Hardwarekomponenten sowie deren Verbindungen untereinander und zu Softwarekomponenten in TOSCA modelliert werden können. Dabei soll auch darauf eingegangen werden, welche Anforderungen dabei an die Modellierung gestellt werden um ein Deployment von Hardware durch die im Modell dargestellten Informationen zu ermöglichen. Entsprechende Anforderungen sowie nötige Erweiterungen oder Änderungen werden in den folgenden Abschnitten beschrieben und erklärt.

5.2 Anforderungen

In diesem Abschnitt werden die Anforderungen an den in dieser Arbeit vorgestellten Lösungsansatz zum Modellieren von Hardwarekomponenten in TOSCA vorgestellt und deren Hintergründe erklärt. Da eine Modellierung von Hardwarekomponenten in TOSCA zusätzlich zur bisher möglichen Modellierung von Softwarekomponenten möglich sein soll, wird angestrebt, **keine Änderungen am bisherigen Standard** vorzuschlagen, sondern diesen zu erweitern. Zusätzlich ermöglicht dies eine einfachere technische Umsetzung der Vorschläge durch Erweiterungen existierender Implementierungen. Um eine einfache Modellierung zu ermöglichen, **soll das Konzept von TOSCA ganzheitlich bleiben**, auch bei der Modellierung von Hardwarekomponenten. Dabei sollen möglichst sämtliche Bedeutungen und Eigenschaften der in TOSCA spezifizierten Elemente zur

Modellierung beibehalten werden. Grundsätzlich gilt, dass sämtliche Komponenten eines Systems, welche bereitgestellt und verwaltet werden sollen, dafür auch modelliert werden müssen. Daraus folgt, dass mit dem Lösungsansatz erstellte Modelle **alle Hardwarekomponenten eines Systems darstellen** müssen. Wie schon bei der Vorstellung von Schaltplänen gezeigt, ist es bei der Darstellung eines Systems dienlich, wenn entsprechende Modelle **nur relevante Informationen darstellen**. Damit sollten zusätzlich zu den für das Deployment und Manangement nötigen Informationen nur die für Nutzer relevanten oder hilfreichen Daten in die Modellierung aufgenommen werden.

5.3 Modellierung von Hardwarekomponenten in TOSCA

Soll es ermöglicht werden, in TOSCA cyber-physische Systems zu modellieren, so müssen deren einzelnen Komponenten beschrieben werden können. Da diese untereinander verbunden sind oder Fähigkeiten jeweils anderer Komponenten nutzen, muss auch dies dargestellt werden. Hardwarekomponenten sind einzelne Komponenten eines Systems, welche zum Aufbau eines Systems bereitgestellt werden müssen. Ebenfalls können sie mit weiteren Hardwarekomponenten verbunden werden. Zusätzlich können auf, an oder in ihnen teils weitere Hardware- oder Softwarekomponenten angebracht, gespeichert oder ausgeführt werden. Daher müssen Hardwarekomponenten bei einer Darstellung in TOSCA wie Softwarekomponenten als einzelne Komponenten dargestellt werden. Weiterhin bieten viele Hardwarekomponenten bestimmte Fähigkeiten an, ebenso benötigen viele Hardwarekomponenten bestimmte Fähigkeiten anderer Hardwarekomponenten um ihrer Funktion nachkommen zu können. Bei vielen Hardwarekomponenten ist auch beides der Fall. In TOSCA ist bereits die Möglichkeit definiert, für Komponententypen (Node Types) sogenannte *Capabilities* und *Requirements* zu definieren. Eine typische graphische Darstellung hiervon ist in Abbildung 5.1 gezeigt, darauf bauen weitere Darstellungen auf. Diese Capabilities und Requirements können auch für die Darstellung von Hardwarefähigkeiten einer Hardwarekomponente genutzt werden beziehungsweise für das Bedürfnis einer solchen Fähigkeit, welche eine andere Hardwarekomponente über eine Verbindung stellen kann. Dies können sowohl Fähigkeiten für Hardwarekomponenten wie Kommunikations-, Stromversorgungs- oder Schnittstellen für einzelne Signale sein, als auch Capabilities für Softwarekomponenten, welche auf der entsprechenden Hardwarekomponente gehostet sind oder eine ihrer Hardwarefähigkeiten wie Hardwareports oder Hardwarebeschleuniger nutzen wollen. Hardwarekomponenten bieten häufig Hardwarefähigkeiten eines Typs mehrfach an, jedoch sind dabei die Details für den Zugriff auf diese, wie zum Beispiel die ID, für jede einzelne Fähigkeit unterschiedlich. Daher bietet es sich an, lediglich die Typen der Fähigkeiten einer Hardwarekomponente als Capabilities anzugeben und für jede einzelne angebotene Fähigkeit dieses Typs die Details in einem Constraint dieser Capability darzustellen. Selbiges gilt auch für die Anforderungen von Hardwarekomponenten, welche als Requirements dargestellt werden können. Da alle Hardwarekomponenten ohne das Vorhandensein von Software existieren und ihre Funktionen anbieten können, sind Requirements von Hardwarekomponenten nur von Capabilities weiterer Hardwarekomponenten zu bedienen.

Damit das automatische Erkennen von Hardwarekomponenten als solche für jegliche Algorithmen wesentlich vereinfacht wird, wird an dieser Stelle ein neuer Wurzel-Node Type für Hardwarekomponenten namens *HardwareNode* vorgeschlagen. Mit Hardwarekomponenten kann sowohl durch physische als auch über direkt oder indirekt verbundene Softwarekomponenten interagiert werden. Hierzu gehören unter anderem das Deployment, welches ausschließlich durch physische Aktionen erfolgen kann, sowie Konfigurieren der Komponente und Installieren oder Konfigurieren

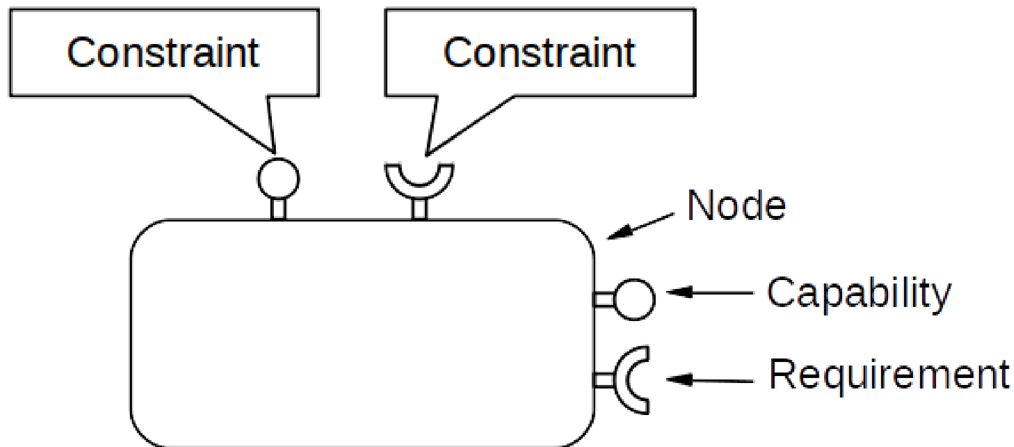


Abbildung 5.1: Grafische Darstellung einer in TOSCA beschriebenen Node mit Capabilities und Requirements sowie zu diesen zugehöriger Constraints.

von Softwarekomponenten, welche darauf installiert werden sollen oder dies bereits sind. Daher müssen Hardwarekomponenten ein generisches physisches Management Interface besitzen und zusätzlich unter Umständen weitere Management Interfaces für Software. Details dieser Management Interfaces werden in dieser Arbeit nicht behandelt und können Gegenstand weiterer Forschung zu diesem Thema sein. Wird eine Hardwarekomponente zusammen mit einer Softwarekomponente wie einem Betriebssystem als eine Komponente dargestellt, so kann diese hybride Komponente ebenfalls zusätzlich zu von Software ausführbaren Management Interfaces ein physisch ansprechbares anbieten. Auch dies spricht für ein generisches physisches Management Interface, welches in einem solchen Fall verwendet werden kann.

Bei manchen Hardwarekomponenten können bestimmte Parameter zum Empfang von Signalen konfiguriert werden, wie beispielsweise Adressen in bestimmten Netzwerken wie Bussystemen nach dem Standard DMX512 [ESTA18], einem Protokoll für Bühnentechnik, oder Funkgesteuerte Steckdosen in der Heimautomation. Solche Parameter sollten beim Deployment der entsprechenden Hardwarekomponente eingestellt werden. Daher bietet es sich an, sie wie Parameter zur initialen Konfiguration bei Softwarekomponenten zu behandeln und sie als Properties der Komponente anzugeben. Sie müssen jedoch spätestens vor dem Erstellen einer Datenverbindung zu dieser Komponente eingestellt werden, da hierfür die Adresse der Hardwarekomponente benötigt wird.

5.3.1 Hardwareports

Oft ist eine Hardwarefähigkeit mehrfach auf einem Bauteil vorhanden, wodurch deren Fähigkeit mehrfach angeboten werden kann oder über welche eine benötigte Fähigkeit anderer Hardwarekomponenten bezogen werden kann. Im Vergleich zu Softwarekomponenten, welche über Netzwerke praktisch unendlich viele eingehende Verbindungen über einen Port annehmen können, können Hardwarekomponenten häufig nur eine ein- oder ausgehende Verbindung über einen ihrer Hardwareports anbieten. Dabei kann es jedoch durchaus sein, dass die Hardwareports jeder dieser Fähigkeiten durch einen anderen Anschlusstyp zur Verfügung gestellt werden. Beispielsweise kann bei einem Computer ein Audioport durch eine Buchse für 3,5mm Klinkenstecker zur Verfügung gestellt werden, ein weiterer jedoch durch einen Anschluss für Lichtwellenleiter, wie schon in

Abbildung 2.3 gezeigt wurde. Alle drei Anschlüsse haben die Funktion, Audiosignale auszugeben, bieten dies jedoch über unterschiedliche Anschlussarten an. Beispielsweise sind in Abbildung 2.3 fünf der sechs Audioanschlüsse für gängige 3,5mm Klinkenstecker rechts der Mitte sowie der optische Anschluss unten links der Mitte (Schwarzes Quadrat mit grauer Schutzklappe innen) allesamt Audioausgänge, jedoch mit verschiedenen Anschlüssen. Ebenso besteht die Möglichkeit, dass der Anschluss eines Hardwareports über mehrere einzelne Stecker oder Buchsen verteilt ist, die fünf genannten Audioanschlüsse für 3,5mm Klinkenstecker können beispielsweise zusammen als ein Surroundsound-Ausgang genutzt werden. Auch kann ein Anschluss eine oder gar mehrere kontaktlose Verbindungen eingehen wie beispielsweise über Funk oder Infrarotsignale. Daraus folgt, dass diese Informationen über die zugehörigen Hardwareports als Teil der entsprechenden Capability beziehungsweise des Requirements dargestellt werden müssen. Da diese Informationen größtenteils Einschränkungen der Anzahl an parallel nutzbaren Fähigkeiten beziehungsweise der nutzbaren Anschlüsse darstellen, liegt es nahe, sie in den in TOSCA vorgesehenen Constraints von Capabilities und Requirements darzustellen. Konkret bietet es sich in TOSCA an, wie in Abbildung 5.2 anhand eines RaspberryPi gezeigt, jeden Fähigkeittyp einer Hardwarekomponente als eine Capability darzustellen, in welcher weiterhin in einem Constraint die verschiedenen Hardwareports aufgezählt sind über welche diese Fähigkeit bezogen werden kann. Diese Hardwareports werden später in der Darstellung in XML lediglich als *ports* bezeichnet. Dabei sollten zu jedem der Hardwareports die zugehörigen Verbinder anhand ihres eindeutigen Namens aufgelistet werden. Dieser ist auch zum Referenzieren auf diesen Hardwareport nutzbar. Diese Verbinder können jede Art von Steckern, Steckbuchsen, Lötstellen, Kontaktbereichen, Sendern, Empfängern oder Kombinationen daraus sein. Soll eine Verifikation oder automatische Erkennung zueinander passender Stecker ermöglicht werden, so muss zusätzlich der Typ eines Verbinders sowie dessen Geschlecht angegeben werden. Dies kann durch einen weiteren Eintrag namens connectorType geschehen, das Geschlecht des Verbinders kann dabei in einem Attribut *gender* als *male*, *female*, oder in seltenen Fällen als *universal* angegeben werden. Der eindeutige Name eines Verbinders ist meist auf der entsprechenden Komponente neben dem Verbinder zu finden oder im Handbuch dieser Komponente aufgelistet. In manchen Fällen ist nur der Name eines Verbunds an Verbindern angegeben, in diesem Fall existiert im Normalfall eine Nummerierung der einzelnen Verbinder diese Verbunds. Hierbei setzt sich der eindeutige Name eines Verbinders aus dem Namen des Verbundes sowie der individuellen Nummer zusammen. Um in diesen Fällen Fehlkonfigurationen zu vermeiden, müssen jedoch zusätzlich die Funktionen der einzelnen Verbinder angegeben werden. Dies muss sowohl im Constraint eines Requirements als auch in dem einer Capability angegeben werden, um einen Abgleich dieser Funktionen zu ermöglichen. Entsprechend wird in dem genannten Beispiel bei den Hardwareports *Hardwareport 2* und *Hardwareport 3* die Funktion der einzelnen Anschlüsse angegeben um eine Verwechslung beim Anschluss von Verbindern miteinander zu verhindern.

Anforderungen einer Hardwarekomponente können wie in Abbildung 5.2 gezeigt ebenfalls auf diese Art als Requirements dargestellt werden, inklusive der Anschlüsse über welche diese Anforderung erfüllt werden kann. Die Nennung der Verbinder ermöglicht später bei Relationen zwischen Hardwarekomponenten eine automatische Erkennung möglicher Verbindungen. Auch ermöglicht dies die Verbindung von Hardwareports, welche über mehrere Verbinder aufgeteilt sind. Da im letzteren Fall möglicherweise sämtliche Verbinder eines Hardwareports vom selben Typ sind, muss hier zusätzlich ihre Funktion angegeben werden. Ohne diese Angabe müssten für jede Verbindung mit einem solchen Hardwareport die zu verbindenden Verbinder in einer Relation manuell angegeben werden. Durch diese Angabe können automatisch Verbinder mit derselben Funktion als zusammengehörig erkannt werden Im elektrotechnischen Umfeld sind die Bezeichnungen ”Stecker” und ”Steckbuchse”

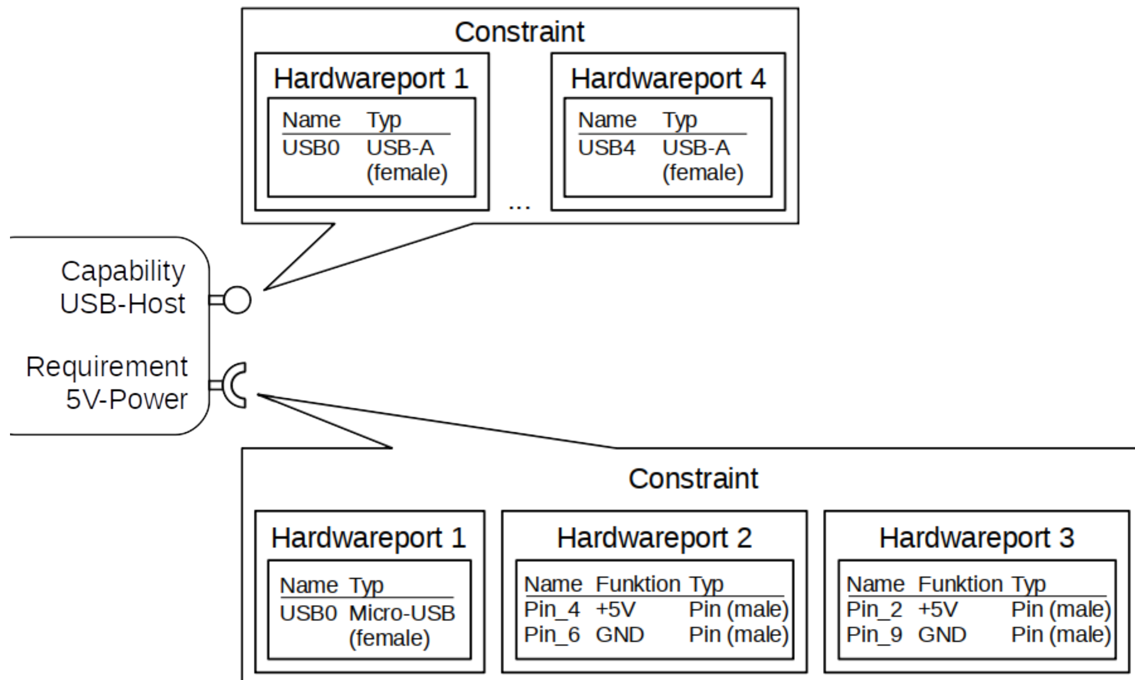


Abbildung 5.2: Grafische Darstellung einer von Hardwarekomponenten nutzbaren Capability USB-Host und eines Requirements einer Node am Beispiel eines RaspberryPi. In Constraints sind zugehörige Hardwareports mit ihren Anschlüssen angegeben.

beziehungsweise ”Male Connector” und ”Female Connector” für Kontaktverbindungen geläufig, dabei lassen sich jeweils ein Stecker mit einer Steckbuchse verbinden. Hierbei zeigen Stecker ihre Kontakte als herausstehende, oft berührbare Elemente, bei Steckbuchsen sind die Kontakte oft nicht von Hand berührbar. Häufig werden Steckbuchsen zum Anbieten von Fähigkeiten verwendet, um ungewollten Kontakt mit Objekten zu vermeiden wie es zum Beispiel bei Steckdosen des Stromnetzes ersichtlich ist. Daher kann bei der Darstellung einer Hardwarekomponente oft darauf zurückgegriffen werden, dass über Steckbuchsen häufig Capabilities angeboten werden während Stecker oft auf ein vorhandenes Requirement hinweisen. Geht der Hardwareport einer Hardwarefunktion kontaktlos Verbindungen ein, wie zum Beispiel über Funk oder Infrarotsignale, so besteht häufig keine Beschränkung auf nur eine verbundene Hardwarekomponente. In diesem Fall kann die Anzahl maximal möglicher Verbindungen im Constraint bei dem entsprechenden Hardwareport angegeben werden, beispielsweise als zusätzliche Eigenschaft *maxConnections*.

Montagemöglichkeiten stellen ebenfalls Hardwarefähigkeiten dar, welche wie auch Daten-, Signal- und Stromverbindungen über Hardwareports angeboten werden. Ihre Verbinder ermöglichen jedoch keine Daten- oder Stromverbindung, sondern lediglich eine mechanische Verbindung. Damit können auch Montagemöglichkeiten auf die genannte Art als Hardwarefähigkeiten dargestellt werden.

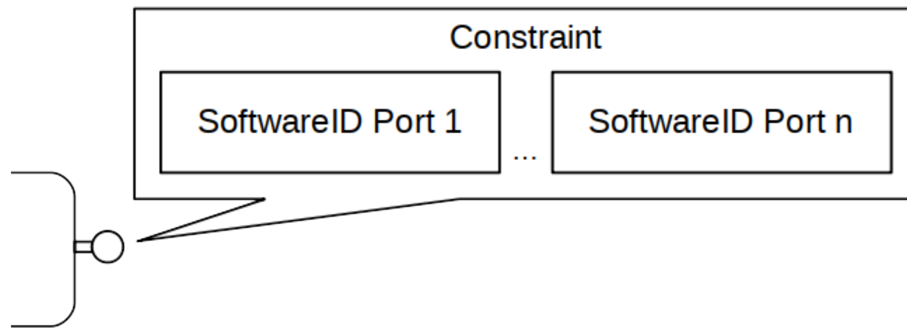


Abbildung 5.3: Grafische Darstellung einer mehrfach vorhandenen und von Softwarekomponenten nutzbaren Capability.

5.3.2 Computingkomponenten

Computingkomponenten sind eine spezielle Art von Hardwarekomponenten, da sie die Fähigkeit besitzen, dass auf ihnen Softwarekomponenten gespeichert und ausgeführt werden können. Damit können sie sowie ihre Hardware-Capabilities und -Requirements wie schon im vorherigen Abschnitt dargestellt werden. Die Fähigkeit, Softwarekomponenten auszuführen, lässt sich ebenfalls als Capability darstellen. Möglicherweise wird hierbei zum Ausschluss von Inkompatibilität mit Softwarekomponenten auch eine Angabe des Befehlssatzes beziehungsweise der Architektur in einem Constraint benötigt. Einige Computingkomponenten bieten verschiedene Befehlssätze für auf ihnen ausgeführte Softwarekomponenten an, so auch das in der Fallstudie verwendete RaspberryPi. Dieses bietet sowohl die Befehlssätze einer 32Bit ARM-Architektur als auch eine 64-Bit Variante davon an. Daher müssen diese Architekturen als einzelne Capabilities dargestellt werden. Eine Darstellung dieser Capabilities und ihrer Details in Properties der Computingkomponente wäre auch denkbar, dies ergibt jedoch kein einheitliches Konzept zur Modellierung und führt zu unnötiger Komplexität bei Implementierung des Konzeptes. Zusätzlich stellen Computingkomponenten weitere ihrer Fähigkeiten diesen Softwarekomponenten zur Verfügung, beispielsweise den Zugriff auf Hardwareports, Hardware-(De)Kodiereinheiten zur Ver- und Entschlüsselung oder (De)Komprimierung von Daten nach bestimmten Standards oder den Zugriff auf Sensoren. Hierzu gehören beispielsweise Hardwarebeschleuniger zum Dekomprimieren von Video- und Audiodaten oder Hardwareverschlüsselungsmodule, welche schnell und manipulationssicher Daten ver- und entschlüsseln können. Diese können in TOSCA wie sämtliche von anderen Komponenten nutzbare Fähigkeiten einer Komponente als Capabilities beschrieben werden. Auch können bei mehrfachem Vorhandensein einer solchen Fähigkeit diese einzeln in einem Constraint aufgelistet sein. Dabei muss jedoch auch eine Information über die Zugriffsmöglichkeit für eine darauf zugreifende Softwarekomponente hinterlegt sein, beispielsweise der Gerätepfad oder eine Schnittstellen-ID, welche einer Softwarekomponente zur Konfiguration einer Verbindung mit der Hardwarekomponente übergeben werden kann. Beispielhaft wird dies in Abbildung 5.3 dargestellt, im Constraint der Capability sind die aufgelisteten IDs zu sehen.

Falls es sich bei der Hardwarefähigkeit um den Zugriff für Softwarekomponenten auf einen Hardwareport handelt und man verifizieren können möchte, dass am zu dieser ID gehörigen physikalischen Anschluss die gewünschte Signalquelle oder -Senke durch eine Hardwareverbindung angeschlossen ist, so muss im Constraint zusätzlich ein Verweis auf die entsprechende Hardwarefähigkeit angegeben werden. In diesem Fall muss ebenfalls pro ID auch auf den entsprechenden

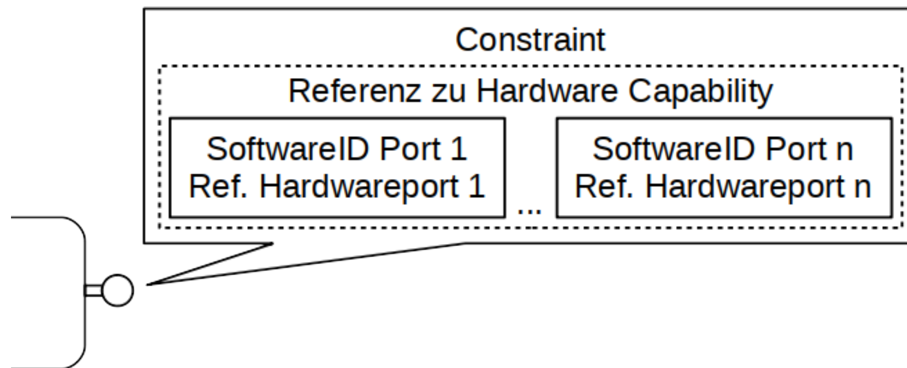


Abbildung 5.4: Grafische Darstellung einer von Softwarekomponenten nutzbaren Capability um auf Hardwareports eines Typs zuzugreifen. Das Constraint definiert Referenzen zur zugehörigen Capability für Hardwarekomponenten und den entsprechenden Ports sowie die von Softwarekomponenten nutzbaren IDs.

Hardwareport dieser zugehörigen Fähigkeit verwiesen werden. Dies lässt sich anhand der mehrfach vorhandenen Fähigkeit zu einzeln und im Verbund ansteuerbaren und auslesbaren Anschlüssen eines Mikrocontrollers der AVR-Familie demonstrieren, hier anhand des in den Modulen Arduino Pro Mini verbauten ATMega328, dessen mögliche Anschlussbelegungen in Abbildung 2.7 gezeigt wurden. Bei diesen Modulen können in Arduino geschriebene Softwarekomponenten mit der ID *1* das an Anschluss RXI anliegende Signal einzeln digital auslesen [ARD15a]. Soll an diesem Anschluss ein digitales Signal ausgegeben werden, so muss dazu ebenfalls die ID *1* genutzt werden. Wird dagegen von dieser Softwarekomponente auf die serielle Schnittstelle mit den Signalleitungen TXD und RXD über die ID *SERIAL* zugegriffen, so dürfen keine weiteren Funktionen über die Anschlüsse TXO und RXI genutzt werden, da diese zusammen den Hardwareport der UART-Schnittstelle bilden. Werden andere Bibliotheken mit möglicherweise anderen IDs für diese Hardwareports genutzt, so müssen hierfür adaptierende Bibliotheken oder Tabellen zur Konvertierung genutzt werden.

5.3.3 Interne Verbindungen zwischen Hardwareports

Einige Hardwarekomponenten leiten Signale, ohne diese oder ihre physikalische Darstellung zu verändern, von einem ihrer Hardwareports an einen oder mehrere ihrer Hardwareports durch. Dies ist im Fallbeispiel unter anderem beim RaspberryPi der Fall. Hierbei sind sämtliche Anschlüsse für die Stromversorgung der Computingkomponenten miteinander verbunden und leiten den an einem angeschlossenen Strom zu allen weiteren Anschlüssen weiter. Dies wird sich teilweise als relevant für die automatische Erkennung passender Steckverbindungen zeigen, insbesondere bei der Verwendung universeller Kabel. Dies kann erreicht werden indem in den Constraints einer Hardwarefähigkeit bei allen beteiligten Hardwareports aufgelistet wird, mit welchen weiteren Ports sie direkt verbunden sind. Ist keine solche Erkennung erwünscht, so ist diese Darstellung nicht nötig, allerdings müssen entsprechende Capabilities und Requirements von Hand angegeben werden. Abschnitt 5.3.4 greift darauf zurück um Kabel wie in Abbildung 5.5 gezeigt darzustellen.

5.3.4 Kabel

Da Kabel zwischen verschiedenen Hardwarekomponenten einzeln ausgebracht und daraufhin mit diesen Hardwarekomponenten verbunden werden müssen, müssen sie wie diese auch modelliert werden, insbesondere wenn ein hoher Detailgrad notwendig ist. Daher werden Kabel in diesem Konzept als Hardwarekomponenten angesehen, dies ermöglicht auch einen einheitlichen Ansatz dieses Modellierungskonzeptes. Nebenbei ermöglicht dies zusätzlich die Auflistung aller benötigten Komponenten eines modellierten Systems, beispielsweise um diese Komponenten für den Aufbau eines Systems zu kaufen oder um deren Verfügbarkeit zu prüfen. Jedes Kabel besitzt mindestens zwei Hardwareports, über deren Verbinder es an weitere Hardwarekomponenten angeschlossen werden kann. Für die Mehrheit der Verbinder an Hardwarekomponenten wird zur Vermeidung von Fehlkonfigurationen ein nur für das zu übertragende Signalformat standardisierter Verbindertyp verwendet. So werden beispielsweise USB-Anschlüsse (USB-C Standard ausgenommen) nur genutzt, um die Funktion einer 5V-Stromversorgung oder die eines USB-Hosts mit Stromversorgung anzubieten oder zu beziehen. Hierdurch können die Ports der meisten Kabel wie die Hardwareports anderer Hardwarekomponenten durch Capabilities und Requirements dargestellt werden. Möglicherweise muss dies entsprechend der Capabilities und Requirements der zu verbindenden Hardwarekomponenten erfolgen. So müssen die Ports eines Typs Kabel als Requirements dargestellt werden, wenn alle zu verbindenden Hardwarekomponenten diese Schnittstelle als Capability anbieten. Wird jedoch diese Schnittstelle von einer Komponente vorausgesetzt und daher als Requirement dargestellt, so muss ein Port dieses Kabels für diese Komponente als Capability dargestellt werden. Können Kabel jedoch für verschiedene Zwecke genutzt werden, so muss für jeden dieser Zwecke eine entsprechende Capability und ein Requirement dem Node Type hinzugefügt werden. Zusätzlich müssen Modellierer darauf achten, dass sie beim Erstellen von Node Templates die passenden Capabilities und Requirements hinzufügen. So können USB-Datenkabel dargestellt werden, indem sie an einem Anschluss ein Requirement für einen USB-Host haben und diese Fähigkeit am anderen Anschluss als Capability zur Verfügung stellen. Möchte man diese Kabel allerdings als reine Kabel zur Stromversorgung nutzen und beispielsweise mit einem USB-Netzteil verbinden, so muss man sie mit entsprechender Capability und dem Requirement für eine USB-Stromversorgung darstellen, da dieses Netzteil keine USB-Host Capability anbietet. Andererseits erlaubt diese Darstellung auch die Unterscheidung zwischen USB-Stromkabeln ohne Datenleitungen und USB-Datenkabeln obwohl beide dieselben Verbindertypen haben.

Viele Kabel besitzen lediglich zwei Hardwareports, welche sie miteinander verbinden, zusätzlich sind beide Hardwareports mit verschiedenen Verbindern ausgestattet. Andere Kabel verbinden jedoch mehrere Hardwareports untereinander, beispielsweise Kabel für sogenannte Bussysteme in Fahrzeugen. Zusätzlich verbinden spezielle Kabel nicht alle ihrer Anschlüsse untereinander, sondern vereinen verschiedene andere Kabel in sich. Ein Beispiel hierfür sind kombinierte "Keyboard-Video-Mouse" (KVM) Kabel, welche jeweils ein Kabel zum Anschluss einer Tastatur, einer Maus und eines Bildschirms an einem Computer in sich vereinen. Auch diese Kabel sind mit der genannten Vorgehensweise darstellbar, jedoch muss dabei auf systemspezifische Fallstricke geachtet werden. So darf bei Bussystemen wie DMX512 nur ein Knoten mit entsprechender Busmaster-Fähigkeit angeschlossen sein [ESTA18].

Manche Kabel sind universell einsetzbar, beispielsweise wenn sie Anschlüsse für Löt- oder Schraubkontakte besitzen. Da das Format der über ein Kabel übertragenen Daten meist irrelevant für die Funktion des Kabels ist, sind diese Kabel kaum durch in TOSCA verbreitete Methoden für ihre universelle Einsetzbarkeit darzustellen. Prinzipiell müssten hierfür Unmengen an möglicher

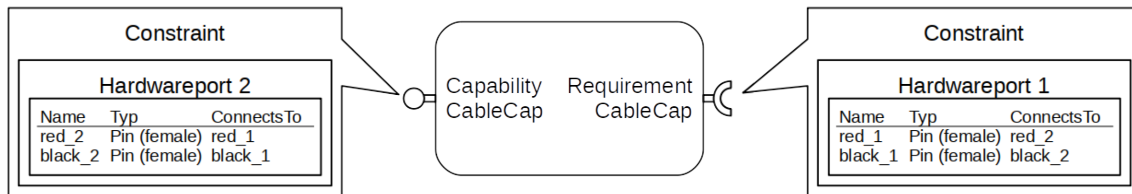


Abbildung 5.5: Grafische Darstellung eines Kabels als Hardwarekomponente mit einer generischen Capability und einem generischen Requirement. Interne Verbindungen zwischen Anschlüssen sind angegeben.

Capabilities und Requirements in den Node Types dieser Kabel definiert werden. Wird durch eine Ausnahme gebrochen, dass in TOSCA ausschließlich Capabilities und Requirements von zusammengehörigen CapabilityType und RequirementType durch Relationship Templates verbunden werden können, so ermöglicht dies eine Alternative. Hierdurch können Kabel auch als Node Templates mit generischen Weiterleitungs-Capabilities und -Requirements dargestellt werden. Sind mehrere Kabel zu einem vereint, wie die bereits angesprochenen KVM-Kabel, so bietet sich hierbei die schon genannte Darstellung von Verbindungen innerhalb einer Komponente an. Eine solche Darstellung eines Kabels als Node Template ist in Abbildung 5.5 anhand eines zweiadrigen Kabels mit roter und schwarzer Färbung gezeigt, welches entsprechend seine roten sowie schwarzen Verbinder (*red_1* und *red_2* sowie *black_1* und *black_2*) jeweils intern miteinander verbindet. Dass dies erforderlich ist kann insbesondere bei dem in Kapitel 4 genannten als Fallstudie entwickelten autonomen Fahrzeug gesehen werden. Bei diesem sind ein Großteil der Verbinder Stiftkontakte beziehungsweise Kontaktbuchsen, über welche verschiedenste Hardwarefähigkeiten der Computingkomponenten angeboten beziehungsweise bezogen werden. Diese Darstellung von Kabeln mit Weiterleitungs-Capability ist nur anwendbar, wenn berücksichtigt wird, dass dabei eine Capability, welche mit dem Weiterleitungs-Requirement verbunden ist, durch die Weiterleitungs-Capability des Kabels ein Requirement bedienen kann. Die Funktionen der Verbinder von anderen Capabilities und Requirements werden dabei über die internen Verbindungen der Kabel weitergeleitet. Dies ist auch bei kaskadierend miteinander verbundenen Kabeln zu berücksichtigen. Ebenfalls ist hierbei darauf zu achten, dass die entsprechenden Verbinder der Hardwareports kompatibel sind.

Ist ein physikalisches Kabel fester Bestandteil einer anderen Hardwarekomponente und eine Trennung von dieser technisch nicht vorgesehen, so ist es unnötig, dieses Kabel als einzelne Komponente darzustellen, da es Teil dieser Hardwarekomponente ist und lediglich einen ihrer Anschlüsse darstellt. Dies ist beispielsweise bei Anschlusskabeln von Webcams oder Kabeln von Steckernetzteilen häufig der Fall.

5.3.5 Splitter und Merger

Spezielle Kabel oder Adapterstecker teilen Signale auf oder führen sie zusammen, beispielsweise Adapter von einem kombinierten Anschluss für Kopfhörer/Mikrofon auf einen Anschluss für ein Mikrofon und einen Anschluss für Kopfhörer mit Stereo-Audio. Solche Kabel können als Komponente mit entsprechenden Capabilities und Requirements dargestellt werden, wobei die Capabilities und Requirements denen der anzuschließenden Komponenten entsprechen müssen. Kann ein Adaptertyp für verschiedene Capabilities genutzt werden, so muss für jede Kombination an Capabilities und Requirements ein weiterer Knotentyp erstellt werden. Eine solche Darstellung

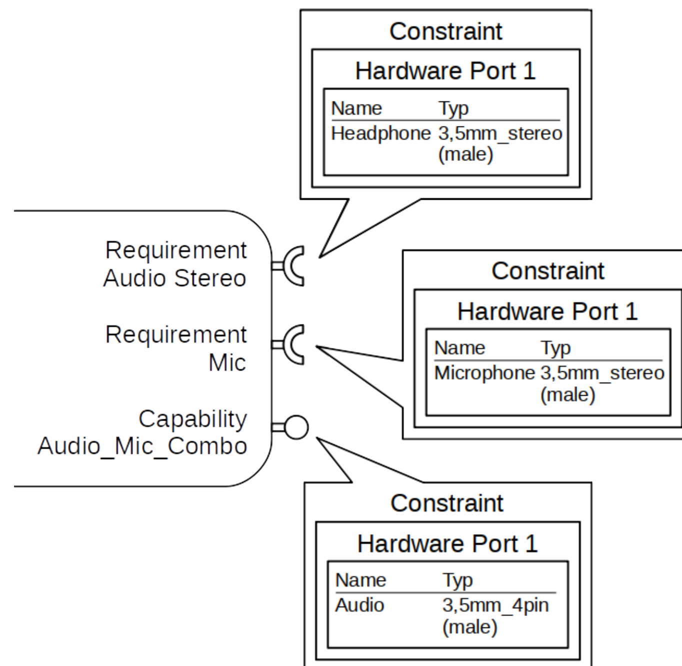


Abbildung 5.6: Grafische Darstellung des in Abbildung 2.5 gezeigten Kabels zum Aufteilen beziehungsweise Zusammenführen von Audiosignalen.

wird in Abbildung 5.6 anhand des in Abbildung 2.5 gezeigten Audio-Adapterkabels gezeigt. Dieses Kabel benötigt die Hardwarefunktionen Stereo-Audio sowie Mikrofon jeweils separat über die definierten Hardwareports und bietet einen Hardwareport für ein kombiniertes Signal von Audio und Mikrofon an.

5.3.6 Aktive Adapter und Erweiterungsmodule

Aktive Adapter sind Hardwarekomponenten, welche Signale zwischen den Datenformaten verschiedener Hardwareports umwandeln. Dies kann je nach Adapter unidirektional oder bidirektional geschehen. Dabei können Adapter diese Funktion unabhängig von weiteren Komponenten bieten und damit einfach als eine Komponente mit entsprechenden Capabilities und Requirements dargestellt werden. Bei anderen Adaptern muss sich eine Softwarekomponente mit dem Adapter verbinden, um eine von ihm angebotene Hardwarefunktion zu nutzen. Diese Adapter müssen dafür erst mit Computingkomponenten verbunden werden. Weiterhin existieren Erweiterungsmodule, welche ähnlich wie ein Adapter Softwarekomponenten bestimmte Hardwarefähigkeiten zur Verfügung stellen. Allerdings sind dies interne Hardwarefähigkeiten wie beispielsweise ein Temperatur- oder Kamerasensor. Bei der Modellierung solcher Adapter und Erweiterungsmodule kann auf dieselbe Art vorgegangen werden, wie schon bei der Modellierung von solchen Computingkomponenten, die Hardwarefunktionen für Softwarekomponenten anbieten.

In manchen Fällen ist der Zugriff auf Adapter nur einer für ihre Verwaltung zuständigen Softwarekomponente möglich, einem sogenannten Treiber. Solche Treiber werden in Abschnitt 5.4 genauer behandelt, sie bieten jedoch weiteren Softwarekomponenten Zugriff auf die Hardwarefunktionen der entsprechenden Hardwarekomponente. Unter Umständen kann die Darstellung von Treibern im

TOSCA-Modell für manche Hardwarekomponenten entfallen, wenn die Treiber beispielsweise Teil eines Betriebssystems sind und die IDs der von den Treibern angebotenen Ports bekannt sind oder zur Laufzeit bestimmt werden können. Dies wird ebenfalls in Abschnitt 5.4 behandelt.

5.4 Modellierung von CPS-Softwarekomponenten in TOSCA

TOSCA wird bisher für die Modellierung von Softwarekomponenten und virtuellen Maschinen in Cloud-Umgebungen eingesetzt. Da die Hardware dieser Cloud-Umgebungen von Cloud-Anbietern verwaltet wird und lediglich virtuelle Maschinen in dieser Umgebung bezogen werden können ist eine Darstellung von Hardware nicht nötig und mangels Wissen über die vorhandene Hardware auch nicht möglich. Daher muss und kann Hardware bei der Modellierung von Cloud-Anwendungen in TOSCA nicht modelliert werden. Soll nun auch die Modellierung von Hardwarekomponenten, darauf gehosteter Softwarekomponenten sowie deren Zugriff auf Fähigkeiten der Hardwarekomponenten ermöglicht werden, so bedarf es dafür Erweiterungen des TOSCA-Standards.

5.4.1 Betriebssysteme

Da Cloud-Anbieter normalerweise anbieten, beim Beziehen einer virtuellen Maschine direkt ein Betriebssystem darauf zu installieren, kann in diesen Fällen beides zusammen als eine Komponente dargestellt werden. Werden nun Computingkomponenten in TOSCA dargestellt, so müssen darauf gehostete Betriebssysteme als eigenständige Komponenten dargestellt werden, sobald sie nicht vom Hersteller vorinstalliert mitgeliefert werden. Eine Computingkomponente mit einem darauf gehosteten Betriebssystem oder einer anderen Softwarekomponente kann unter Umständen auch als eine Komponente dargestellt werden. In diesem Fall ist jedoch möglicherweise bei Implementierungen dieses Standards zu beachten, dass dieser Komponententyp ein Hybrid aus Software und Hardware ist und Interaktionsmöglichkeiten sowohl auf physische Weise als auch durch Software anbietet. Müssen mögliche Inkompatibilitäten zwischen Hardware und Betriebssystem wie zum Beispiel der benötigte Befehlssatz oder eine benötigte Grafikeinheit beachtet werden, so müssen auch diese dargestellt werden. Auch hierbei bietet es sich an, diese Voraussetzungen des Betriebssystems als Requirements darzustellen, auch um eine einheitliche Vorgehensweise bei der Modellierung von Komponenten zu ermöglichen. Im Falle des in dem autonomen Fahrzeug der Fallstudie verwendeten Betriebssystems Raspbian ist dies ein Requirement für *Architecture_ARM32*.

5.4.2 Treiber

Treiber für Hardwarekomponenten sind Softwarekomponenten, welche auf diese Hardwarekomponenten zugreifen und deren Hardwarefähigkeiten verwalten und weiteren Softwarekomponenten zur Verfügung stellen. Da die dabei für Softwarekomponenten angebotene Funktionalität lediglich die vorhandenen Hardwarefunktion anbieten kann ist es beispielsweise bei dem Zugriff auf Hardwareports von Funktionen notwendig, den entsprechenden Hardwareport anzugeben. Hierfür müssen eine Art virtuelle Hardwareports angeboten werden. Dies lässt sich darstellen wie die Fähigkeit einer Hardwarekomponente welche Softwarekomponenten den Zugriff auf eine Hardwarefähigkeit anbietet. Eine solche Darstellung ist in Abbildung 5.7 zu sehen, hier ist auch das entsprechende Requirement auf die zugehörige von einer Hardwarekomponente angebotene Fähigkeit zu sehen.

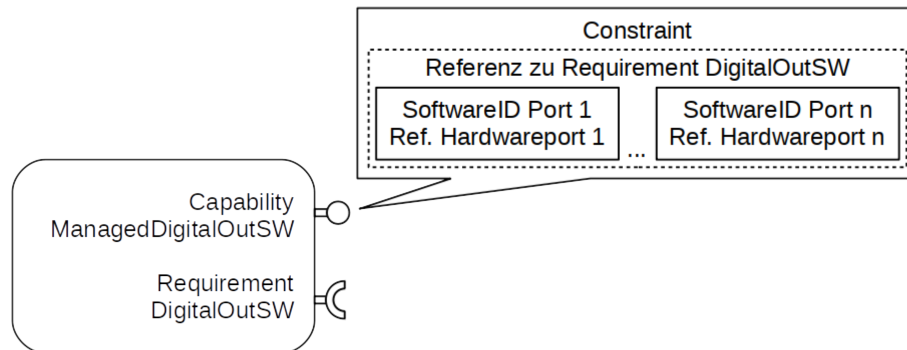


Abbildung 5.7: Grafische Darstellung eines Treibers als Softwarekomponente mit einer Capability mit virtuellen Ports sowie einem Requirement, welches durch eine Fähigkeit einer Hardwarekomponente erfüllt werden kann.

Oftmals suchen Treiber sich selbstständig sämtliche kompatiblen Geräte, die an der Hardwarekomponente angeschlossen sind, welche das dem Treiber unterliegende Betriebssystem hostet. In diesen Fällen dient eine Relation zur entsprechenden Hardwarekomponente nur dem Verständnis sowie möglichen Algorithmen zum Erkennen von Datenverbindungen und muss nicht konfiguriert werden. Ist ein Treiber ein standardmäßiger Bestandteil eines Betriebssystems und muss nicht einzeln installiert werden, so muss dieser nicht als einzelne Komponente dargestellt werden. In diesem Fall müssen von diesem Treiber angebotene Capabilities für Softwarekomponenten so dargestellt werden, dass sie entweder von dem Betriebssystem oder wie schon genannt dem Adapter selbst angeboten werden. In vielen Fällen zeigen Treiber betriebssystemspezifisch unterschiedliches Verhalten und müssen daher für jeden Betriebssystemtyp einzeln modelliert werden. Beim Modellieren eines Node Types für einen Treiber ist oftmals nicht bekannt, wie viele Hardwareports die zu verwaltende Hardwarefunktion hat. Daher kann in einem solchen Fall die später angebotene Anzahl an Funktionen sowie deren Details wie IDs nicht angegeben werden. Dies ist ein nicht vollständig geklärter Aspekt dieser Arbeit und muss voraussichtlich in jedem Einzelfall geklärt werden.

5.5 Modellierung von Relationen mit Hardwarekomponenten in TOSCA

Soll in TOSCA die Möglichkeit bestehen, Hardwarekomponenten darzustellen, so muss es auch ermöglicht werden, ihre Beziehungen untereinander sowie zwischen ihnen und Softwarekomponenten darzustellen. Untereinander können Hardwarekomponenten eine Verbindung für Daten-, Signal- oder Stromtransfer eingehen, allerdings kann auch eine Hardwarekomponente an einer anderen montiert sein. Ebenfalls kann eine Hardwarekomponente von einer weiteren abhängig sein und entsprechend ohne deren Präsenz ihre Funktionalität nicht bieten oder gewährleisten. Zusätzlich existieren Relationen zwischen Softwarekomponenten und Hardwarekomponenten, hierbei handelt es sich meist um ähnliche Beziehungen wie zwischen Softwarekomponenten untereinander. Softwarekomponenten können auf Hardwarekomponenten gespeichert und ausgeführt werden oder sich mit ihnen verbinden um eine ihrer Funktionen zu nutzen. Auch können Softwarekomponenten von der Präsenz bestimmter Hardwarekomponenten abhängig sein, um ihre Funktion ausführen zu können. Solche direkten Abhängigkeiten zwischen Hardwarekomponenten oder auch zwischen Software- und Hardwarekomponenten können wie auch die Abhängigkeiten

zwischen Softwarekomponenten als Relation dargestellt werden. Diese Art an Relation muss im Gegensatz zu anderen Relationen zwischen Komponenten nicht eingerichtet werden, sondern dient in Modellen lediglich zum Verständnis sowie beim Deployment eines Systems zur Steuerung der Reihenfolge beim Instantiieren von Komponenten. Damit können solche Abhängigkeiten zwischen Hardwarekomponenten wie Abhängigkeiten zwischen Softwarekomponenten durch ein Relationship von einem abstrakten Relationship Type *DependsOn* dargestellt werden.

Da andere Arten an Relationen die Konfiguration der beteiligten Komponenten zueinander darstellen, können diese ebenfalls als Relationen dargestellt werden. Konkret bedeutet dies beispielsweise, dass eine Signal-, Daten- oder Stromverbindung zwischen zwei Hardwarekomponenten aufgebaut wird, indem ein entsprechender Kontakt oder die Möglichkeit einer kabellosen Verbindung zwischen diesen Komponenten hergestellt wird. Ebenfalls wird eine Montageverbindung einer Hardwarekomponente zu einer weiteren durch das Montieren der einen Komponente an oder auf der anderen aufgebaut.

5.5.1 Relationen zwischen Hardwarekomponenten

Stellt in TOSCA eine Relation eine Kommunikationsverbindung von einer Softwarekomponente zu einer weiteren dar, so wird beim Deployment der Softwarekomponenten ausgehend von dem Node Template am Ziel der Relation nach Informationen für die Konfiguration des Node Templates an der Quelle der Relation gesucht. Dies sind beispielsweise der Port, auf welchem eine Instanz des Node Templates am Ziel der Relation auf eingehende Verbindungen reagiert und die IP des unterliegenden Betriebssystems. Das Erstellen von elektrischen Verbindungen oder Montageverbindungen zwischen Hardwarekomponenten besteht dagegen oft aus dem Herstellen einer Steck-, Klemm- oder Schraubverbindung und damit dem Einstecken oder Einschrauben des Steckers oder Montageelements einer Hardwarekomponente in die Steckbuchse oder Montageposition einer anderen. Solche Verbindungen haben im Vergleich zu denen zwischen Softwarekomponenten meist die Einschränkung, dass zwar oftmals mehrere Hardwareports mit derselben Funktionalität pro Hardwarekomponente verfügbar sind, allerdings für jeden dieser Hardwareports nur eine von exakt einer weiteren Komponente nutzbare Verbindung vorhanden ist.

Elektrische Verbindungen

Um die Konfiguration der Schnittstellen zum Erstellen einer Verbindung von einer Hardwarekomponente zu einer weiteren darzustellen, kann eine Erweiterung des Konzepts zwischen Softwarekomponenten angewendet werden. Wird wie in Abschnitt 5.3 vorgestellt jede Hardwarefähigkeit als Capability oder Requirement angegeben, so kann anhand deren Constraints bei beiden beteiligten Hardwarekomponenten ein geeigneter Hardwareport zum Erstellen einer Verbindung ausgewählt werden. Dies kann manuell erfolgen, indem ein Relationship Template von dem Requirement ausgehend zu der Capability führt. Sollen dabei bestimmte Hardwareports miteinander verbunden werden, so müssen in dem RelationshipConstraint dieses Relationships die IDs der entsprechenden Hardwareports für die Quell- und Zielnode angegeben werden. Werden nun Hardwareports mit mehreren Verbindern miteinander verbunden, so kann die Funktion der einzelnen Verbinder miteinander abgeglichen werden um eine Ist bei den Verbindern keine Funktion angegeben, so muss in dem RelationshipConstraint des Relationships statt dem Hardwareport jedes Paar an Verbindern

angegeben werden, welche miteinander verbunden werden sollen. Da dies allerdings den Aufwand lediglich beim Definieren von Node Types verringert, beim Modellieren eines Systems jedoch zu deutlichem Mehraufwand führt, wird von einer solchen Darstellung sofern möglich abgeraten.

In Abbildung 5.8 ist diese Darstellung am Beispiel eines Arduino-Moduls, eines USB zu Seriell Adapters und eines sie verbindenden Kabels mit zwei Verbindern pro Port gezeigt. Dabei verbindet ein Relationship Template das Requirement des Node Templates des Arduino-Moduls mit der generischen Capability des Node Templates des Kabels. Ein weiteres Relationship Template verbindet das zugehörige generische Requirement des Node Templates des Kabels mit der Capability des Node Templates des Adapters. Hierdurch wir das Requirement des Node Template des Arduino-Moduls über das Node Template des Kabels durch die Capability des Node Templates des Adapters erfüllt. Durch die angegebenen Funktionen der Verbinder des Arduino-Moduls und des Adapters sowie der internen Verbindungen zwischen den Hardwareports des Kabels kann eine Anschlussmöglichkeit des Kabels an diese beiden Komponenten ermittelt werden, welche deren Verbinder entsprechend ihrer Funktion miteinander verbindet. Da bei der angegebenen seriellen Schnittstelle (UART) die Verbinder zum Senden von Daten jeweils mit einem Verbinder zum Empfangen verbunden werden müssen, muss die Funktion bei den Anschlüssen des Requirement vertauscht angegeben. Entsprechend wird ein angeschlossenes Kabel als sogenanntes ”Nullmodem-Kabel” verwendet.

Zusätzlich sind wie schon genannt bei vielen Mikrocontrollern und System-on-a-Chips die Hardwareports mehrerer Kommunikations- und Signalschnittstellen auf denselben Anschlüssen verteilt und schließen sich bei Benutzung teilweise gegenseitig aus. Dies kann entweder bei der Erstellung eines Systems bedacht werden oder durch einheitliche Benennung der einzelnen Verbinder wird eine automatische Überprüfung ermöglicht. Handelt es sich bei der Daten- oder Signalverbindung um eine kabellose Verbindung, so ist das Erstellen davon lediglich die Positionierung und unter Umständen die Ausrichtung der Hardwarekomponenten zueinander, so beispielsweise bei Verbindungen über Funk oder Infrarotlicht. Hierbei müssen in einem RelationshipConstraint die Parameter für die Positionierung oder Ausrichtung angegeben werden, beispielsweise der maximale Abstand.

Montageverbindungen

Stellt eine Relation die Montageverbindung zwischen zwei Komponenten dar, so kann ihre Darstellung ähnlich der im vorhergehenden Abschnitt genannten Hardwareports geschehen. Bei einer solchen Verbindung stellen Verbinder keine elektrischen, sondern rein mechanische Verbinder dar. Hierbei kann es sich jedoch auch rein um die Platzierung einer Hardwarekomponente auf einer weiteren handeln, ohne dass die Komponenten eine mechanische Verbindung eingehen. Ein Beispiel hierfür wäre das Platzieren einer Hardwarekomponente in einem Fach oder auf einem Tisch. Das Darstellen einer kontaktlosen Montage durch Platzierung einer Hardwarekomponente frei im Raum, analog zur Verbindung zweier Hardwarekomponenten durch Funksignale, ist hierbei prinzipiell nicht notwendig. Dies könnte jedoch theoretisch in Extremfällen in gesteuerten Magnetfeldern oder in der Raumfahrt vorkommen, möglicherweise auch wenn Drohnen als einzelne Komponenten gesehen werden. Für solche rein mechanischen Verbindungen bietet sich für diverse Darstellungsformen und Algorithmen ein zusätzlicher Relationship Type an.

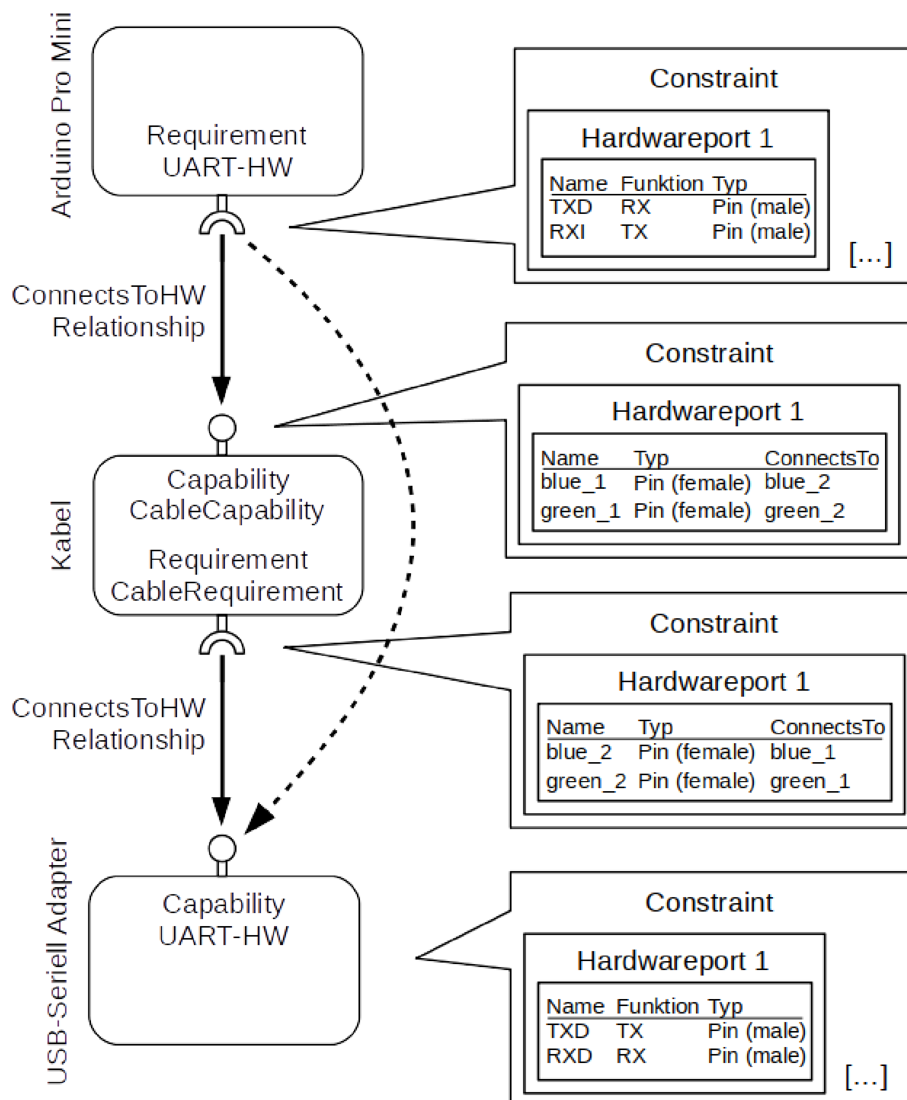


Abbildung 5.8: Grafische Darstellung eines Arduino Pro Mini und eines USB zu Seriell Adapters sowie eines sie verbindenden Kabels. Zwei Relationen definieren, welche Capabilities und Requirements der beteiligten Komponenten miteinander verbunden werden sollen.

5.5.2 Relationen zwischen Softwarekomponenten und Hardwarekomponenten

Soll eine Softwarekomponente auf eine Hardwarefähigkeit einer Hardwarekomponente zugreifen, so erfordert dies eine Konfiguration der Softwarekomponente und muss damit als Relation zwischen diesen beiden Komponenten dargestellt werden. Stellt die Hardwarekomponente dabei lediglich eine einzige durch eine Capability dargestellte Hardwarefähigkeit zur Verfügung, so können die darin angegebenen Daten genutzt werden, um eine Verbindung einer Softwarekomponente zu dieser Hardwarefähigkeit einzurichten. Stellt die Hardwarekomponente jedoch eine Fähigkeit mehrfach zur Verfügung und soll eine Softwarekomponente auf eine spezielle dieser Fähigkeiten zugreifen,

so ist ohne weitere Angaben nicht feststellbar, welche dieser Fähigkeiten die Softwarekomponente nutzen soll. In diesem Fall muss das verbindende RelationshipTemplate ein RelationshipConstraint mit der Angabe der Capability und des Requirements besitzen.

Hosting

Soll eine Softwarekomponente auf einer Hardwarekomponente gespeichert und ausgeführt werden, so muss sie dafür auf diese Hardwarekomponente übertragen werden. In TOSCA wird dies zwischen Softwarekomponenten sowie Softwarekomponenten und Komponenten, welche Virtuelle Maschinen darstellen, durch Relationen vom Typ *HostedOn* dargestellt. Diese Relation lässt sich auch nutzen für die Beziehung zwischen einer Softwarekomponente und einer Hardwarekomponente, auf welcher sie gespeichert ist und ausgeführt wird. Bei detaillierterer Darstellung von cyber-physischen Systemen muss diese Relation unter Umständen zu einer Relationen zu der speichernden Komponente sowie einer Relation zu der ausführenden Komponente aufgeteilt werden. In diesem Fall sollten die Relationen zum Beispiel *StoredOn* und *ComputesOn* genannt werden. Anhand eines auf einer Computingkomponente mit integriertem Speicher installierten Betriebssystem sowie einem weiteren auf einem RaspberryPi, welches in seine Komponenten Platine und Speicherkarte aufgeteilt ist, werden diese beiden Möglichkeiten in Abbildung 5.9 aufgezeigt. Entsprechend stellt eine Relation vom Typ StoredOn dar, dass die Softwarekomponente an der Quelle der Relation auf der Hardwarekomponente am Ziel der Relation gespeichert wird. Ebenso zeigt eine Relation vom Typ *ComputesOn*, dass die Softwarekomponente an ihrer Quelle von der Hardwarekomponente an ihrem Ziel ausgeführt wird und möglicherweise ein Requirement der Softwarekomponente nach einer bestimmten Hardwarearchitektur erfüllt. Eine Relation vom Typ *HostedOn* von einer Softwarekomponente auf eine Hardwarekomponente hat unter anderem ebenfalls eine solche Bedeutung. Da im Normalfall eine Softwarekomponente nicht für das Ausführen auf einer Computingkomponente konfiguriert werden muss, dient eine Relation vom Typ *ComputesOn* dabei lediglich dem Verständnis und möglicherweise der Validierung eines Modells. Relationen vom Typ *StoredOn* zeigen jeweils auf, dass auf der Hardwarekomponente am Ziel der Relation nach Informationen und Management Operationen für das Installieren der Softwarekomponente an der Quelle der Relation gesucht werden muss. Die Fähigkeit, Softwarekomponenten zu speichern, kann als Capability dargestellt werden. Wird dies gemacht, so müssen Softwarekomponenten ein entsprechendes Requirement auf diese Capability sowie eine Relation vom Typ *StoredOn* zu einer entsprechenden Hardwarekomponente haben. Dies gilt ebenfalls für Relationen vom Typ *HostedOn*. In der Praxis dürfte dies jedoch nur für automatische Verifikation von Modellen sowie für automatische Vervollständigung dieser relevant sein. Ebenfalls zeigen diese Relationen auf, dass beim Deployment des Gesamtsystems erst die Hardwarekomponente am Ziel der Relation bereitgestellt werden muss bevor die Softwarekomponente an der Quelle der Relation darauf bereitgestellt werden kann.

Zugriff auf Hardwarefähigkeiten

In cyber-physischen Systemen ist es naturgemäß häufig der Fall, dass Softwarekomponenten auf Funktionen von Hardwarekomponenten zugreifen. Soll eine Softwarekomponente auf eine Hardwarefähigkeit zugreifen, so muss dies durch ein Relationship Template dargestellt werden, welches von ihr zu der Hardwarekomponente führt, welche diese Fähigkeit anbietet. Da Hardwarefähigkeiten

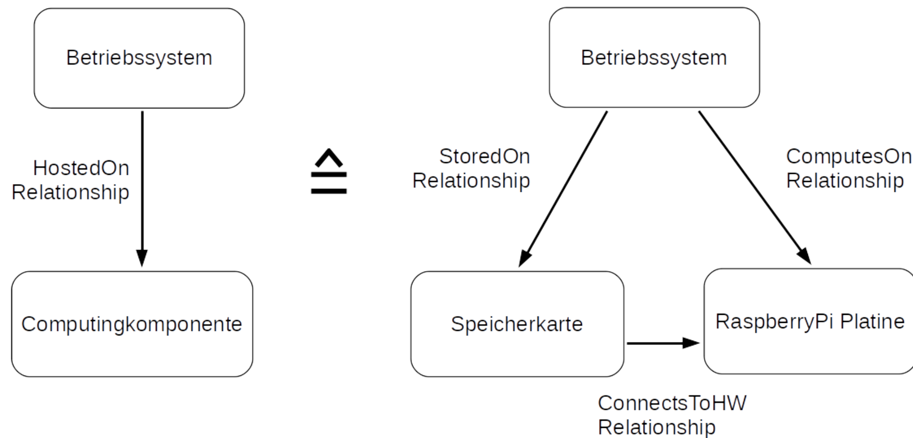


Abbildung 5.9: Mögliche Relationen zwischen Hardwarekomponenten und darauf gespeicherten und ausgeführten Softwarekomponenten am Beispiel eines Betriebssystems.

nach ihrem Typ zusammengefasst durch Capabilities dargestellt werden, sind in der Relation die Informationen nötig, auf was für eine Capability die Softwarekomponente zugreifen soll. Da ein Node Template allerdings mehrere Requirements des selben Requirement Types besitzen kann, muss ebenfalls dargestellt werden, welches Requirement durch dieses Relationship Template erfüllt werden soll. Dies wird durch die Angabe des Requirements in einem RelationshipConstraint des Relationship Templates realisiert. Da über den Requirement Type der zugehörige Capability Type und darüber wiederum die passende Capability des Node Templates am Ziel des Relationship Templates ermittelt werden kann ist eine Angabe der zu nutzenden Capability unnötig.

Soll dabei allerdings aus bestimmten Gründen eine bestimmte Hardwarefunktion einer Capability der Hardwarekomponente genutzt werden, so muss auch deren ID in dem RelationshipConstraint angegeben werden, hier als Element *TargetPort*. Alternativ muss je nach Implementierung zufällig oder entsprechend der verbundenen Kabel sowie der Relationen von und zu der beteiligten Softwarekomponente automatisch ein passender Hardwareport ausgewählt werden. Soll eine Softwarekomponente auf mehrere bestimmte Hardwareports zugreifen, so können entweder mehrere Relationship Templates genutzt werden, sofern diese Verbindungen einzeln bei der entsprechenden Softwarekomponente eingerichtet werden können. Ist dies jedoch nur in einem Schritt möglich, so können schlichtweg mehrere *TargetPort* Einträge im RelationshipConstraint angegeben werden. Die Bedeutung ihrer Reihenfolge ist in diesem Fall abhängig von der Implementierung der Softwarekomponente. Für solche Relationen bietet sich der Relationship Type *ConnectsTo* oder ein neuer Relationship Type an. Eine solche Relation wird in Abbildung 5.10 anhand der seriellen Schnittstelle eines im Fallbeispiel genutzten Arduino Pro Mini verdeutlicht. Diese serielle Schnittstelle wird dabei als Capability *UART-SW* dargestellt. Nun soll eine Relation vom Typ ConnectsTo die Verbindung einer Arduino-Firmware mit dieser Funktion der seriellen Schnittstelle verbinden. Dabei soll sie sich auf die erste serielle Schnittstelle verbinden, sprich dem ersten Hardwareport, da dieser wie in Abbildung 5.8 dargestellt über ein Kabel an eine weitere Hardwarekomponente angeschlossen ist.

Greift eine Softwarekomponente auf eine standardmäßig vorhandene interne Hardwarefunktion einer Komponente zu und ist es nicht nötig, dies zu konfigurieren, so kann von der Darstellung dieses Zugriffs durch eine Relation abgesehen werden. Dies verhindert eine unnötig höhere Komplexität des modellierten Systems und bedeutet weniger Aufwand für Modellierer.

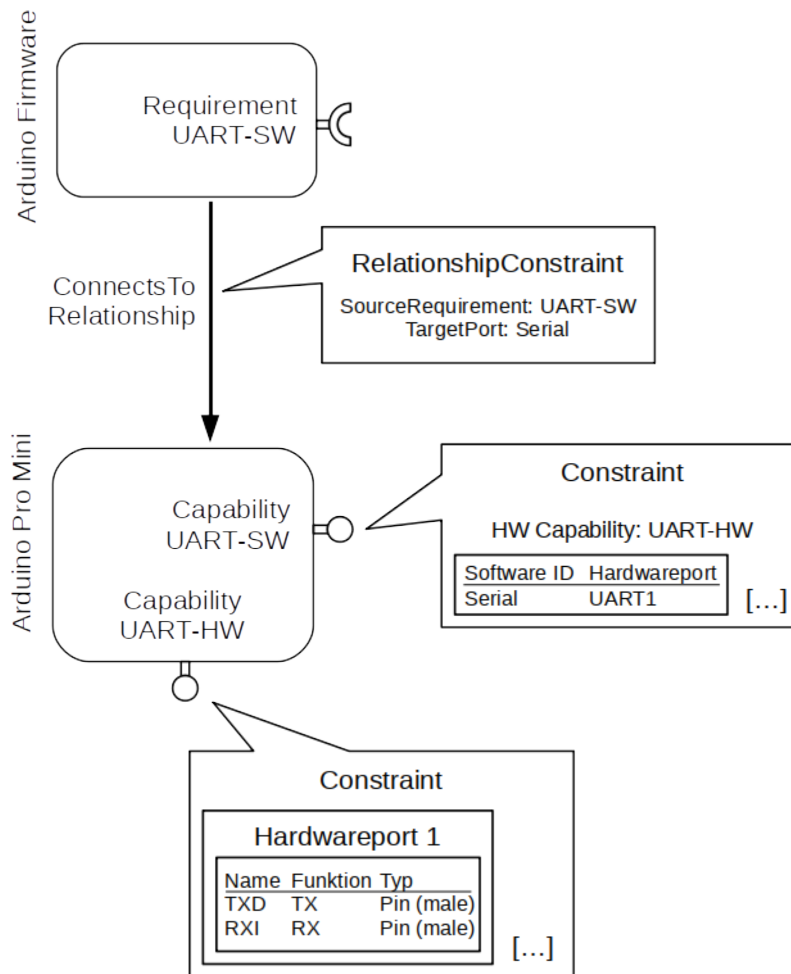


Abbildung 5.10: Zugriff einer Softwarekomponente auf einen Hardwareport einer Hardwarekomponente, dargestellt durch eine Relation.

Relationen von Hardwarekomponenten zu Softwarekomponenten

Reine Hardwarekomponenten können nicht dazu konfiguriert werden, sich mit einer Softwarekomponente zu verbinden, sie senden maximal Daten an eine weitere Hardwarekomponente, welche wiederum Softwarekomponenten ermöglicht, sich auf den entsprechenden Hardwareport zu verbinden und sich dadurch mit der sendenden Hardwarekomponente zu verbinden. Es existieren lediglich wenige Hardwarekomponenten, welche erst nach der Installation eines zugehörigen Treibers angeschlossen werden dürfen. Aus diesem Grund ist maximal die Abhängigkeit durch ein Relationship Template von einer Hardwarekomponente zu einer Softwarekomponente darstellbar. Weitere mögliche Verbindungen oder Relationen von Hardwarekomponenten zu Softwarekomponenten wurden im Rahmen der Recherche zu dieser Arbeit nicht gefunden.

Relationen zwischen CPS-Softwarekomponenten

Beziehungen zwischen Softwarekomponenten in cyber-physischen Systemen unterscheiden sich kaum von Beziehungen zwischen Softwarekomponenten in Cloud-Anwendungen. Bei Zugriffen von einer Softwarekomponente auf eine von einem Treiber verwaltete Hardwarefunktion wird dies durch ein Relationship Template dargestellt. Bei diesem Zugriff muss möglicherweise zusätzlich in dem diese Verbindung darstellenden Relationship Template angegeben werden, auf welche von mehreren Funktionen zugegriffen werden soll. Ähnlich wie bei einem direkten Zugriff auf eine Hardwarefunktion durch die Angabe der entsprechenden ID kann dies in einem RelationshipConstraint des Relationship Template erfolgen. Ist diese ID allerdings erst zur Laufzeit bekannt, so verhindert dies eine Angabe im Modell. Soll eine Softwarekomponente eines cyber-physischen Systems mit einer weiteren auf einer anderen Hardwarekomponente kommunizieren, so ist nicht zwangsläufig gegeben, dass diese Kommunikation wie zwischen den Komponenten einer Cloud-Anwendung über ein IP-Netzwerkprotokoll erfolgt. Dies ist beispielsweise in der Fallstudie zwischen der Python-Softwarekomponente zum Steuern des Fahrzeugs und den Softwarekomponenten auf den Arduino-Modulen der Fall. Dies muss beim Implementieren eines Systems zum Deployment und Management solcher in TOSCA modellierten Systeme bedacht werden.

Bei cyber-physischen Systemen ist es häufig der Fall, dass Softwarekomponenten auf unterschiedlichen Hardwarekomponenten ausgeführt werden und über eine Hardwareverbindung miteinander kommunizieren. eine solche Kommunikationsverbindung kann in TOSCA als Relationship Template zwischen diesen beiden Softwarekomponenten angegeben werden. Ist durch die Verbindung von Hardwarekomponenten lediglich eine einzige Möglichkeit zur Kommunikation gegeben, so kann diese automatisch ermittelt werden. Daraufhin können die Informationen der daran beteiligten Hardwareknoten sowie derer Capabilities und Requirements zum Einrichten der Verbindung verwendet werden. Unter Umständen ist diese automatische Erkennung auch bei der Existenz mehrerer möglicher Verbindungen über Hardwarekomponenten möglich, auf diese Erkennung wird jedoch an dieser Stelle nicht weiter eingegangen.

6 Prototypische Implementierung und Analyse

Um Anwendbarkeit der in dieser Arbeit vorgestellten Konzepte zu überprüfen, wurde das in Kapitel 4 vorgestellte Roboterfahrzeug in TOSCA mit Hilfe des Modellierungswerkzeugs Winery aus OpenTOSCA dargestellt. Ein Deployment anhand dieser Darstellung wurde nicht durchgeführt. Im Folgenden soll das Modell des Fahrzeugs sowie typische Details davon erklärt werden. Ebenso werden die Vorgehensweise und Entscheidungen in diesem Prozess erläutert. Anschließend werden die daraus resultierenden Folgen für das Modellieren, Deployment und Management eines cyber-physischen Systems in TOSCA erklärt. Daraufhin werden Erweiterungen für die grafische Darstellung eines solchen Modells genannt, welche in die Topologiedarstellung von Winery zum besseren Verständnis implementiert wurden. Diese Implementierung diente ebenso zum Überprüfen der Praktikabilität der Darstellung. Am Ende dieses Kapitels folgen dann einige Lehren, welche in dieser Arbeit aus dem Modellieren von cyber-physischen Systemen gezogen wurden.

6.1 Modell des autonomen Fahrzeugs

Nach den in Kapitel 5 genannten Vorgehensweisen wurde nun das in Kapitel 4 vorgestellte autonome Fahrzeug in TOSCA dargestellt. Hierbei wurden für den Fokus der Arbeit nur die elektronischen Komponenten sowie die Softwarekomponenten beachtet, das Chassis sowie Montagekomponenten sind somit nicht abgebildet. Eine grafische Darstellung der Komponenten und Relationen ist in Abbildung 6.1 zu sehen. Da in diesem Modell verschiedenste Komponententypen verwendet wurden, welche auf unterschiedlichste Arten miteinander verbunden sind, soll im Folgenden auf die Details der einzelnen Systemteile eingegangen werden. Um in späteren Abschnitten vorgestellte Erweiterungen des Editors zu ermöglichen, wurden neue NodeTypes definiert, von welchen weitere NodeTypes abgeleitet wurden. Es wurde der NodeType *HardwareNode* sowie von ihm abgeleitet der NodeType *CableNode* erstellt. Diese NodeTypes haben an sich keinerlei Eigenschaften und sind lediglich für verschiedene Darstellungsmodi sowie zum Erkennen von Kabelverbindungen relevant. Für die Verbindungen zwischen Hardwarekomponenten wurde ein neuer Relationship +Type namens *ConnectsToHW* erstellt. Dies dient vorwiegend der besseren Übersicht über das System, da hiermit erkennbar ist, welche Verbindungen zwischen Hardwarekomponenten verlaufen und welche Verbindungen Softwarekomponenten beinhalten. Auch kann damit in Zukunft für den Topologieeditor von Winery angegeben werden, dass ein RelationshipTemplate dieses Typs lediglich zwischen Hardwarekomponenten aufgebaut werden darf. Zusätzlich wurden eine Vielzahl an CapabilityTypes und zugehörigen RequirementTypes benötigt und erstellt. Diese werden zu gegebener Zeit im Text genannt und erklärt. Um ein weiteres Abstraktionslevel zu untersuchen, wurde die primitive Schaltung des *Signalmerger* als eine Komponente dargestellt, obwohl sie aus mehreren Komponenten auf dem Steckbrett aufgebaut wurde. Da viele verschiedene Kabelverbindungen über das Steckbrett verlaufen wurden zum besseren Verständnis jeweils zwei Reihen von fünf verbundenen Steckkontakten eines Steckbretts als virtueller NodeType *MultiIn_MultiOut* dargestellt.

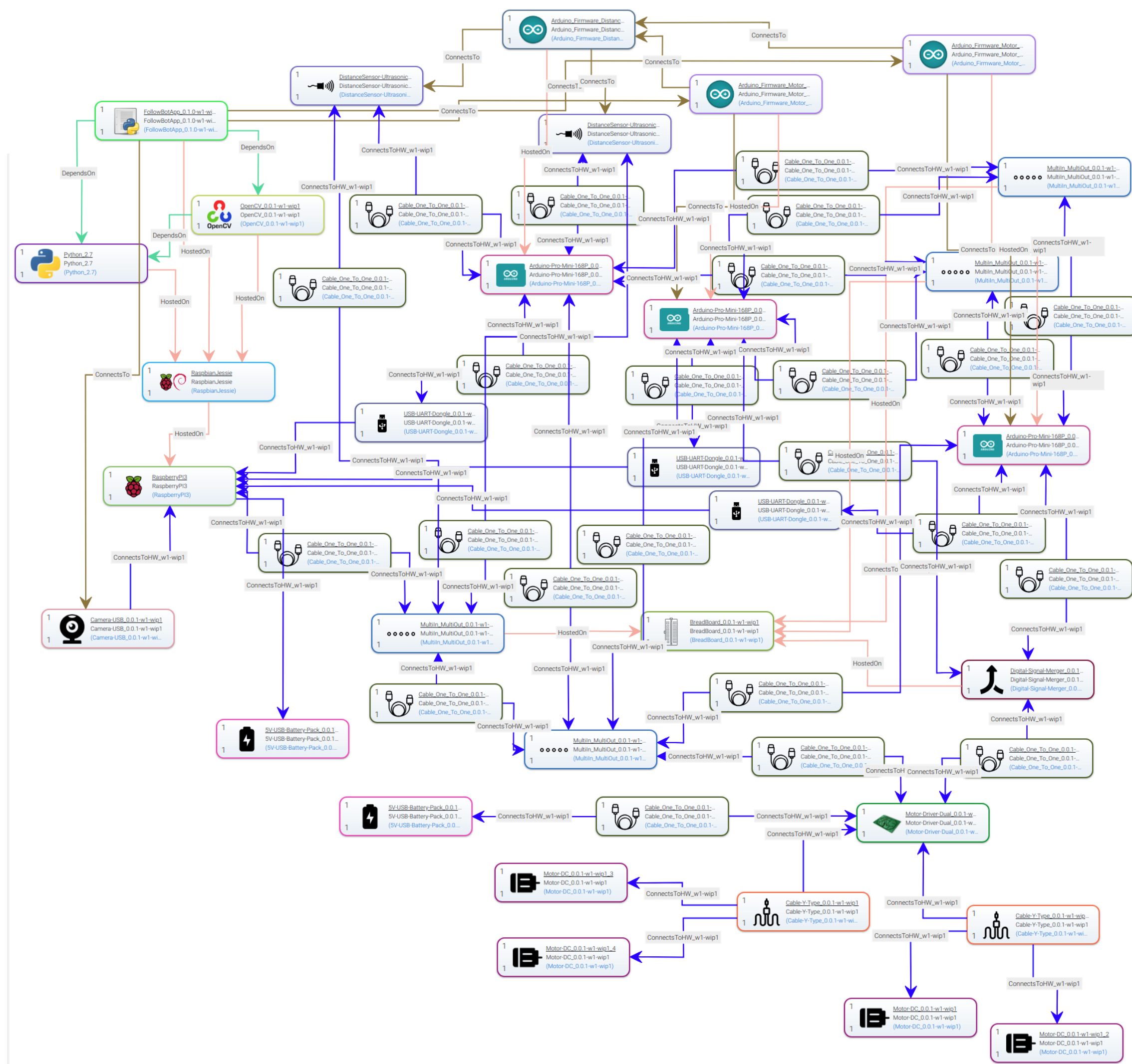


Abbildung 6.1: Gesamtansicht aller Komponenten und Beziehungen im Modell des autonomen Fahrzeugs in TOSCA. Ansicht mit dem Topologymodeler von Winery generiert.

6.1.1 Stromversorgung

Die einzigen Komponenten ohne Requirements in diesem System sind die beiden Batterien sowie das Steckbrett. Die Batterien bieten über einen USB-Anschluss eine Stromversorgung an, das Steckbrett die Möglichkeit für Verbindungen zwischen bis zu fünf Kabeln oder weiterer Komponenten, welche auf ihm eingesteckt werden können. Entsprechend wurde für diese Batterien ein vom Typ *HardwareNode* abgeleiteter NodeType *5V-USB-Battery-Pack* erstellt und der CapabilityType *PowerSupply* für diesen erstellt. Da eine der Batterien eine USB-Buchse besitzt, die andere jedoch ein Kabel mit Micro-USB-Stecker fest verbaut hat, wurden zwei mögliche Capabilities für den Node Type der Batterien definiert. Jede dieser Capabilities hat einen Constraint, welcher den entsprechenden Anschluss darstellt. Entsprechend wurde bei jedem der beiden NodeTemplates die dem Anschluss entsprechende Capability angegeben. Dessen einfache Darstellung in XML wird in Listing 6.1 gezeigt. Hierbei werden sämtliche Hardwareports, über welche eine Capability bezogen

werden kann, aufgelistet als *ports*. Zusätzlich werden die einzelnen Verbinder jedes Ports aufgelistet, hier *connectors* genannt. In diesem Fall besitzt die Komponente für diese Funktion lediglich einen Hardwareport, welcher wiederum aus nur einem Verbinder besteht. Der angegebene Verbindertyp hat das Attribut *gender="male"*. Dies wird benötigt um bei identischem Verbindertyp männliche und weibliche Verbinder als zueinander passend zu erkennen. Details zum Verifizieren bestimmter Eigenschaften wie die angebotene Spannung oder maximal verfügbare Stromstärke können dabei in weiteren Constraints oder im selben Constraint wie die Verbinder angegeben werden. In manchen Fällen sind diese Eigenschaften für jeden Hardwareport unterschiedlich, dadurch bietet es sich an, die Eigenschaften jedes Hardwareports an dieser Stelle darzustellen. Da diese Eigenschaften irrelevant für elektrische Verbindungen sind und lediglich für Verifikationen notwendig sind, werden sie als Attribute des entsprechenden *ports* angegeben.

Dass eine Batterie direkt an das RaspberryPi angeschlossen werden soll, wird durch ein Relationship Template vom Typ *ConnectsToHW* dargestellt. Da der Anschluss der Batterie, welche an das RaspberryPi angeschlossen werden soll, lediglich mit einem Anschluss des RaspberryPi kompatibel ist, wurde dem verbindenden Relationship Template keine weitere *Property* mit Informationen über die zu verbindenden Anschlüsse hinzugefügt. Durch den Abgleich der Verbindertypen ist eine Zuordnung der zu verbindenden Anschlüsse leicht möglich.

Das RaspberryPi hat mit vier USB-Anschlüssen, Anschlüssen für Grafik und Soundausgabe, einer Netzwerkbuchse, WLAN, Bluetooth und verschiedensten Hardwareports über seine GPIO-Anschlussleiste eine Vielzahl an Hardwarefähigkeiten, welche über diese Verbinder angeboten werden. Zusätzlich benötigt es eine Stromversorgung, entweder über einen Micro-USB-Anschluss oder aber über Anschlüsse der GPIO-Anschlussleiste. Daher wurde für das RaspberryPi der Version 3 ein NodeType *RaspberryPi3* erstellt und mit einem Requirement des neu erstellten CapabilityTypes *Power-Supply* versehen, welcher auf die zuvor genannte Capability *PowerSupply* als benötigt verweist. Ebenfalls wurde diesem Requirement ein Constraint hinzugefügt, welches die Verbinder auflistet, über welche die Stromversorgung bezogen werden kann. Diese Details sind teilweise in Abbildung 6.2 zu sehen, jede Anschlussmöglichkeit wird darin als *port* mit den entsprechenden Verbindern angegeben. Zusätzlich wird für jeden Verbinder seine Funktion angegeben. Damit kann durch Verbinden von Verbindern mit entsprechender Funktionen ein inkorrektes Anschließen von Kabeln vermieden werden, da dies andernfalls beispielsweise zu Kurzschlüssen führen kann.

Da die Stromversorgung des RaspberryPi intern weitergeleitet wird an Anschlüsse seiner GPIO-Leiste und von dort für die Stromversorgung weiterer Komponenten genutzt wird, wurde diese Fähigkeit dem NodeTemplate des RaspberryPi als Capability hinzugefügt. Auch dieser Capability wurde ein Constraint hinzugefügt, in welchem die Hardwareports und deren Anschlüsse definiert wurden. Da es sich bei diesen Anschlüssen um jeweils zwei einzelne Steckanschlüsse handelt, wurden pro Hardwareport zwei Verbinder angegeben. Dies ist in Listing 6.2 zu sehen, die Namen der Verbinder wurden aus der Nummerierung der Anschlussbelegung entnommen und sind somit für Menschen auffindbar. Auch hierbei wurde zur Vermeidung von Fehlkonfigurationen zusätzlich bei jedem Verbinder dessen Funktion angegeben. Dies kann bei Relationen zu dieser Komponente für eine automatische Erkennung der zu verbindenden Verbinder genutzt werden. Dass diese Verbinder sowohl in Hardwareports der Capability einer Stromversorgung als auch dem Requirement nach einer Stromversorgung angegeben sind, widerspricht sich nicht, denn sobald ein Hardwareport für eines davon belegt ist, steht er für das andere nicht mehr zur Verfügung. Ist eine mehrfache Verwendung der Verbinder modelliert, so kann dies durch die einheitliche Bezeichnung automatisch erkannt und vermieden werden.

Listing 6.1 Anschluss des Hardwareports einer Batterie in einem Constraint dargestellt.

```
<tosca:Constraint xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12">
  <ports xmlns:[..]>
    <port voltage="5V" maxAmperage="1A">
      <portId>USB-A</portId>
      <connectors>
        <connector>
          <name>USB-A</name>
          <function>power</function>
          <connectorType gender="male">Micro-USB-A</connectorType>
        </connector>
      </connectors>
    </port>
  </ports>
</tosca:Constraint>
```

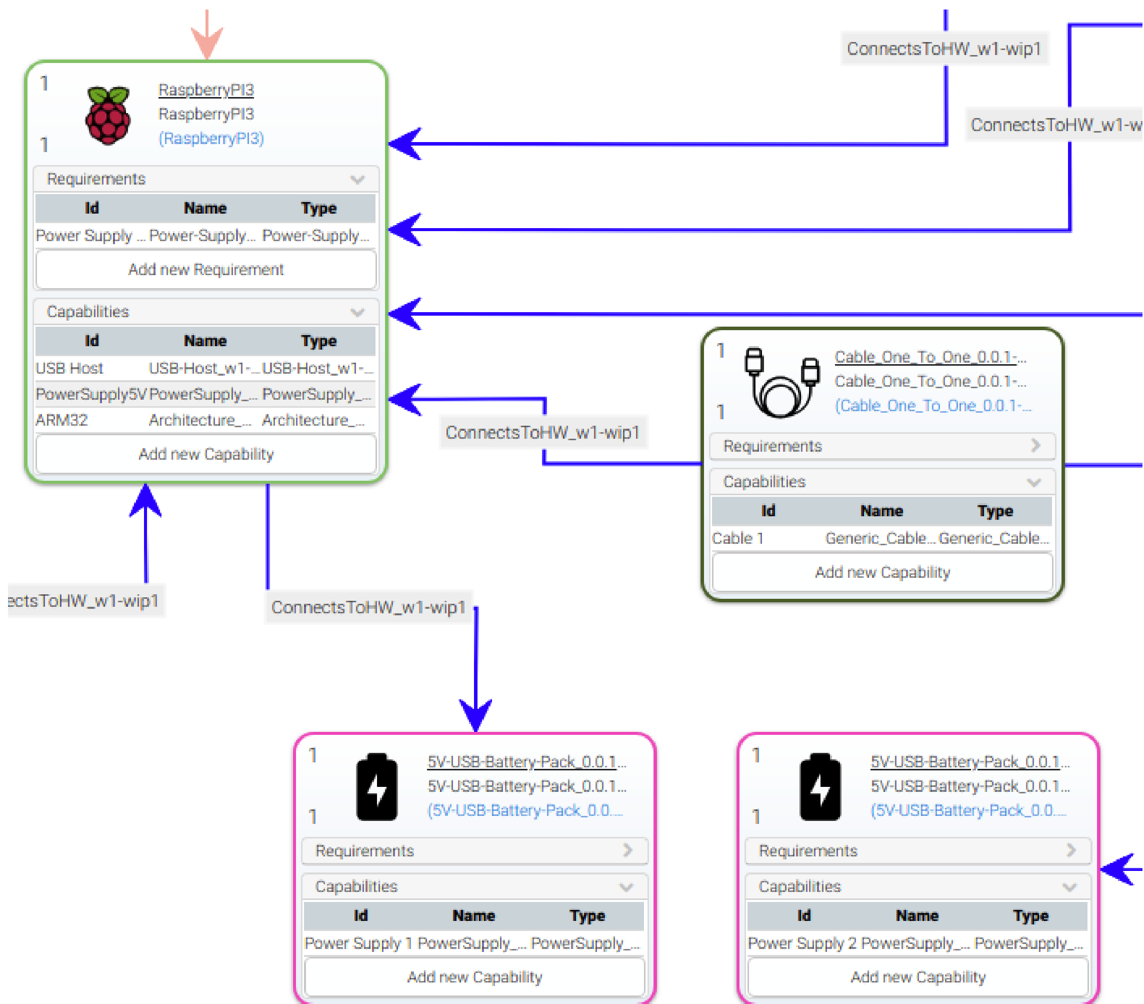


Abbildung 6.2: Ausschnitt des in TOSCA erstellten Modells des autonomen Fahrzeugs mit angezeigten Capabilities und Requirements der Batterien und des RaspberryPis.

Listing 6.2 Die Anschlüsse der Hardwareports der ausgehenden Stromversorgung eines RaspberryPi, dargestellt in einem Constraint.

```
<tosca:Constraint xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12">
  <ports xmlns:[...]>
    <port>
      <connectors>
        <connector>
          <name>Pin_4</name>
          <function>+5V</function>
          <connectorType gender="male">pin</connectorType>
        </connector>
        <connector>
          <name>Pin_6</name>
          <function>GND</function>
          <connectorType gender="male">pin</connectorType>
        </connector>
      </connectors>
    </port>
    <port>
      <connectors>
        <connector>
          <name>Pin_2</name>
          <function>+5V</function>
          <connectorType gender="male">pin</connectorType>
        </connector>
        <connector>
          <name>Pin_9</name>
          <function>GND</function>
          <connectorType gender="male">pin</connectorType>
        </connector>
      </connectors>
    </port>
  </ports>
</tosca:Constraint>
```

6.1.2 Antriebsstrang

Der Antriebsstrang des Fahrzeugs besteht ausschließlich aus Hardwarekomponenten ohne Computingfähigkeiten. Die Motortreiberkomponente für Motoren, bezeichnet als *Motor_Driver_Dual*, benötigt um ihrer Funktion nachkommen zu können eine Stromversorgung für ihre Logikeinheiten, eine separate Stromversorgung für die Steuerung der Motoren sowie Steuersignale. Ist dies gegeben, so bietet sie das Steuern von zwei Motoren an. Entsprechend wird diese Komponente mit drei Requirements und einer Capability dargestellt. Dass die Komponente an zwei Stromversorgungen angeschlossen werden muss wurde nicht als ein Requirement dargestellt, welches über zwei Hardwareports erfüllt werden kann. Das kommt daher, dass zum einen durch den Anschluss einer Stromversorgung nicht die gesamte Stromversorgung des Moduls erfüllt wird und zum anderen die Stromversorgung für die Motoren weitaus höhere Spannungen liefern darf als die Stromversorgung der Logikeinheiten. Die an der Motortreiberkomponente angeschlossenen Kabel werden mit ei-

Listing 6.3 Anschlüsse eines der Hardwareports von Y-Kabeln in einem Constraint dargestellt.

```
<tosca:Constraint xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12">
  <ports xmlns:[...]>
    <port>
      <connectors>
        <connector>
          <name>red_1</name>
          <connectorType gender="male">solder_terminal</connectorType>
          <connectsTo>red_2</connectsTo>
          <connectsTo>red_3</connectsTo>
        </connector>
        <connector>
          <name>black_1</name>
          <connectorType gender="male">solder_terminal</connectorType>
          <connectsTo>black_2</connectsTo>
          <connectsTo>black_3</connectsTo>
        </connector>
      </connectors>
    </port>
    [...]
</ports>
</tosca:Constraint>
```

ner generischen Capability *Generic_Cable_Capability* dargestellt, um wie auch bei den anderen Hardwarekomponenten in einem Constraint die Anschlüsse definieren zu können. Hierbei werden die Hardwareports dieses Kabels in einem Constraint wie in Listing 6.3 gezeigt angegeben. Wie dabei zu sehen ist, wurde für jeden Verbinder angegeben, zu welchen weiteren Verbindern eine interne Verbindung von ihm ausgehend besteht. Die Motoren wurden als NodeTemplates vom NodeType *Motor-DC* (DC = "Direct Current", Gleichstrom) mit dem Requirement *Motor-Driver-Req* dargestellt. Dabei wurde in einem Constraint ein Hardwareport mit den beiden Anschlüssen angegeben, die Funktionen dieser wurden als + und - angegeben. Bei dem NodeTemplate, welches das Motortreibermodul darstellt, wurde die zugehörige Capability *Motor-Driver* angegeben und in einem Constraint dazu die beiden Hardwareports mit den entsprechenden Anschlüssen, welche ebenfalls die Funktionen + beziehungsweise - haben.

Die Anforderung einer Stromversorgung, die Batterie und ihre Fähigkeit einer Stromversorgung und der Anschluss der Batterie an das Motortreibermodul wurden ähnlich der beschriebenen Stromversorgung des RaspberryPi realisiert. Lediglich ein zusätzliches Kabel sowie andere Verbinder der Hardwareports sind der Unterschied. Das Kabel ist wie alle anderen Kabel als NodeTemplate vom NodeType *Cable_One_To_One* dargestellt. Die Anschlüsse dieses Kabels sind wiederum zu Hardwareports zusammengefasst in einer *Generic_Cable_Capability* des NodeTemplates angegeben.

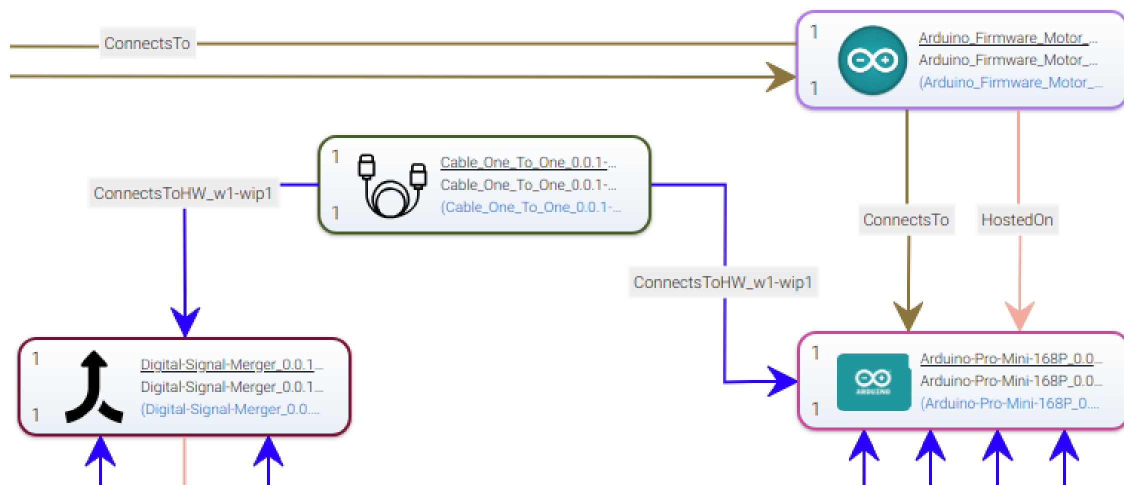


Abbildung 6.3: Detail des in TOSCA erstellten Modells des autonomen Fahrzeugs, zu sehen eine Softwarekomponente zum Ansteuern des Motortreibermoduls sowie zugehörige Hardwarekomponenten welche diese Verbindung und das Ausführen der Softwarekomponente ermöglichen.

6.1.3 Arduino-Software

Die Arduino-Module werden in diesem Modell durch NodeTemplates vom NodeType *Arduino-Pro-Mini-168P* dargestellt. Die darauf installierte und ausgeführte Software zur Ansteuerung des Motortreibermoduls wird durch NodeTemplates vom NodeType *Arduino_Firmware_Motor_Driver* dargestellt. Wie in Abbildung 6.3 zu sehen ist, verbindet sich die Arduino-Software mit der Komponente *Digital-Signal-Merger*, welche Signale aus zwei Quellen annimmt und an das Motortreibermodul weitergibt. Dieses Verbinden stellt dar, dass die Softwarekomponente auf vier bestimmte Ports zugreift, über welche sie dann Signale an die Signalmerger-Komponente senden kann. Da die für die Fallstudie entwickelte Arduino-Firmware nur vor dem Kompilieren konfiguriert werden kann, müssen diese vier Zugriffe bei der Arduino-Firmware in einem Schritt eingerichtet werden, um daraufhin die Firmware zu kompilieren und auf das entsprechende Arduino zu installieren. Dies ist ein Workaround für das in diesem Fall nicht nutzbare Prinzip von TOSCA, erst Softwarekomponenten auszubringen und danach für Kommunikationsverbindungen zu konfigurieren. Jedoch sind die meisten Softwarekomponenten für Mikrocontroller auf eine Weise geschrieben, in der sie durch Konfigurationsdateien des Quellcodes konfiguriert werden, um daraufhin kompiliert und installiert zu werden. Ebenfalls ist auf diese Weise lediglich ein RelationshipTemplate für die Verbindung notwendig, wodurch das Modell etwas übersichtlicher wird. Werden entsprechende Frameworks oder Bibliotheken bei der Entwicklung von Softwarekomponenten für Mikrocontroller verwendet, so können diese Softwarekomponenten auch nach dem Installationsvorgang konfiguriert werden. Dadurch können bei der Modellierung sämtliche einzelnen Zugriffe einer solchen Softwarekomponente auf eine Hardwarefunktion dargestellt werden.

6.1.4 Adapter

An dem RaspberryPi sind über USB-Anschlüsse Adapter angeschlossen, welche über einen Hardwareport eine serielle Schnittstelle anbieten. Mit diesem Hardwareport ist ein Kabel verbunden, welches wiederum mit seinem anderen Ende mit dem Hardwareport einer seriellen Schnittstelle eines Arduino-Modul verbunden ist. Die Darstellung dieser Verbindung wurde schon konzeptionell in 5.8 aufgezeigt und entsprechend wurde dies im Modell dargestellt. Die an dem RaspberryPi angeschlossenen Adapter bieten jeweils die Funktion einer seriellen Schnittstelle (UART) über einen Hardwareport an. Entsprechend bieten sie den Zugriff auf diesen Hardwareport auch Softwarekomponenten an. Daher wurden diese Adapter als NodeTemplates vom NodeType *USB-UART-Dongle* dargestellt, welche das Requirement *USB-Host* auf eine entsprechende Capability haben. Zusätzlich haben diese NodeTemplates die Capability *UART-HW*, welche die Anschlussmöglichkeit einer seriellen Schnittstelle darstellt sowie die Capability *UART-SW*, welche Softwarekomponenten den Zugriff auf diese serielle Schnittstelle bietet. Bei jedem NodeTemplate der Adapter geht von dem Requirement *USB-Host* ein RelationshipTemplate vom RelationshipType *ConnectsToHW* zu der entsprechenden Capability *USB-Host* am NodeTemplate des RaspberryPi. Die Adapter dürfen an beliebige USB-Anschlüsse des RaspberryPis angeschlossen werden, daher sind keine weiteren Angaben hierfür in dem RelationshipTemplate nötig. Die Adapter sind jeweils über ein Kabel mit dem Hardwareport einer seriellen Schnittstelle des Arduino-Moduls verbunden. Diese Kabel werden durch NodeTemplates vom NodeType *Cable_One_To_One* dargestellt. Da die Hardwareports dieses Kabels aus anderen Anschlüssen bestehen wie bisher angegebene Kabel, wurde diesem NodeType eine weitere Capability vom CapabilityType *Generic_Cable_Capability* sowie ein entsprechendes Requirement hinzugefügt. In einem Constraint dieser wurden daraufhin die Hardwareports des Kabeltyps sowie deren Verbinder definiert.

6.1.5 Verbindungen über Adapter

Über diese im vorherigen Abschnitt definierten Verbindungen zwischen Hardwarekomponenten besteht bei dem als Fallstudie genutzten autonomen Fahrzeug eine Kommunikationsverbindung zwischen zwei Softwarekomponenten. Bei diesen Softwarekomponenten handelt es sich um die Software zur Steuerung des Fahrzeugs (*FollofBotApp*) auf dem RaspberryPi sowie um die Software zur Ansteuerung des Motortreibermoduls. Zum besseren Verständnis ist der hierfür relevante Teil der in Abbildung 6.4 gezeigten Komponenten nochmals mit Requirements und Capabilities in 6.5 gezeigt. Die Softwarekomponente zur Steuerung des Fahrzeugs ist auf dem Betriebssystem Raspbian gehostet, welches wiederum auf dem RaspberryPi gehostet ist. Die Softwarekomponente zur Ansteuerung des Motortreibermoduls ist auf einem Arduino-Modul gehostet. Die Kommunikation zwischen den Softwarekomponenten erfolgt über eine serielle Schnittstelle, welche von einem Adapter von USB zu Seriell gestellt wird. Die Verbindung von diesem Adapter über ein Kabel zu dem Ardunino-Modul sowie deren Darstellung im Modell wurde im vorhergehenden Abschnitt erklärt. Die Softwarekomponente zum Steuern des Fahrzeugs besitzt ein Requirement vom RequirementType *Motor_Driver*, die Softwarekomponente zum Steuern des Motortreibermoduls bietet die entsprechende Capability an. In dieser Capability ist die Kommunikation über eine serielle Schnittstelle definiert. Die Kommunikationsverbindung zwischen den beiden Softwarekomponenten wird hier entsprechend als RelationshipTemplate vom RelationshipType *ConnectsTo* zwischen dem Requirement der Softwarekomponente zur Fahrzeugsteuerung und der zum Steuern des Motortreibermoduls angegeben. Wie schon genannt, ist die Darstellung des Zugriffs von

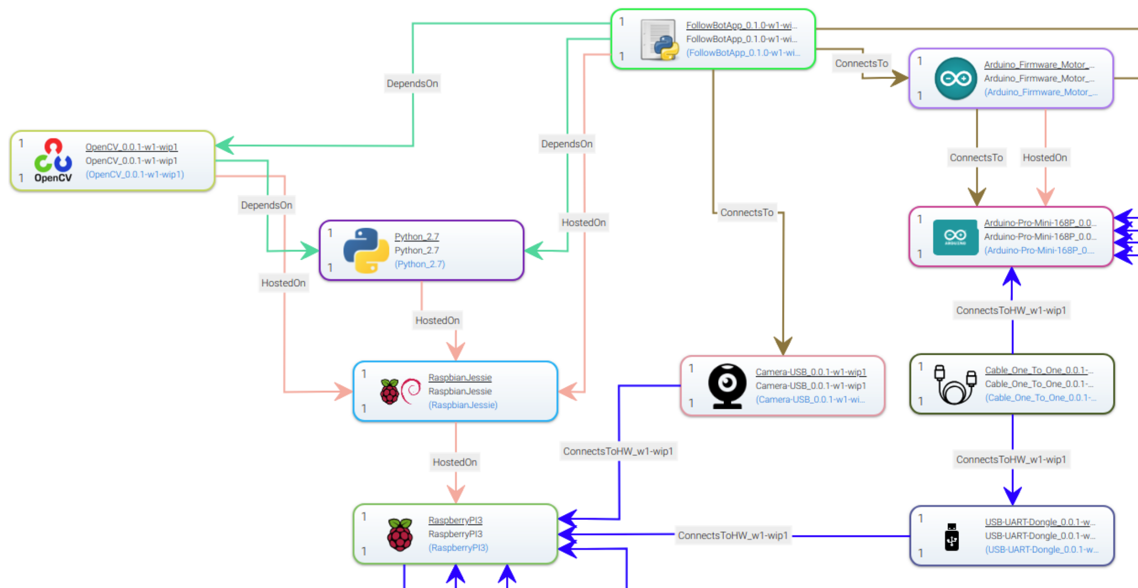


Abbildung 6.4: Ausschnitt des in TOSCA erstellten Modells des autonomen Fahrzeugs, zu sehen eine Softwarekomponente zum Ansteuern des Motortreibermoduls sowie zugehörige Hardwarekomponenten welche diese Verbindung und das Ausführen der Software ermöglichen.

Softwarekomponenten auf Adapter beziehungsweise Erweiterungsmodule in TOSCA teils äußerst schwierig. Dies ist auch bei den Adaptern von USB zu Seriell der Fall. Um die SoftwareID eines Hardwareports dieser seriellen Schnittstellen zu ermitteln, muss diese ausgehend von der ID der in der Capability angegebenen USB-Hardwareports generiert werden. Da der Treiber für diese Adapter Teil des Betriebssystems ist, sollte dieser nicht dargestellt werden. Allerdings muss dennoch eine entsprechende Capability in dem NodeType des RaspberryPi namens *USB-Host-SW* erstellt werden, dem wiederum ein Constraint hinzugefügt wurde, in welchem die Referenz der USB-Anschlüsse als Ports angegeben wurden. Dabei wurde bei den Ports zusätzlich der Gerätepfad der USB-Anschlüsse in Linux als SoftwareID angegeben. Nachdem diese Capability als auf die Capability für Hardwarekomponenten verweisend erkannt wurde, lassen sich nun anhand dieser SoftwareIDs und der an die entsprechenden Hardwareports angeschlossenen Adapter die SoftwareIDs für den Zugriff auf die Hardwareports der seriellen Schnittstellen ermitteln. Um von der Softwarekomponente zur Fahrzeugsteuerung ausgehend die ID für den Zugriff einer auf die entsprechende serielle Schnittstelle zu erfahren, muss auf die der Softwarekomponente unterliegende Hardwarekomponente zugegriffen werden. Hierbei führt eine Relation vom RelationshipType *HostedOn* zum NodeTemplate des Betriebssystems Raspbian und von dort aus eine weitere zum NodeTemplate des RaspberryPi. Da von dem NodeTemplate des RaspberryPi keine Capability einer seriellen Schnittstelle ausgeht, muss nach angeschlossenen Adaptern gesucht werden, welche eine solche Capability besitzen. Wurde dadurch die Adapter gefunden, so muss über deren Verbindungen der Adapter gefunden werden, der mit der Hardwarekomponente verbunden ist, welche die Softwarekomponente zum Steuern des Motortreibermoduls hostet. Sind alle diese Hardwarekomponenten gefunden, so können anhand der Capabilities und Requirements, welche die serielle Schnittstelle sowie die USB-Anschlüsse

darstellen, nach den zugehörigen Capabilities für Softwarekomponenten gesucht werden. Daraufhin können anhand der genutzten Hardwareports die entsprechenden SoftwareIDs ermittelt und mittels dieser die Verbindung zwischen den Softwarekomponenten eingerichtet werden

6.2 Analyse für Modellierung, Deployment und Management

Soll ein System mit Hardwarekomponenten, welches in TOSCA modelliert wurde, bereitgestellt werden, so müssen die Schritte zum Bereitstellen dieser Hardwarekomponenten sowie der teils direkt darauf auszuführenden Softwarekomponenten durch physische Aktionen ausgeführt werden. Ebenfalls müssen die Verbindungen zwischen Hardwarekomponenten hergestellt werden, auch dies kann nur durch physische Aktionen ausgeführt werden. Zum Deployment von Softwarekomponenten durch Menschen wurden von [ICK+10] sogenannte *Human Tasks* vorgestellt. Da solche Aufgaben allerdings prinzipiell auch durch entsprechend ausgerüstete und programmierte Roboter ausgeführt werden könnten, werden sie in dieser Arbeit als *Physical Tasks* bezeichnet. Rein im Virtuellen von Software ausführbare Aufgaben werden im Gegenzug als *Virtual Tasks* bezeichnet.

Um solche Physical Tasks auszuführen sind ähnlich der Virtual Tasks Informationen über die beteiligten Komponenten sowie für den auszuführenden Prozess nötig. Beispielsweise sind für das Erstellen einer Verbindung die Informationen über die beteiligten Komponenten sowie der zu nutzenden Verbinder nötig. Ebenfalls ist hierfür die Information notwendig, welcher Verbinder mit welchem weiteren verbunden werden soll. Diese Informationen sollten für ein automatisiertes Deployment von Systemen, wie es für in TOSCA erstellte Systeme möglich sein soll, automatisch erkannt und extrahiert werden, selbst wenn sie nur implizit angegeben sind.

Zusätzlich ergeben sich Möglichkeiten aus der vorgeschlagenen Modellierung von Systemen um weitere Informationen zu generieren, anhand derer beispielsweise Möglichkeiten für das Ersetzen von Physical Tasks durch Virtual Tasks gefunden werden können. Ebenfalls können mit diesen Informationen Modellierer unterstützt werden, wie in Abschnitt 6.3 gezeigt wird.

6.2.1 Generierbare Informationen

Aus Systemen, welche nach den in dieser Arbeit vorgestellten Prinzipien dargestellt sind, können verschiedene weitere Informationen generiert werden. Diese können beim Modellieren eines cyber-physischen Systems unterstützen, indem sie beim Verständnis des Systems helfen. Ebenso können diese Informationen beim Deployment eines solchen Systems genutzt werden, um vorhandene Teile des Systems für das Deployment weiterer Teile zu nutzen. Im Folgenden werden die im Rahmen dieser Arbeit erkannten generierbaren Informationen sowie deren Nutzen vorgestellt.

Netzwerke, Strom- und Datenverbindungen

Bei einer einfachen Darstellung von Kabeln als Nodes mit zwei Requirements oder einer Capability und einem Requirement bietet es sich wie genannt an, für Kabel einen eigenen vom Hardware Node Type abgeleiteten Node Type einzuführen, von welchem wiederum die einzelnen Kabel Node Types abgeleitet sind. Da alle auf diese Weise dargestellten Kabel sämtliche ihrer Anschlüsse miteinander

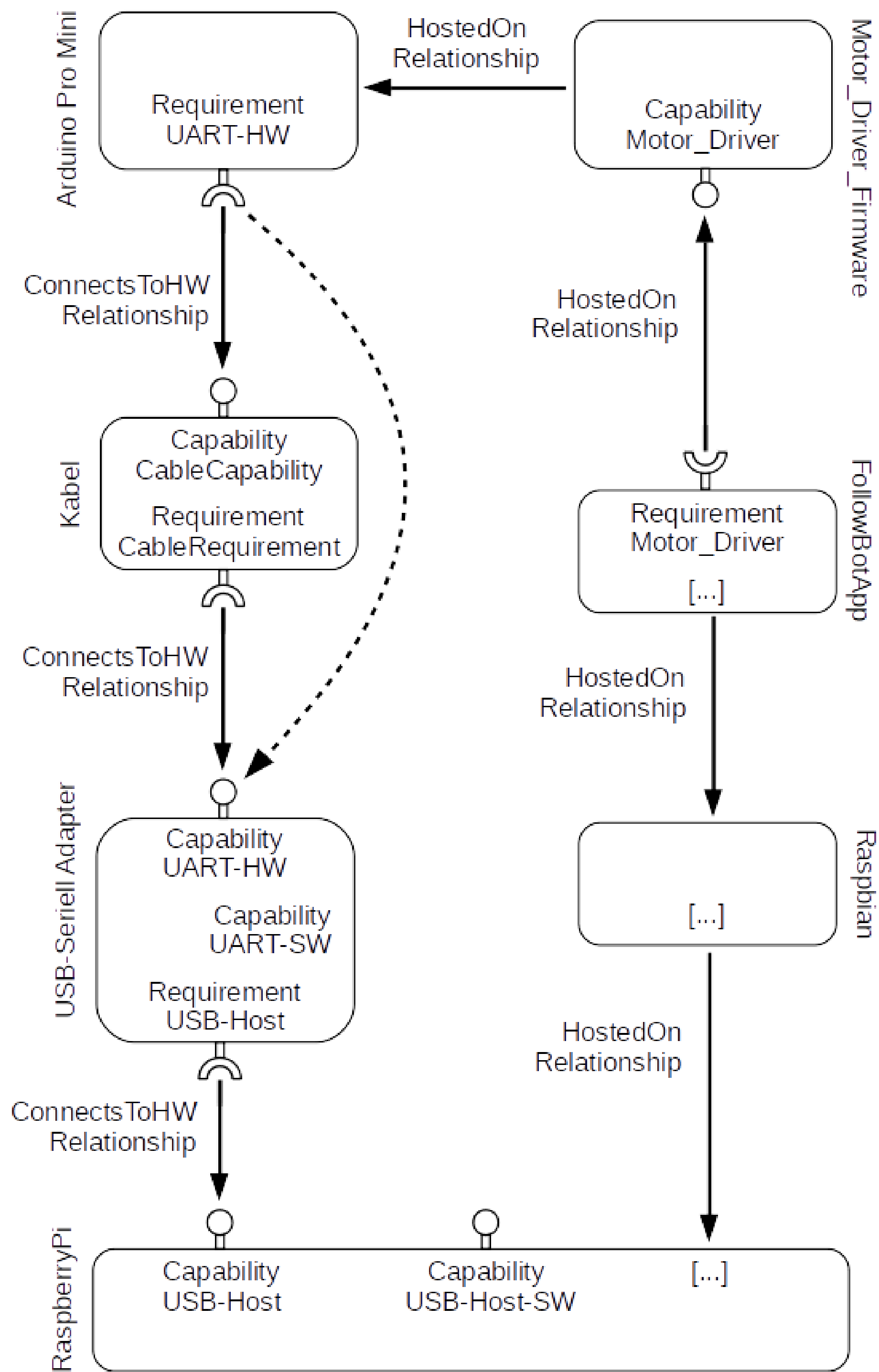


Abbildung 6.5: Grafische Darstellung der in 6.1.5 genannten Komponenten mit Capabilities und Requirements sowie deren Relationen untereinander.

verbinden, ermöglicht dies eine einfache automatische Erkennung aller direkt möglichen Daten-, Signal- oder Stromverbindungen über auf diese Art dargestellte Kabel. Dies ist mit bekannten Algorithmen zur Breitensuche oder Tiefensuche realisierbar.

Ebenfalls existiert auf dieser Basis aufbauend die Möglichkeit, dass verifiziert werden kann, ob eine Kommunikationsverbindung zwischen zwei Softwarekomponenten durch die Verbindungen der Hardwarekomponenten ermöglicht wird. Dabei kann anhand der Capabilities der Komponenten validiert werden, ob die Softwarekomponenten für diese Datenverbindung die entsprechenden Schnittstellen unterstützen. Ist dies nicht gegeben, so kann dies dem Modellierer angezeigt werden und ein Deployment eines solchen Systems verhindert werden.

Abhängigkeiten beim Deployment

Softwarekomponenten können auf weiteren Softwarekomponenten oder virtuellen Maschinen installiert sein. Dies wird in TOSCA durch eine Relation vom Relationship Type *HostedOn* zwischen diesen beiden Komponenten dargestellt. Dabei gibt die Relation an, welche Komponente auf welcher anderen installiert werden muss. Implizit wird dadurch auch die Reihenfolge der Komponenten beim Deployment angegeben, da erst die eine Komponente vorhanden sein muss bevor die andere auf ihr installiert wird. Ähnlich verhält es sich, wenn Hardwarekomponenten auf weiteren montiert werden müssen. Beispielsweise kann in dem als Fallstudie genutzten autonomen Fahrzeug die Komponente ”Digital Signal Merger” erst angebracht werden, wenn das dafür benötigte Steckbrett (”Breadboard”) im System vorhanden ist. Eine weitere Darstellung von Abhängigkeit zwischen Softwarekomponenten ist eine funktionale Abhängigkeit wie sie durch Relationen vom Typ *DependsOn* dargestellt werden. Hierbei ist die Komponente an der Quelle der Relation nicht funktionsfähig ohne die Präsenz der Komponente am Ziel der Relation. Hierdurch wird ebenfalls implizit angegeben, dass die Komponente am Ziel der Relation bereitgestellt werden muss bevor dies mit der Komponente an der Quelle der Relation getan werden kann. Dieser Zusammenhang gilt ebenfalls bei Hardwarekomponenten. Daher muss diese durch Relationen implizierte Reihenfolge beim Deployment von Hardwarekomponenten ebenso beachtet werden wie beim Deployment von Softwarekomponenten.

6.2.2 Deployment von Hardwarekomponenten

Da Hardwarekomponenten physische Objekte sind, können diese bis auf den Bestellvorgang oder das Abrufen aus einem Lagersystem nur durch das Ausführen physischer Aufgaben bereitgestellt werden. Wie diese Bereitstellung konkret abläuft ist Implementierungs- und Systemabhängig, jedoch müssen dem ausführenden Menschen oder Roboter sämtliche relevanten Informationen zur Verfügung gestellt werden. Hierzu gehören die Information über die bereitzustellende Komponente sowie die Information über die Komponente, auf oder an welche diese montiert, angeschlossen oder zu welcher hin sie ausgerichtet werden soll. Zusätzlich ist möglicherweise die Information nötig, welche Konfiguration der Verbinder der Hardwarekomponenten die Montage oder der Anschluss als Ziel haben soll. Dies ist aus der Relation zwischen diesen beiden Komponenten auszulesen. Die hierfür nötigen Abläufe für das Deployment müssen nach dem Prinzip von TOSCA als Pläne dargestellt werden. BPEL4People [KKL+05] stellt *Human-Tasks* als Erweiterung der schon in TOSCA für Pläne vorgeschlagenen Sprache BPEL vor, jedoch ist das erklärte Ziel insbesondere nicht das Darstellen

von Human-Tasks für das Deployment von Komponenten. Möglicherweise ist dieser Standard allerdings erweiterbar um mit ihm Workflows für das Bereitstellen von Hardwarekomponenten sowie Softwarekomponenten auf diesen zu beschreiben.

6.2.3 Deployment von Softwarekomponenten auf Hardwarekomponenten

Wurden die Hardwarekomponenten eines in TOSCA modellierten Systems bereitgestellt, so müssen auf manchen dieser Komponenten weitere Softwarekomponenten bereitgestellt werden. Die Möglichkeiten hierzu sind abhängig von den angebotenen Schnittstellen der Hardwarekomponenten und werden im Folgenden beschrieben.

Ausgelöstes Selbstupdate

Viele Hardwarekomponenten mit Computingfähigkeiten werden schon bei der Herstellung mit einem initialen Bootloader beziehungsweise einer initialen Software mit Kommunikationsfähigkeiten und Funktionen für ein Update der Softwarekomponenten auf dies en Hardwarekomponenten ausgestattet. Soll eine Softwarekomponente auf einer solchen Hardwarekomponente bereitgestellt werden, so kann diese initiale Software oder der Bootloader dafür genutzt werden. Dies kann je nach System zum Beispiel durch ein Update über ein Kommando zum Installieren einer Firmware oder eines Betriebssystems von einer über ein Kommunikationsnetzwerk erreichbaren Quelle erfolgen. Dies ist bei der Verwaltung von Rechenzentren häufig der Fall. Ebenso kann die Möglichkeit bestehen, dass eine Software auf einem Gerät installiert werden kann, indem ein Befehl zum Updaten gesendet wird, gefolgt von dem Maschinencode der zu installierenden Software. Die zuletzt genannte Vorgehensweise wird beispielsweise bei vielen Mikrocontrollern angewendet (AVR, PIC, einige ARM SoC). Hierfür sind auch weitere Konzepte denkbar.

Deployment via Proxy

Manche der im letzten Abschnitt genannten Komponenten sind während dem Deploymentprozess nicht direkt von einer TOSCA-Runtime erreichbar. Entsprechend können Softwarekomponenten nicht direkt durch das Ausführen von Deployment Artifacts in dieser Runtime auf diese Komponenten bereitgestellt werden. Unter Umständen kann jedoch eine weitere Komponente hierfür verwendet werden, welche mit der entsprechenden Komponente verbunden ist. Dabei kann die TOSCA-Runtime diese weitere Komponente als Proxy verwenden, indem sie auf ihr eine Routine ausführen lässt, welche wiederum eine Softwarekomponente auf der nicht direkt erreichbaren Komponente installiert. Ein solches Update kann je nach Komponententyp auf verschiedene Arten durchgeführt werden. So werden zum Beispiel Arduino-Module mit einer Bootloader-Software ausgeliefert, über welche nach jedem Reset für einige Sekunden über eine Serielle Schnittstelle ein Befehl zum Installieren weiterer Software auf dieses Modul gesendet werden kann [ARD15b]. Bei anderen Modulen muss während einem Reset an einigen Anschlüssen ein bestimmtes Signal anliegen, um danach Software auf ihnen installieren zu können. Dies ist beispielsweise bei Modulen anzutreffen, welche auf dem SoC ESP8266 basieren [ESP18]. Android-Geräte hingegen können bei entsprechender Konfiguration über ihren USB-Anschluss mit Softwarekomponenten bespielt werden. Zur Installation von Software

auf einer Hardwarekomponente muss in jedem dieser Fälle eine weitere Software auf der Proxy-Komponente ausgeführt werden. Diese Software kann als Deployment Artifact des Node Types der Ziel-Hardwarekomponente oder der zu installierenden Softwarekomponente vorliegen. Unter Umständen muss diese Software auf Proxy-Nodes verschiedener Node Types ausgeführt werden. In diesem Fall muss für jeden Node Type dieser Nodes ein eigenes Deployment Artifact mit einer entsprechenden Softwarevariante angegeben werden. Dies ist zum Beispiel der Fall, wenn eine Softwarekomponente auf ein Arduino-Modul installiert werden soll. Ist die angeschlossene Proxy-Node ein Rechner mit dem Betriebssystem Windows, so muss ein anderes Program zum installieren der Softwarekomponente auf dem Arduino genutzt werden als wenn auf dem Rechner ein auf Linux basierendes Betriebssystem installiert ist.

Automatisches Selbstupdate

Manche Hardwarekomponenten bieten durch eingebaute oder vorinstallierte Softwarekomponenten wie einem BIOS oder Netzwerk-Bootloader die Fähigkeit an, nach dem Einschalten selbstständig oder nach entsprechender Konfiguration in Netzwerken von einem anderen Rechner Softwarekomponenten anzufragen, zu laden und auszuführen. Ein Typisches Beispiel hierfür ist das Laden und Ausführen eines Betriebssystems über ein Ethernet-Netzwerk nach dem PXE-Standard [Hen99]. Um eine solche Fähigkeit zum Deployment von Softwarekomponenten zu nutzen, muss ähnlich wie bei der im letzten Abschnitt genannten Vorgehensweise eine weitere Softwarekomponente in dieses Netzwerk eingebracht werden, welche auf entsprechende Anfragen reagieren und die entsprechenden Daten zur Verfügung stellen kann. Dafür kann temporär ein zusätzliches System aus Hardware- und Softwarekomponenten in das bereitzustellende System eingebracht und verwendet werden. Alternativ kann temporär ein schon existierender Teil des Systems dafür zweckentfremdet werden, um auf ihm entsprechende Softwarekomponenten und Daten auszuführen und abzulegen. In beiden Fällen ist das weitere Vorgehen daraufhin ähnlich dem bei Deployment via Proxy beschriebenen.

Deployment als Physical Task

Parallel zu Deployment von Hardwarekomponenten ist auch das Deployment von Softwarekomponenten auf ihnen über einen Physical Task mit entsprechendem Werkzeug realisierbar. Hierbei ist es möglich, dass lediglich ein Datenträger mit einem Programm zur Installation mit der entsprechenden Hardwarekomponente verbunden werden muss und eine schon installierte Softwarekomponente, wie ein Bootloader oder Betriebssystem, den Installationsprozess unterstützt. Eine solcher Physical Task stellt eine geringe Komplexität dar und ist ähnlich der in [KG14] gezeigten Tasks darstellbar. Ebenso ist es allerdings möglich, dass ein Mensch oder Roboter ein System wie einen portablen Rechner mit Betriebssystem und weiteren Programmen sowie der zu installierenden Softwarekomponente als Werkzeug nutzt und diesen mit der Hardwarenode verbindet. Daraufhin kann der Mensch oder Roboter mittels dieses Werkzeugs die Softwarekomponente auf der Hardwarekomponente installieren. Ein solcher Vorgang ließe sich jedoch auch automatisieren, indem das zugehörige Managementsystem selbst auf diesen portablen Rechner zugreifen kann und mittels den in Abschnitt 6.2.3 genannten Techniken selbst alle weiteren Schritte durchführen kann. In diesem Fall ist lediglich das Positionieren und Verbinden des portablen Rechners mit der Hardwarekomponente ein Physical Task. Ist dies erfolgt, so kann das Deploymentsystem weitere Schritte übernehmen.

6.2.4 Einschränkungen

Bei manchen Hardwarekomponenten, insbesondere bei Mikrocontrollern, besteht nur die Möglichkeit eine einzige Softwarekomponente auf ihnen bereitzustellen. Dies ist teilweise aufgrund der beschränkten Fähigkeiten der vorhandenen Entwicklungswerkzeuge und Softwareframeworks der Fall. Jedoch ist es auch teilweise mangels Rechenleistung und Speicherplatz nicht praktikabel, eine Art Betriebssystem oder einen Interpreter zum Hosten weiterer Softwarekomponenten auf diesen Hardwarekomponenten bereitzustellen. Aus diesem Grund kann auf manchen Hardwarekomponenten nur eine einzige Softwarekomponente installiert werden. Rekonfiguration einer solchen Softwarekomponente ist, sofern ihr Zugriff auf einen persistenten Speicher ermöglicht wird, mit etwas Entwicklungsaufwand meist realisierbar. Da die Möglichkeit damit nicht immer gegeben ist, mehrere Softwarekomponenten auf einer Hardwarekomponente bereitzustellen, verhindert dies die Bereitstellung mehrerer Softwarekomponenten auf einer entsprechenden Hardwarekomponente. In diesen Fällen muss diese Einschränkung hingenommen werden oder alternativ ein entsprechendes Framework wie µKevoree [FMF+12] für die Entwicklung der entsprechenden Softwarekomponenten sowie für ihr Deployment und ihre Rekonfiguration im Betrieb genutzt werden.

6.2.5 Management

Alle Möglichkeiten zum Deployment von Hardwarekomponenten sind auch Möglichkeiten zum Management dieser Komponenten. Sämtliche Möglichkeiten zum Deployment von Softwarekomponenten auf Hardwarekomponenten können ebenfalls zum Management dieser Softwarekomponenten genutzt werden. Zusätzlich können unter Umständen nun installierte Softwarekomponenten Management- und Updatefunktionalitäten über Kommunikationskanäle anbieten oder für das Ausführen von weiterer Software durch diese Funktionalität genutzt werden.

6.3 Reduktion von Komplexität durch Modellierungssichten

Wie leicht in Abbildung 6.1 zu erkennen ist, ist eine Darstellung aller Nodes und Relationships bei jedem annähernd komplexen cyber-physischen System extrem unübersichtlich. Daher wurden im Rahmen dieser Arbeit zur Verbesserung der Nutzbarkeit von OpenTOSCA dem Topologieeditor von Winery einige Funktionen hinzugefügt, welche auf den Prinzipien von Abstrahierung und Information Hiding basieren. Diese sowie ihre Hintergründe werden im Folgenden erklärt.

6.3.1 Selektives Ausblenden von Nodes und Relationships

Bei der Entwicklung von cyber-physischen Systemen arbeiten oftmals Experten verschiedener Bereiche an demselben System, beispielsweise Softwareexperten, Elektrotechnikexperten und Experten aus dem Bereich Maschinenbau [VDI04]. Jedoch haben diese häufig wenig Wissen über die Bereiche der anderen Experten und müssen für ihre Aufgaben auch oftmals kein Wissen darüber haben. Daher können beispielsweise zur besseren Übersicht bei der Ansicht des Modells alle Nodes ausgeblendet werden, deren NodeType nicht in den Bereich des entsprechenden Experten fällt. Daher ist es möglicherweise sinnvoll, zur selektiven Anzeige von Hardwarekomponenten in TOSCA weitere vom Node Type *HardwareNode* abgeleitete Wurzel-Node Types wie *ElectronicNode*

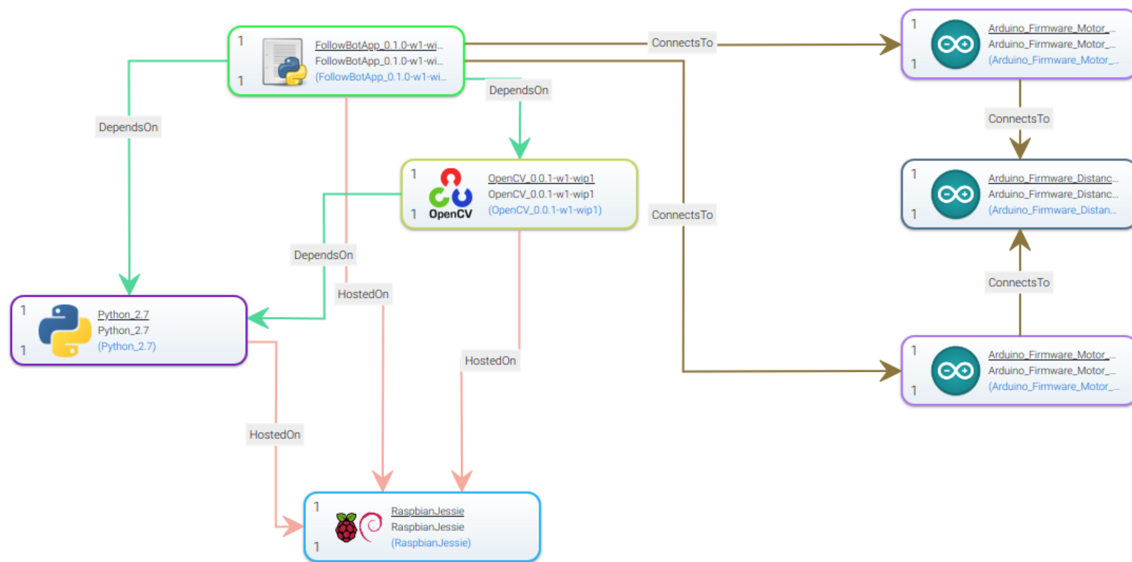
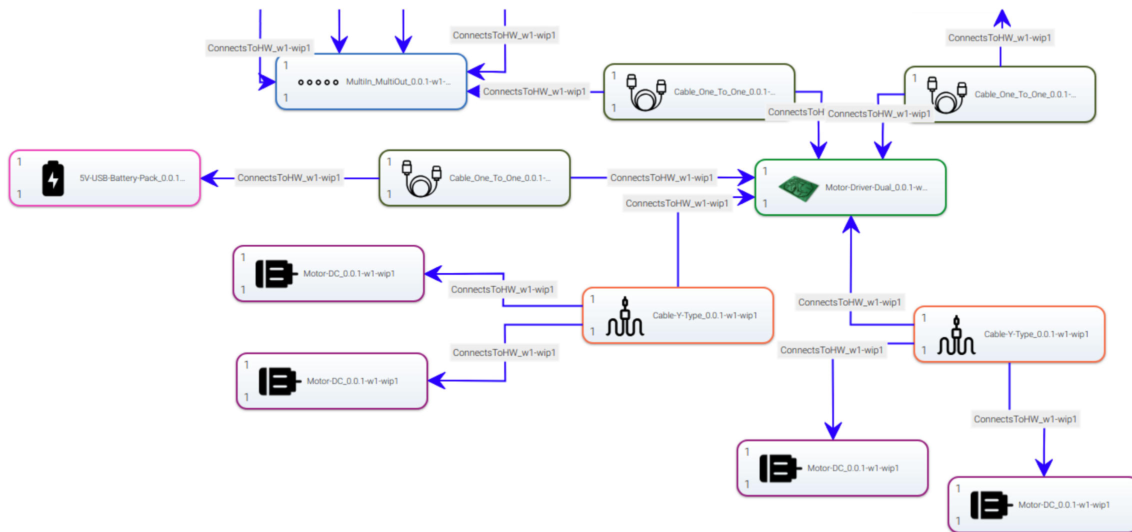


Abbildung 6.6: Ansicht des Modells des autonomen Fahrzeugs mit ausgeblendeten HardwareNodes.

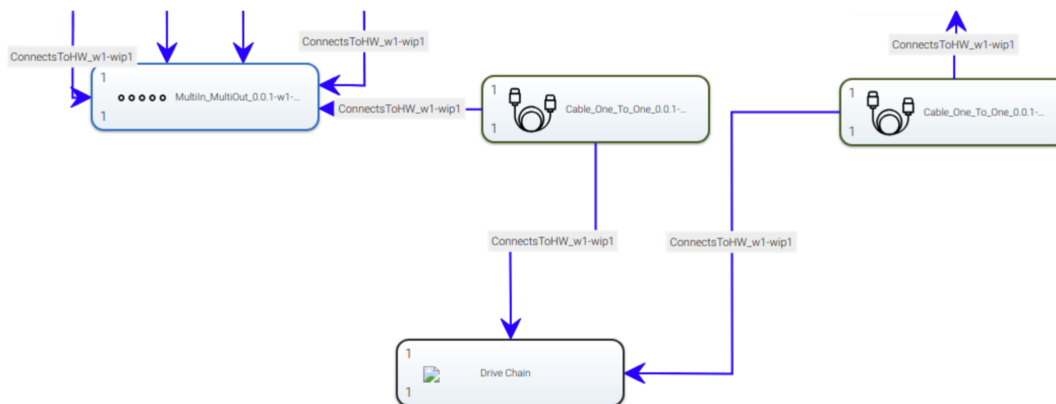
und *MechanicalNode* für entsprechende Spezialisierungsbereiche hinzuzufügen. Dies hat auch den Nebeneffekt, dass bei Deployment- und teils auch Managementaktionen bei einem solchen NodeType sofort klar ist, dass ein Physical Task zur Durchführung nötig ist. In die Topologieansicht von OpenTOSCA wurde daher die Möglichkeit eingebaut, Knoten nach ihrem Root-Nodetype HardwareNode auszublenden oder alternativ alle Nodes, welche nicht von diesem NodeType sind, auszublenden. Zusätzlich wurde die Möglichkeit implementiert, Gruppen an beliebigen Nodes zu erstellen, welche gezielt ausgeblendet werden können. Dadurch besteht die Möglichkeit, in der Ansicht sämtliche für die aktuelle Aufgabe irrelevanten Nodes sowie deren Relationships auszublenden und nur bestimmte Nodes und deren Relationships anzuzeigen.

6.3.2 Substituieren von Gruppen von Nodes

Werden wie im letzten Abschnitt genannt gezielt Komponenten und deren Relationen ausgeblendet, so wird die Darstellung des Systems zwar überschaubarer, aber die Präsenz dieser Komponenten kann dabei übersehen werden. Ebenfalls kann die Gesamtfunktion einer Gruppe an Komponenten nicht so leicht erkannt werden, wenn sämtliche dieser Komponenten sowie ihre Relationen untereinander angezeigt werden. Um dieses Problem zu lösen, kann das aus Schaltplänen bekannte Prinzip genutzt werden, um mehrere Komponenten und deren Relationen in der Anzeige durch eine neue, diese Gesamtfunktion bezeichnende Komponente zu ersetzen. Durch eine solche Substitution können weiterhin sämtliche Komponenten und Relationen im Modell dargestellt werden, während dennoch in einer grafischen Anzeige durch diese Abstraktion die Komplexität des angezeigten Systems für Nutzer verringert werden kann. Daher wurde zusätzlich zum Ausblenden von im vorhergehenden Absatz genannten Gruppen von Nodes auch eine Substitution dieser durch eine neue Substitutionsnode implementiert. Dies zeigt sich beispielsweise an dem in Abbildung 6.7a nochmals im Detail gezeigten Antriebsstrang des als Fallstudie genutzten Roboterfahrzeugs. Dieser besteht aus einer Batterie, einer Treiberschaltung, vier Motoren und einigen Kabeln. Für einen Softwareentwickler ist der Aufbau dieses Antriebsstranges irrelevant, es ist lediglich wichtig,



(a) Ausschnitt aus Abbildung 6.1, Fokus auf die Komponenten des Antriebsstrangs



(b) Selbiger Ausschnitt wie in Abbildung 6.7a gezeigt, die Komponenten des Antriebsstrangs wurden durch eine Dummy-Komponente ersetzt.

Abbildung 6.7: Detail der in Abbildung 6.1 gezeigten Darstellung des als Fallstudie genutzten autonomen Fahrzeugs. Gezeigt sind unter anderem die Komponenten des Antriebsstrangs sowie die Darstellung mit substituierten Komponenten.

dass dieser existiert und über welche Verbindungen er angesprochen werden kann. Durch eine Substitution dieser Komponenten und Relationen ist für einen Softwareentwickler deren Funktion besser erkennbar und das System etwas übersichtlicher. Dies ist in Abbildung 6.7b zu sehen, hier wurden die Komponenten des Antriebsstrangs in der grafischen Darstellung durch eine Dummy-Komponente ersetzt. Substituiert wird dadurch lediglich ein kleiner Teil des Systems, die Substitution reduziert die Anzahl an angezeigten Nodes und Relationships nur jeweils um acht. Dennoch kann dies in Kombination mit weiteren Substitutionen oder Ausblenden von Nodes das Verständnis eines Systems deutlich unterstützen. Da TOSCA die Möglichkeit der Substitution eines Service Templates zu einem Node Type definiert, ist diese automatische Substitution eines Teils eines Service Templates möglicherweise nicht nur grafisch, sondern auch im Modell möglich.

6.3.3 Substituieren von Kabeln und Adaptern

Ebenfalls ist es für Entwickler oftmals nicht nötig, die genaue Art der Kommunikationsverbindungen auf Hardwareebene zwischen verschiedenen Nodes vom Typ HardwareNode zu kennen. Da eine Kommunikationsverbindung jedoch über beliebig viele Kabel und Adapter verlaufen kann, ist diese möglicherweise sehr schwer zu erkennen. Daher ist es für Entwickler hilfreich, diese Kabel und optional auch Adapter sowie deren Relationen in der Ansicht durch abstrakte Relationen zu ersetzen, welche die Möglichkeit einer Kommunikationsverbindung darstellen. Dadurch ist für Entwickler schnell erkennbar, welche Hardwarekomponenten direkt miteinander kommunizieren können. Gleiches gilt auch für die Verbindungen der Stromversorgung. So können Entwickler schnell erkennen, welche Geräte an welcher Stromversorgung angeschlossen sind und ob dies bei Komponenten noch fehlt. Daher wurde in die Topologieansicht von OpenTOSCA eine primitive Möglichkeit eingebaut, Kabel durch direkte Verbindungen zwischen den damit verbundenen Geräten visuell zu ersetzen. Hierbei wurde eine einfache Tiefensuche genutzt und die internen Verbindungen der Kabel ignoriert, diese Vorgehensweise kann deutlich verfeinert werden, um auch kabelinterne Verbindungen sowie Adapter zu berücksichtigen.

6.3.4 Kommunikationsnetzwerke

Analog zu den 1:1 Kommunikationsverbindungen auf Hardwareebene bietet es sich in einer grafischen Oberfläche für Entwickler auch an, Kommunikationsnetzwerke anstelle von Kabeln, Adaptern und Bridges anzuzeigen. Die Erkennung von Kommunikationsnetzwerken kann ähnlich der Erkennung von Kommunikationsmöglichkeiten auf Hardwareebene realisiert werden. Hierbei müssen jedoch auch Bridges-Relationships zu Bridges einbezogen werden. Zur grafischen Darstellung dieser Kommunikationsnetzwerke sind verschiedene Darstellungsformen denkbar. Dies wurde nicht im Rahmen dieser Arbeit in OpenTOSCA implementiert.

6.4 Lektionen

Während dem Erstellen und Evaluieren der in dieser Arbeit vorgeschlagenen Konzepte wurden einige Erkenntnisse zum Modellieren von cyber-physischen Systemen sowie zu deren Komponenten gesammelt. Wie in dieser Arbeit zu sehen war, können cyber-physische Systeme aus verschiedensten Komponenten mit verschiedenen Eigenschaften und Eigenarten bestehen. Wird aus diesen Komponenten ein Gesamtsystem erstellt, so ergeben sich dabei verschiedene Problemquellen. Im Folgenden sollen diese Erkenntnisse zusammengefasst und vorgestellt werden.

6.4.1 Stromversorgung und automatisches Booten

Die Stromversorgung einer Hardwarekomponente ist immer eines ihrer grundlegenden Requirements. Sobald sie an die entsprechende Hardwarekomponente angeschlossen ist, wird diese jedoch häufig sofort aktiv und versucht ein Betriebssystem oder eine Firmware auszuführen. Ist kein Bootloader oder anderweitig eine initiale Software ab Werk vorhanden, so verhält sich die Hardwarekomponente passiv und erkennt auch keine zu einem späteren Zeitpunkt durch Anschluss eines Speichermediums hinzugefügte Software. Auf Befehle zur Installation oder zum Ausführen einer solchen Software

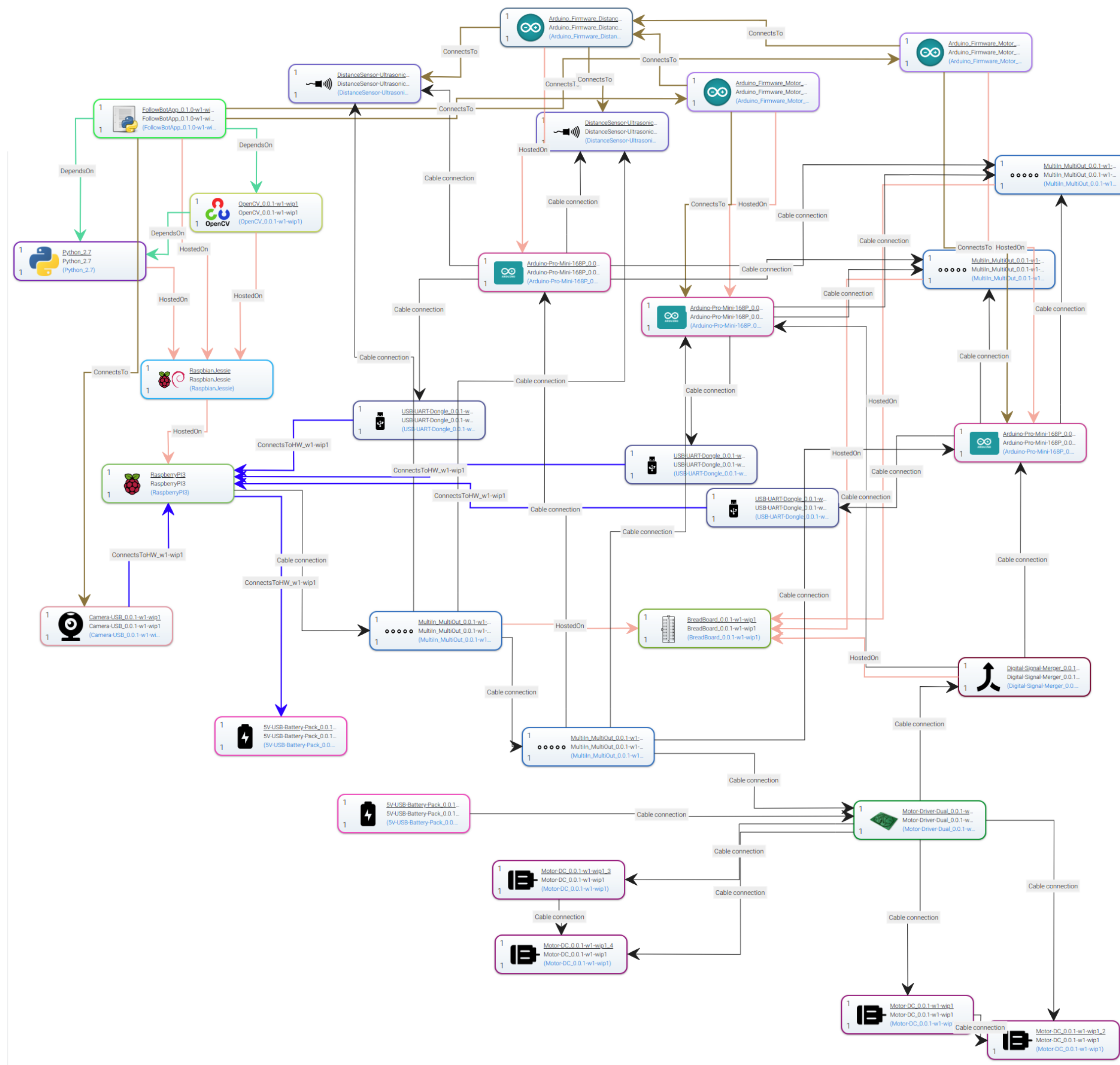


Abbildung 6.8: Ansicht des Modells des autonomen Fahrzeugs mit abstrahierten Kabelverbindungen.

reagiert die Komponente in einem solchen Zustand ebenfalls nicht. In einem solchen Fall muss die Hardwarekomponente neu gestartet werden. Existiert dazu kein Reset-Anschluss, Reset-Schalter oder Hauptschalter, so muss als Workaround bei initialer Betriebssystem- beziehungsweise Firmware-Installation unter Umständen zum Geräteneustart ein Physical Task angegeben werden, welcher durch kurzzeitiges Trennen der Verbindung der Stromzufuhr einen Neustart hervorruft. Dies entspricht leider nicht den Modellierungsprinzipien von TOSCA, ist jedoch kein verbreitetes Merkmal von Hardwarekomponenten. Lediglich Geräte der Reihe RaspberryPi sind bei der Recherche zu dieser Arbeit mit solchen Eigenschaften aufgefallen, diese Funktion kann jedoch nachgerüstet werden.

6.4.2 USB-Geräte

Werden mehrere USB-Geräte (aktive USB-Hardwarekomponenten) vom selben Typ mit einer Computingkomponente verbunden, so ist meist unklar, welche dieser angeschlossenen Hardwarekomponenten zu welchem intern von einem Treiber zur Verfügung gestellten Port gehört. So werden beispielsweise den seriellen Schnittstellen von USB zu Seriell Adaptern bei Linux eine ID vom Typ *USBtty[Zahl]* und unter Windows vom Typ *COM[Zahl]* zugeordnet. Dabei wird diese fortlaufende Zahl in der Reihenfolge der Erkennung dieser Anschlüsse zugeordnet und beinhaltet keinerlei Information über den physikalischen Anschluss, kann also keiner seriellen Schnittstelle zugeordnet werden. Je nach Implementierung des Treibers werden diese Informationen auch gespeichert um beim nächsten Anschluss des Adapters anhand seiner Hardware-ID dieselbe ID wieder zu vergeben. Werden nun mehrere solche Adapter an eine Computingkomponente angeschlossen, so erkennt der Treiber die serielle Schnittstelle des zuerst angeschlossenen Adapters als *USBtty0* bzw. *COM0*, die des zweiten als *USBtty1* bzw. *COM1* und so fort. In diesem Fall besteht nur die Möglichkeit, das Betriebssystem den Gerätepfad aller USB-Geräte anzeigen zu lassen und über den Vergleich der Geräte-IDs, sowie möglicherweise vom Treiber bereitgestellte Informationen, auf den USB-Anschluss zu schließen, an welchem der Adapter angeschlossen ist. Verliert ein solcher Treiber jedoch die Verbindung zu einem Adapter, so kann es unter Umständen sein, dass der von dem Adapter zu Verfügung gestellten Schnittstelle eine neue ID zugeordnet wird. Teilweise wird hierbei auch eine nicht mehr vergebene ID wieder verwendet. Wird unter Linux noch durch ein Programm auf die Schnittstelle mit der bisherigen ID zugegriffen, so bleibt diese ID bis zum Ende der Verwendung bestehen, jedoch werden keine Daten an den Adapter gesendet, da dessen Schnittstelle nun eine neue ID zugewiesen wurde. Dies stellt nicht zu unterschätzende Hürden und Aufwand bei der Modellierung und Softwareentwicklung dar. Der Treiber im Betriebssystem Raspbian für die in der Fallstudie dieser Arbeit verwendeten Adapter von USB zu Seriell vergab zuverlässig dieselben IDs an die Adapter, sofern diese schon beim Bootvorgang angeschlossen waren und kein Ausfall durch Spannungsschwankungen auftrat.

Werden keine zwei gleichen Gerätetypen an einer Hardwarekomponente mit Computingfähigkeiten angeschlossen, kann schlicht nach verfügbaren Schnittstellen gesucht werden und die gefundene verwendet werden. Dies vereinfacht die Darstellung und reduziert bei entsprechender Implementierung der verwendeten Softwarekomponenten die Fehleranfälligkeit beim Deployment. Diese Umsetzung ermöglicht jedoch Fehlkonfigurationen, sobald aus Versehen mehrere Komponenten dieses Typs in einem dargestellten System verwendet werden.

6.4.3 Aufwand

Die ausführliche Modellierung von gängigen Hardwarekomponenten als NodeTypes in TOSCA ist auch mit Unterstützung durch Werkzeuge wie OpenTOSCA sehr aufwändig. Nutzt man diese NodeTypes zur Modellierung eines Systems, so zeigt sich jedoch die Stärke von TOSCA durch das baukastenartige System sowie die grafische Darstellung der Topology Templates als Graphen, welches zusammen mit gezieltem Anzeigen und Ausblenden von Nodes und Relationships sowie dem Generieren von Informationen das Verständnis eines solchen Systems deutlich fördern kann. Auch automatische Auflösung von Verbindungen zwischen HardwareNodes sowie Verifikation dieser aufgrund von Requirements und Capabilities dürfte weiter dazu beitragen, dass der Aufwand beim Modellieren von Hardware mittels TOSCA geringer wird, da hierdurch bestimmte Fehler

automatisch erkannt werden können. Die Modellierung von Hardware mittels TOSCA ist bei manueller Definition der Verbindungen (Relationships) auf Hardwareebene etwas komplexer als die Modellierung von Softwarekomponenten in einer Cloud-Umgebung, welche über das Internet miteinander kommunizieren.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurden Erweiterungen für den TOSCA-Standard vorgestellt, mit welchen auch die Komponenten von cyber-physischen Systemen sowie deren Relationen untereinander und zu Softwarekomponenten dargestellt werden können. Dadurch ist nun die Möglichkeit gegeben, in TOSCA die elektronischen Komponenten sowie die Softwarekomponenten von cyber-physischen Systemen sowie IoT-Systemen darzustellen. Ebenso ist hierdurch zumindest begrenzt eine Darstellung rein mechanischer Komponenten in TOSCA möglich. Diese Möglichkeit der Darstellung von Hardwarekomponenten sowie deren Relationen untereinander und zu Softwarekomponenten wurde anhand einer Fallstudie, eines autonomen Fahrzeugs, überprüft. Dafür wurden verschiedene Ausschnitte des Modells der Fallstudie ausgewählt und ihre Modellierung präsentiert. Ebenfalls wurden einige Möglichkeiten für das Deployment von cyber-physischen Systemen thematisiert sowie für die Generierung von Informationen, welche dabei benötigt werden oder unterstützen können. Weiterhin wurden die Lektionen zum Modellieren von cyber-physischen Systemen sowie zu dessen besonderen Herausforderungen in dieser Arbeit vorgestellt. Da die Modellierung eines cyber-physischen Systems wie gezeigt sehr komplex ist, wurden einige Möglichkeiten für die Unterstützung von Modellierern dieser Systeme aufgezeigt.

Diese Arbeit behandelt lediglich einen der vielen Aspekte bei der Modellierung von cyber-physischen Systemen. Wie weit sich der hier genannte Ansatz zur Darstellung von Hardwarekomponenten in TOSCA auch zur Darstellung von mechanischen Hardwarekomponenten, insbesondere zur Kraftübertragung, erweitern oder übertragen lässt, ist noch zu erörtern. Dies könnte sich insbesondere für die Modellierung von Fertigungsanlagen relevant erweisen. Ebenfalls wurden einige Möglichkeiten zum automatischen Erkennen von Kommunikations- und Stromversorgungsnetzwerken angerissen, welche noch detailliert ausgearbeitet werden können. Auch ist eine automatische Vervollständigung von Kabelverbindungen auf Basis der zwischen Softwarekomponenten modellierten Kommunikationsverbindungen eine mögliche Erweiterung für die in dieser Arbeit genannten Konzepte. Werden die Kommunikationsmöglichkeiten zwischen Hardwarekomponenten jedoch manuell vorgegeben, so ist es nützlich, wenn auf der Basis dieser Darstellung automatisch überprüft werden kann, ob die angegebene Kommunikation zwischen Softwarekomponenten auch durch die Verbindungen zwischen Hardwarekomponenten ermöglicht wird. Dafür ist zu erörtern, wie dies Anhand der in dieser Arbeit vorgestellten Darstellung von Hardwarekomponenten und deren Relationen umgesetzt werden kann. Auch fällt die Darstellung von Softwarekomponenten, welche Daten zwischen Hardwareports weiterleiten, sogenannte Bridges, in dieses Themengebiet. Bridges sind bei manchen Systemen essentiell zur Erkennung von Kommunikationsnetzwerken, da ohne sie lediglich Bussysteme und einzelne Verbindungen zwischen zwei Computingkomponenten erkannt werden können. Auf diese Erkennung von Kommunikationsnetzwerken kann auch die Erkennung von Möglichkeiten zur Verwaltung von Softwarekomponenten über diese Netzwerke aufbauen. Hierauf folgend stellt sich außerdem die Frage, wie dies durch Ausführen von Softwareartefakten auf Hard- oder Softwarekomponenten, welche mit dem Zielknoten verbunden sind, wie in Abschnitt 6.2.3 angesprochen möglich ist. Ebenfalls stellt sich die Frage, wie diese Möglichkeit in

Modellen erkannt und beim Deployment automatisch ausgeführt werden kann. Da in dieser Arbeit häufig Physical Tasks angesprochen wurden, unter anderem für das Bereitstellen und Verbinden von Hardwarekomponenten, ist zu erörtern, wie diese praktisch dargestellt werden können. Dies kann möglicherweise durch eine entsprechende Erweiterung der von BPEL4People [KKL+05] eingeführten Human-Tasks erreicht werden. Beispielsweise müssen Hardwarekomponenten beim Aufbau eines cyber-physischen Systems bereitgestellt werden. Daraus folgt die Frage, wie dies genau dargestellt werden kann und muss. Zu überprüfen ist außerdem, bis zu welchem Detailgrad eine Darstellung von Hardware in TOSCA möglich und notwendig ist und welche weiteren Möglichkeiten sich durch eine detailliertere Darstellung eines Systems ergeben. Beispielsweise könnte selbst ein Kleinstcomputer wie ein RaspberryPi, wie in Abschnitt 5.5.2 gezeigt, nicht nur als einzelne Komponente, sondern auch als Hauptplatine und Speicherkarte dargestellt werden. Desktop- oder Serverrechner könnten sogar weitaus detaillierter dargestellt werden, da bei deren Deployment auch Komponenten wie Prozessor, Arbeitsspeicher, Laufwerke, Erweiterungskarten und Kühlelemente als einzelne Komponenten verbaut werden müssen.

Literaturverzeichnis

[AI15] *NEW PRODUCT – ESP8266 SMT Module – ESP-12*. Adafruit Industries. Mai 2015. URL: https://blog.adafruit.com/2015/05/06/new-product-esp8266-smt-module-esp-12/ (zitiert auf S. 26).

[AIM10] L. Atzori, A. Iera, G. Morabito. „The Internet of Things: A Survey“. In: *Comput. Netw.* 54.15 (Okt. 2010), S. 2787–2805. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010. URL: http://dx.doi.org/10.1016/j.comnet.2010.05.010 (zitiert auf S. 22).

[ARD15a] *Arduino Language Reference*. Arduino. 2015. URL: https://www.arduino.cc/reference/en/ (zitiert auf S. 43).

[ARD15b] *Arduino Software (IDE)*. Arduino. 2015. URL: https://www.arduino.cc/en/guide/environment (zitiert auf S. 69).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Vinothek-A Self-Service Portal for TOSCA.“ In: *ZEUS*. Citeseer. 2014, S. 69–72 (zitiert auf S. 16).

[BDG+14] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, S. Thiele, W. Schäfer, M. Meyer, U. Pohlmann, C. Priesterjahn, M. Tichy. „The MechatronicUML design method-process and language for platform-independent modeling“. In: *Technical Repeport tr-ri-14* 337 (2014) (zitiert auf S. 30).

[Blu02] L. Bluma. „Das Blockdiagramm und die “Systemingenieure”. Eine Visualisierungspraxis zwischen Wissenschaft und Öffentlichkeit in der US-amerikanischen Nachkriegszeit“. In: *NTM International Journal of History & Ethics of Natural Sciences, Technology & Medicine* 10.4 (2002), S. 247 (zitiert auf S. 26).

[DIN97] *DIN EN 60617: Graphische Symbole für Schaltpläne*. 1997 (zitiert auf S. 25).

[Edw13] C. Edwards. „Not-so-humble raspberry pi gets big ideas“. In: *Engineering & Technology* 8.3 (2013), S. 30–33 (zitiert auf S. 19).

[ESP18] *ESP8266EX Datasheet*. V6.0. Espressif Systems. 2018. URL: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (zitiert auf S. 29, 69).

[ESPA18] *ESP8266 Arduino Core Github Pages*. ESP8266 Community Forum. 2018. URL: http://esp8266.github.io/Arduino/versions/2.3.0/doc/boards.html (zitiert auf S. 27).

[ESTA18] *American National Standard ANSI E1.11 – 2008 (R2018) Entertainment Technology—USITT DMX512-A Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories*. ESTA. Mai 2018 (zitiert auf S. 39, 44).

[FMF+12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, J.-M. Jezequel. „A Dynamic Component Model for Cyber Physical Systems“. In: *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*. CBSE ’12. Bertinoro, Italy: ACM, 2012, S. 135–144. ISBN: 978-1-4503-1345-2. DOI: 10.1145/2304736.2304759. URL: http://doi.acm.org/10.1145/2304736.2304759 (zitiert auf S. 71).

[Fra17] B. Frankston. „Whither Consumer Electronics [Bits Versus Electrons]“. In: *IEEE Consumer Electronics Magazine* 6.4 (Okt. 2017), S. 130–132. ISSN: 2162-2248. DOI: 10.1109/MCE.2017.2714270 (zitiert auf S. 18).

[GBMP13] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. „Internet of Things (IoT): A vision, architectural elements, and future directions“. In: *Future generation computer systems* 29.7 (2013), S. 1645–1660 (zitiert auf S. 22).

[Handson17] *ESP8266 NodeMCU WiFi Devkit User Manual*. V1.2. Handson Technology. März 2017. URL: http://www.handsontec.com/pdf_learn/esp8266-V10.pdf (zitiert auf S. 28).

[Hen99] M. Henry M. and. Johnston. *Preboot Execution Environment (PXE) Specification*. Version 2.1. Intel Corporation. Sep. 1999. URL: http://www.pix.net/software/pxeboot/archive/pxespec.pdf (zitiert auf S. 70).

[ICK+10] D. Ings, L. Clement, D. König, V. Mehta, R. Mueller, R. Rangaswamy, M. Rowley, I. Trickovic. „Web Services—Human Task (WS-HumanTask) Specification, Version 1.1“. In: *OASIS Committee Specification (August 2010)* (2010) (zitiert auf S. 66).

[IEEE93] „Graphic Symbols for Electrical and Electronics Diagrams Including Reference Designation Letters“. In: *IEEE Std. 315-1975* (1993) (zitiert auf S. 25).

[ITEAD16] *ESP-12E: ESP8266 Serial Port WIFI Wireless Transceiver Module for Arduino*. ITEAD. 2016. URL: www.itead.cc/esp-12.html (zitiert auf S. 27).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery–a modeling tool for TOSCA-based cloud applications“. In: *International Conference on Service-Oriented Computing*. Springer. 2013, S. 700–704 (zitiert auf S. 16).

[KG14] T. Kuroda, A. Gokhale. „Model-based Automation for Hardware Provisioning in IT Infrastructure“. In: *Proceedings of Systems Conference (StoscasCon), 2014 IEEE International*. 2014 (zitiert auf S. 23, 25, 70).

[KKL+05] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic. „Ws-bpel extension for people–bpel4people“. In: *Joint white paper, IBM and SAP* 183 (2005), S. 184 (zitiert auf S. 68, 80).

[Lep17] H. Leppinen. „Current use of linux in spacecraft flight software“. In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017), S. 4–13 (zitiert auf S. 19).

[MC13] *Datenblatt AVR ATmega8 Microcontroller*. Rev. 2486AA- 02/2013. Microchip, former ATMEL. Feb. 2013. URL: https://www.microchip.com/wwwproducts/en/ATmega8 (zitiert auf S. 29).

[MG+11] P. Mell, T. Grance et al. *The NIST definition of cloud computing*. 2011 (zitiert auf S. 13).

[Mil06] R. Milner. „Pure bigraphs: Structure and dynamics“. In: *Information and computation* 204.1 (2006), S. 60–122 (zitiert auf S. 23, 24).

[MQM11] C. A. Macana, N. Quijano, E. Mojica-Nava. „A survey on Cyber Physical Energy Systems and their applications on smart grids“. In: *2011 IEEE PES conference on innovative smart grid technologies latin america (ISGT LA)*. Okt. 2011, S. 1–7. DOI: 10.1109/ISGT-LA.2011.6083194 (zitiert auf S. 22).

[Mue06] F. Mueller. „Challenges for cyber-physical systems: Security, timing analysis and soft error protection“. In: *High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS) Workshop, Alexandria, Virginia*. 2006, S. 4 (zitiert auf S. 19).

[OASIS13a] *Topology and Orchestration Specification for Cloud Applications*. Version 1.0. OASIS. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 11, 13, 15).

[OASIS13b] *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer*. Version 1.0. OASIS. Jan. 2013 (zitiert auf S. 13).

[OT18] *OpenTOSCA Research Prototype Website*. 2018. URL: http://www.opentosca.org/ (zitiert auf S. 16).

[PRS16] D. Palma, M. Rutkowski, T. Spatzier. „TOSCA Simple Profile in YAML Version 1.0“. In: *OASIS Committee Specification* (2016). URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0-csprd01.html (zitiert auf S. 13).

[RLSS10] R. Rajkumar, I. Lee, L. Sha, J. Stankovic. „Cyber-physical systems: The next computing revolution“. In: *Design Automation Conference*. Juni 2010, S. 731–736. DOI: 10.1145/1837274.1837461 (zitiert auf S. 22).

[SBK+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. „OpenTOSCA for IoT: automating the deployment of IoT applications based on the mosquitto message broker“. In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM. 2016, S. 181–182 (zitiert auf S. 16).

[SF14] *Arduino Pro Mini 328 - 5V/16MHz Graphical Datasheet*. Sparkfun. 2014. URL: https://www.sparkfun.com/products/11113 (zitiert auf S. 21).

[VDI04] *Richtlinie VDI 2206 Ëntwicklungsmethodik für mechatronische Systeme"*. Verein Deutscher Ingenieure, Juni 2004 (zitiert auf S. 71).

[WBX14] C. Wang, Z. Bi, L. D. Xu. „IoT and Cloud Computing in Automation of Assembly Modeling Systems“. In: *IEEE Transactions on Industrial Informatics* 10.2 (Mai 2014), S. 1426–1434. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2300346 (zitiert auf S. 22).

Alle URLs wurden zuletzt am 20. 02. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift