Universität Bremen

FACHBEREICH 3 - INFORMATIK UND MATHEMATIK

MASTERARBEIT

STUDIENGANG INFORMATIK

Modellgetriebene Codegenerierung aus SysML-Modellen für das BeagleBone Black

Christian Manal
manal@uni-bremen.de
Matrikelnummer 3005022

Gutachter

Prof. Dr. Jan Peleska peleska@uni-bremen.de
Dr. Hui Shi shi@uni-bremen.de

19. März 2020

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, 19. März 2020

Christian Manal

Zusammenfassung

Die grafischen Modellierungssprachen Unified Modeling Language (UML) und System Modeling Language (SysML) sind weit verbreitete Standards für die Visualisierung von Software- und Hardwarekomponenten während des Entwicklungsprozesses. Wenn man an dieser Stelle etwas mehr Aufwand investiert und den Grundlagen der durch die Object Management Group (OMG) spezifizierten Model-Driven Architecture (MDA) [2] verfolgt, kann man später bei der Implementierung sehr viel Aufwand einsparen, indem man automatisch Code aus den so entstandenen Modellen generiert. Im Rahmen dieser Arbeit soll eine Software entwickelt werden, die SysML-Modelle in ausführbaren Code für eingebettete Systeme übersetzt und insbesondere auf Hardwareplattform BeagleBone Black (BBB) lauffähig ist.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation und Zielsetzung 1
1.2. Vorgehensweise 2
1.3. Verwandte Arbeiten und Abgrenzung 2

2. Grundlagen **4**
2.1. Die Systems Modeling Language (SysML) 4
2.2. XML Metadata Interchange (XMI) 7
2.3. Abhängigkeiten 8

3. Generierung von C-Code aus SysML-Modellen **10**
3.1. Modell-Konventionen 10
3.2. Abbildung von SysML-Elementen auf C-Code 11
3.3. Abbildung von Schnittstellen 23
3.4. Synchronisation 28
3.5. Artefakte der Codegenerierung 28

4. Verifikation **31**
4.1. SysML-Modell einer Blinkersteuerung 31
4.2. Die Testumgebung 35
4.3. Testfälle 38
4.4. Auswertung des Testergebnisse 42

5. Fazit und Ausblick **47**

Anhänge **48**

A. Abbildungsverzeichnis **49**

B. Tabellenverzeichnis **51**

C. Glossar **52**

D. Abkürzungen und Akronyme **53**

E. Literaturverzeichnis **55**

1. Einleitung

In diesem Kapitel werden zunächst die Motivation und die Zielsetzung dieser Arbeit zusammengefasst und die Methodik erläutert. Abschließend werden verwandte Arbeiten diskutiert und wie diese von dieser Arbeit abgegrenzt werden.

1.1. Motivation und Zielsetzung

Die UML und vermehrt auch die SysML sind häufig genutzte Standards zur Modellierung von Systemen, die insbesondere häufig für die Spezifikation von Soft- und Hardware und im Falle von SysML auch für die Rückverfolgung von Anforderungen genutzt werden.

Wenn man diese Modelle, die häufig sowieso schon erstellt werden, nach bestimmten Konventionen erstellt, kann man aus diesen automatisch Programmcode erzeugen. Dies hat den offensichtlichen Vorteil, dass man eine deutliche Menge an Aufwand in der Implementierung einsparen kann, da nur noch entsprechende Schnittstellen zur Zielplattform von Hand programmiert werden müssen. Und falls sich später Anforderungen bzw. Spezifikationen ändern, muss nicht aufwändig von Hand implementierter Code angepasst werden, sondern man kann einfach auf Knopfdruck aus dem angepassten Modell aktuellen Code erzeugen.

Ein weiterer Vorteil von automatisch generiertem Code ist, dass man die Fehlerquelle Mensch beim Übertragen der Spezifikation in Programmcode zu großen Teilen ausschließen kann, sofern man dem Codegenerator vertraut.

Das Ziel dieser Arbeit ist es, einen Codegenerator zu implementieren, der SysML-Modelle in C-Code übersetzt. Die Zielplattform ist das BBB und insbesondere das minimale Echtzeitbetriebssystem, welches für das BBB im Rahmen der Bachelorarbeit [12] und der Lehrveranstaltung *Real-time Operating Systems Development* (03-ME-702.04) an der Universität Bremen entstanden ist.

Zu erwähnen sei auch, dass ein stark verwandtes Thema der Codegenerierung auch immer die Verifikation von Modellen ist. Dies ist ausdrücklich nicht Teil dieser Arbeit. Es wird angenommen, dass Modelle, die an den Codegenrator übergeben werden, vorweg verifiziert wurden.

1.2. Vorgehensweise

Um SysML-Modelle in C-Code umzusetzen muss zunächst eine Abbildung von SysML-Modellelementen und deren Semantiken auf C-Code definiert werden. Im nächsten Schritt muss eine Software entwickelt werden, die SysML-Modelle programmatisch parsieren kann und dann die Abbildungsregeln auf das Modell anwendet. Dies wird in Kapitel 3 behandelt.

Die SysML-Spezifikation sieht vor, dass Modelle maschinenlesbar in XML Metadata Interchange (XMI) dargestellt werden können [3, Kap. Annex E]. Daher bietet es sich an den Codegenerator mithilfe der Bibliothek Libxml2 [15] zu implementieren.

Für die Modellierung eines Test-Modells wird das Eclipse Plugin Papyrus [7] verwendet. Da dieses zu Beginn dieser Ausarbeitung SysML nur bis zur Version 1.4 unterstützt (inzwischen werden auch neuere Versionen unterstützt), wird der Codegenerator auch für diese Version entwickelt. An dieser Stelle sei auch zu erwähnen, dass Papyrus das **when**-Schlüsselwort für Change-Events an den Transitionen von Zustandsautomaten leider nicht anzeigt. Bei Abbildungen von Zustandsautomaten sind darum Beschriftungen, die wie eine Guard ohne eckige Klammern aussehen, als Change-Event zu interpretieren.

Um den implementierten Codegenerator zu testen, wird ein SysML-Modell für eine Blinkersteuerung von Automobilen genutzt, welches im Rahmen der Lehrveranstaltungen *Systeme hoher Sicherheit und Qualität* (03-MB-700.31) im Wintersemester 2018/2019 und *Spezifikation eingebetteter Systeme* (03-ME-702.03) im Sommersemester 2019 entwickelt wurde. Dies wird in Kapitel 4 behandelt.

1.3. Verwandte Arbeiten und Abgrenzung

Model-Based Systems Engineering (MBSE) und automatische Codegenerierung aus Modellen ist kein neuer Ansatz. Tatsächlich gibt es bereits kommerzielle Produkte, die insbesondere auch aus SysML-Modellen Code generieren können. Beispiele dafür sind *Rational Rhapsody* von IBM [14], oder auch der *Windchill Modeler* (ehemals *Integrity Modeler*) von der Firma PTC [16]. Die Produkte sind allerdings proprietär und für den Einsatz in der Industrie gedacht, sodass die unterliegenden Konzepte und Methoden generell nicht einsehbar sind. Dadurch sind sie eher schlecht für den Einsatz in einer akademischen Umgebung geeignet.

Andere Produkte, wie das Eclipse Plugin Papyrus, welches im Rahmen dieser Arbeit zum Erstellen von SysML-Modellen genutzt wird, unterstützt Codegenerierung für Java und C++, aber nicht C und insbesondere auch nicht für Echtzeitsysteme.

Im akademischen Umfeld gibt es Arbeiten, die sich mit UML/SysML-Codegenerierung befassen, aber aufgrund der Komplexität der Modellierungssprachen häufig nur mit einem bestimmten Aspekt, wie beispielsweise Activity-Diagrammen von AskariHemmat in [5], oder auch nur das Generieren eines Code-Gerüsts, ohne tatsächliche Funktionalität von Hellwege in [11].

Andere Arbeiten, wie die von Harel und Gery für die Codegenerierung aus UML-Zustandsautomaten in Java [10], befassen sich mit der Codegenerierung für objektorientierte Programmiersprachen. Da die Programmiersprache, für die in dieser Ausarbeitung Code generiert werden soll, C ist und somit nicht objektorientiert, kann auf diese Ansätze hier nicht zurückgegriffen werden.

Anders als die genannten Arbeiten, versucht diese Ausarbeitung einen vertikalen Schnitt (*vertical slice*) der Materie umzusetzen. Es soll also nicht ein Element der SysML bis ins letzte Detail unterstützt werden, sondern „ein Bisschen von allem“, sodass vollständige Modelle behandelt werden könne, solange diese nur die Teilmenge der UML und SysML verwenden, die unterstützt werden.

2. Grundlagen

Dieses Kapitel befasst sich mit den theoretischen Grundlagen von SysML und wie diese in maschinenlesbarer Form dargestellt werden kann, sowie vorangehenden Arbeiten, die als Basis für die Implementierung herangezogen wurden.

2.1. Die Systems Modeling Language (SysML)

Die SysML [3] ist eine Erweiterung der weit verbreiteten UML [6]. Die UML ist primär für die Modellierung von Software-Systemen gedacht und bietet sich eher schlecht für das Abbilden von Hardware an. Unter anderem ergänzt die SysML dies durch das Einführen zusätzlicher Modellelemente.

Beispielsweise sieht die SysML als zentrales Modellelement nicht Klassen, sondern sogenannte Blöcke vor. Blöcke können vielseitig angewendet werden, beispielsweise zum Definieren von komplexen Datentypen, oder zur Repräsentation von Hardware- oder Software-Komponenten eines Systems, oder auch als „virtuelle“ Komponente, die in ihre realen Einzelteile dekomponiert wird.

Die (De-)Komposition von Blöcken wird durch das ebenfalls neu eingeführte Block-Definition-Diagram (BDG) dargestellt. Ein simples Beispiel für das BDG eines Systems, welches aus zwei Hardware-Komponenten besteht, auf denen jeweils ein Software-Prozess läuft, ist in Abbildung 2.1 zu sehen. Ebenfalls durch die SysML neu definiert, ist das Internal-Block-Diagram (IBD), welches beschreibt, welche Schnittstellen die internen Komponenten eine Blocks haben und wie diese miteinander und ggf. mit der Außenwelt verbunden sind.

Diese Schnittstellen und Verbindungen werden durch Proxy-, Full- oder allgemeine Ports und Item-Flows modelliert, welche auf Ports und Information-Flows aus der UML aufbauen. Angenommen das in Abbildung 2.1 dekomponierte System hat eine Schnittstelle nach Außen, über die ein Integer-Wert an das System übergeben werden kann, welcher intern in der ersten Hardware-Komponente verarbeitet und ein Ergebnis als String an die zweite Hardware-Komponente weitergegeben wird, dann könnte dies wie im in Abbildung 2.2 dargestellten IBD modelliert werden.

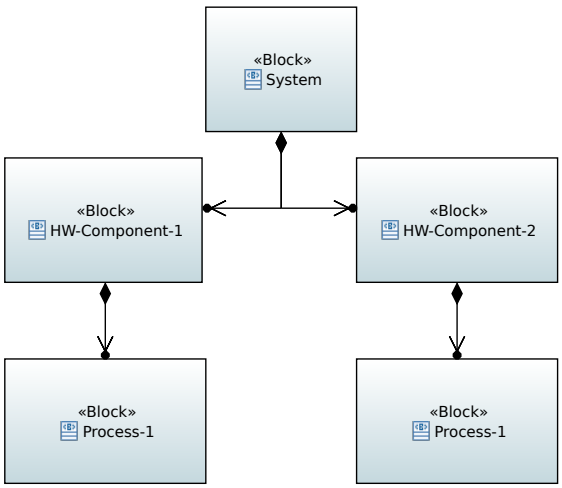

Abbildung 2.1.: Beispiel für ein SysML Block Defintion Diagram

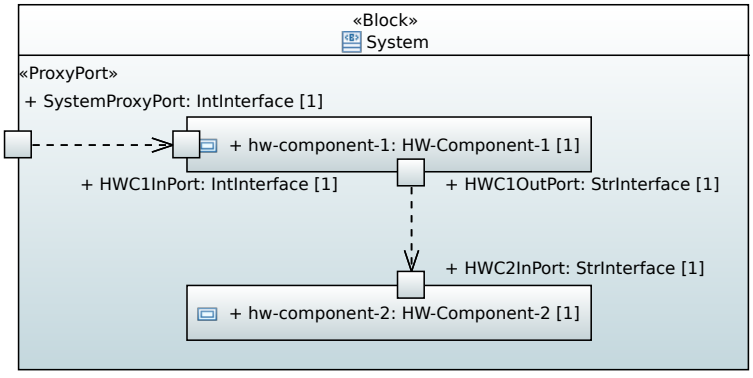

Abbildung 2.2.: Beispiel für ein SysML Internal Block Diagram

Insbesondere für die Verbindung mit der Außenwelt definiert die SysML das sogenannte Kontext-Diagramm, welches unter anderem Schnittstellen mit anderen Systemen, oder auch physikalische bzw. Umwelteinflüsse auf das System in das Modell einbeziehen.

Die zentralen Elemente zum Modellieren von Verhalten in der UML, die von der SysML größtenteils übernommen werden, sind Zustandsautomaten [6, Kap. 14] [3, Kap. 13] und Activities [6, Kap. 15] [3, Kap. 11]. Um den Aufwand in Grenzen zu halten, wird im Rahmen dieser Ausarbeitung der Fokus auf Zustandsautomaten gelegt.

Die Ereignisse (*Events*), die die Transitionen von UML-/SysML-Zustandsautomaten feuern lassen, werden dabei nach dem sogenannten Run-to-completion (RTC)-Paradigma [6, Abschn. 14.2.3.9] abgearbeitet. Das bedeutet, dass ein Zustandsautomat jeweils genau ein Event zur Zeit behandelt und neue Events erst verarbeitet werden, wenn der Zustandsautomat sich in einem stabilen Zustand (*stable state*) befindet.

Ein Manko der UML, welches die SysML behebt, ist, dass sie keinen direkten bzw. einheitlichen Weg anbietet, um Anforderungen zu definieren und zurückzuverfolgen, welche Modellelemente diese umsetzen. In der SysML sind Anforderungen als Modellement vorgesehen und können direkt mit den Modellelementen verknüpft werden, die diese umsetzen. Dies erlaubt unter anderem auch automatisiert festzustellen, ob alle Anforderungen umgesetzt wurden, indem man einfach prüft, ob es Anforderungen gibt, die mit keinem Modellelement verknüpft sind.

Problematisch ist sowohl bei der UML als auch der SysML, dass dessen Standards nicht im mathematischen Sinne formal definiert sind. Dies spiegelt sich insbesondere an den Stellen wieder, an denen die Spezifikation explizit Lücken in der Definition von Elementen lässt, die durch den Nutzer eigenständig gefüllt werden können. Ein Beispiel dafür ist beispielsweise in der Spezifikation von Transitionen von Zustandsautomaten, bei der explizit offen gelassen wird, nach welchen Kriterien entschieden wird, welche von mehreren aktiven Transition tatsächlich feuert [6, Abschn. 14.2.3.7].

Das hat zur Folge, dass das selbe Modell auf unterschiedliche Weisen interpretiert werden kann, je nachdem, wie der Betrachter diese Lücken füllt. Es gibt Ansätze zur Formalisierung der UML, wie beispielsweise von Evans et al. [8], welche allerdings schwer in den Standard zu integrieren sind, ohne Abwärtskompatibilität zu existierenden Werkzeugen und Methoden zu kompromittieren. Dementsprechend waren die Anstrengungen zur Formalisierung der UML bisher vergeblich.

Daraus folgt wiederum, dass Werkzeuge, wie auch ein Codegenerator, die Lücken und Ungenauigkeiten in den Spezifikationen konkret füllen müssen, um konsistente Ergebnisse liefern zu können. Diese domänenspezifischen Zusätze zu den Standards

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI ...>
   <uml:Model xmi:id="..." name="ExampleModel">
      <packagedElement xmi:type="uml:Package" xmi:id="..."
       name="ExamplePackage">
         <packagedElement xmi:type="uml:Class" xmi:id="8nINO4bYv9"
          name="ExampleBlock"/>
      </packagedElement>
   </uml:Model>
   <Blocks:Block xmi:id="..." base_Class="8nINO4bYv9"/>
</xmi:XMI>
```

Abbildung 2.3.: XMI-Kodierung eines SysML-Blocks

müssen entsprechend festgehalten und als Anforderungen an die Modelle dokumentiert werden.

2.2. XML Metadata Interchange (XMI)

Eine Voraussetzung für das automatische Generieren von Programmcode aus Modellen ist, dass die Modelle in maschinenlesbare Form gebracht werden können. Die OMG spezifiziert den XMI-Standard [4], auf Basis von Extensible Markup Language (XML), als maschinenlesbare Darstellungsform für alle Modelle, deren Metamodell in Meta-Object Facility (MOF) [1] dargestellt werden können. Insbesondere können so UML- und SysML-Modelle in XMI abgebildet werden [6, Kap. Annex E] [3, Kap. Annex G] und erlauben so das Parsieren durch gängige XML-Bibliotheken.

Da die SysML als Erweiterung der UML spezifiziert ist, wird auch in deren XMI-Darstellungen auf bestehende Definitionen aufgebaut. Beispielsweise werden Blöcke in der XMI-Repräsentation eines SysML-Modells zunächst als Klasse kodiert und dann in einem separaten XML-Element als Block ausgewiesen, welches die Klasse über eine eindeutige ID referenziert (`xmi:id`). Ein vereinfachtes Beispiel dazu ist in Abbildung 2.3 zu sehen.

Dies gilt analog auch für andere UML-Elemente, die von der SysML erweitert oder modifiziert werden. Beispiele hierfür sind UML-Ports, die als Proxy- oder Full-Port ausgewiesen werden können, oder Properties eines Blocks, die so als Flow-Properties für die Typisierung von Ports genutzt werden. Diese Art der Kodierung von Modellelementen setzt somit voraus, dass beim Parsieren immer darauf geprüft werden muss, ob so ein zusätzliches XML-Element existiert, welches den Typ des Modellelements modifiziert.

Ein Problem bei der automatischen Generierung von Code ist das automatische Erzeugen von eindeutigen Namen für Variablen und dergleichen. Hier würde man zunächst denken, dass es sich anbietet, das `xmi:id`-Attribut zu verwenden. Dies bringt allerdings mehrere Problem mit sich. Zum einen wäre es wünschenswert, wenn der generierte Code menschenlesbar ist, sodass er von Menschen nachvollzogen werden kann. Kryptische Variablennamen machen dies enorm schwer. Zum anderen wird vom XMI-Standard nur vorgegeben, dass der Wert des Attributs im Dokument eindeutig ist, macht aber keine Vorgaben für das Format [4, Abschn. 7.6.1]. Dadurch kann es sein, dass der Wert des Attributs kein gültiger Bezeichner der jeweiligen Programmiersprache ist. Dies würde wiederum erfordern, dass `xmi:id`-Werte vom Codegenerator auf Gültigkeit geprüft und ggf. modifiziert werden müssen.

2.3. Abhängigkeiten

Der zu entwickelnde Codegenerator entsteht nicht in einem Vakuum, sondern baut auf verschiedenen Komponenten auf, die an anderer Stelle entstanden sind. Dieser Abschnitt beschreibt die wichtigsten Abhängigkeiten und deren Funktion.

Micro-Kernel-6

Die Software „Micro-Kernel-6“ ist ein minimales Echtzeitbetriebssystem, welches in der Lehrveranstaltung *Real-time Operating Systems Development* (03-ME-702.04) auf Basis der Ergebnisse aus [12] entwickelt wurde. Das Betriebssystem wurde speziell für das BBB entwickelt und stellt Mechanismen für das Scheduling und Inter-Process-Communication (IPC) zur Verfügung.

Der Scheduler des Micro-Kernel-6 ist ein sogenannter präemptiver Frame-basierter Scheduler, der jeder Anwendung eine feste Zeitscheibe, einen sogenannten Minor-Frame, an CPU-Zeit zur Verfügung stellt. Dies erlaubt es, mehrere Komponenten eines Modells parallel zu betreiben, ohne dass diese sich gegenseitig in negativer Weise beeinflussen, indem z.B. eine Komponente zu viel CPU-Zeit in Anspruch nimmt. Deshalb werden Minor-Frames in diesem Kontext auch Partition genannt.

Innerhalb der Partitionen läuft weiterhin je ein Prioritäten-basierter, kooperativer Scheduler, in dem Tasks ihrer Priorität nach aus Warteschlangen für beliebige Ereignisse aufgeweckt werden, nachdem sie sich selbstständig schlafen gelegt haben. Tasks könnten in diesem Falle beispielsweise Zustandsautomaten aus einem SysML-Modell sein.

Domänenrahmenwerk

Ein Domänenrahmenwerk (Engl. Domain Framework) ist prinzipiell eine Programmbibliothek, die domänenspezifische Schnittstellen bereitstellt, bzw. von diesen abstrahiert. In diesem speziellen Fall wird ein Domänenrahmenwerk, welches in der Lehrveranstaltung *Spezifikation eingebetteter Systeme* (03-ME-702.03) entstanden ist, als Basis genommen, um die Schnittstelle zwischen dem Micro-Kernel-6 und dem generierten Code zu entwickeln.

Das in der Lehrveranstaltung verwendete Rahmenwerk ist jedoch für Linux-basierte Systeme ausgelegt und muss deshalb auf das BBB und den Micro-Kernel-6 portiert werden. Einige Kernkomponenten, wie beispielsweise ein wiederverwendbarer Interpreter für Zustandsautomaten, können jedoch unverändert übernommen werden und ersparen somit einigen Implementierungsaufwand.

Das Domänenrahmenwerk soll, neben dem Interpreter für Zustandsautomaten, von der Hardware abstrahieren, sodass kein Hardware-spezifischer Code generiert werden muss und Schnittstellen für interne und externe Kommunikation bereitstellen. Wie die Abbildung vom Modell auf die jeweiligen Schnittstellen und die letztendliche Codegenerierung dafür passiert, wird in Kapitel 3 beschrieben.

Desweiteren soll ein Rahmenwerk für das Erstellen von Prozessen/Tasks auf dem Micro-Kernel-6 bereitgestellt werden, sodass auch hier kein (oder so wenig wie möglich) Plattform-spezifischer Code generiert werden muss, sondern nur allgemeine Datenstrukturen, die vom Domänenrahmenwerk auf die vom Micro-Kernel-6 bereitgestellten Schnittstellen abgebildet werden können.

Eine weitere wichtige Aufgabe des Domänenrahmenwerkes ist es, einen sogenannten *Dispatcher*-Task bereitzustellen. Dieser kümmert sich nach jedem RTC der Zustandsautomaten darum, dass „verbrauchte“ Events gelöscht, neu aufgetretene Events zur Verfügung gestellt und der nächste RTC angestoßen wird. Da die in der Lehrveranstaltung entstandene Version auf Linux ausgelegt ist, muss dieser Teil ebenfalls auf das BBB und den Micro-Kernel-6 portiert werden.

3. Generierung von C-Code aus SysML-Modellen

Dieses Kapitel behandelt die Abbildung von SysML auf C-Code und weiterhin wie diese so gestaltet werden kann, dass der Code plattformagnostisch generiert werden kann. Dies erfordert, dass vom Domänenrahmenwerk Schnittstellen, beispielsweise für Hardwareabstraktion, bereitgestellt werden, auf die aus dem Modell heraus zugegriffen werden kann, bzw. die separat an das Modell gekoppelt werden können, ohne diese direkt im Modell zu referenzieren.

3.1. Modell-Konventionen

Damit der Codegenerator richtig funktioniert, muss das Modell gewisse Konventionen einhalten. Es muss offensichtlich sichergestellt werden, dass nur unterstützte Modellelemente und -semantiken verwendet werden. Welche das sind, wird weiter unten erläutert.

Eine andere Konvention ist die Struktur des Modells. Der Codegenerator geht davon aus, dass es genau ein Package gibt, das Verhalten in Form von UML-Zustandsautomaten als Owned-Behavior enthält. Zustandsautomaten, die als Nested-Classifier modelliert sind, werden ignoriert. Dies ist das sogenannten *System-Package*. Das System-Package soll weiterhin genau einen Block enthalten, der das modellierte System als Ganzes repräsentiert. Der Block, der das System repräsentiert, kann nun weiter in seine realen Komponenten dekomponiert werden.

Die erste Ebene an Blöcken, in die das System dekomponiert wird, werden im Folgenden als *Deployable* bezeichnet. Ein Deployable besteht aus mindestens einem untergeordneten Block, welcher einen Zustandsautomaten als Owned-Behavior enthält, und kann dediziert auf einer eigenen Hardwareplattform oder parallel mit anderen Deployables auf einer gemeinsam genutzten Plattform laufen. Dies wird zum Teil vom Domänenrahmenwerk vorgegeben, wobei ggf. auch der Codegenerator leicht angepasst werden muss, um dies zu ändern.

Eine Weitere Anforderung an das Modell ergibt sich daraus, dass die Programmier-

sprache C nicht objektorientiert ist und es somit nicht trivial ist, Objekte, Variablen und dergleichen nur in ihrem jeweiligen Scope in der unterliegenden Objekthierarchie Verfügbar zu machen.

Hinzu kommt, dass innerhalb des Modells durch Opaque-Expression- und Opaque-Behavior-Elemente C-Code in das Modell eingebettet werden kann, in dem beispielsweise auf die durch einen Port bereitgestellten Daten zugegriffen werden kann. Wenn man hier dem Modell der Objektorientierung folgen würde, müssten hier jeweils lange, verschachtelte Objektnamen angegeben werden, um eindeutig auf ein Objekt und dessen Attribute zugreifen zu können.

Um dies so zu vereinfachen, dass beispielsweise auf Ports einfach durch ihre Namen, anstatt durch ihren vollständigen Pfad im Modell angesprochen werden können, müssen dessen Namen innerhalb eines Deployable einmalig sein. Dadurch können diese als globale Variable innerhalb des jeweiligen Deployable bereitgestellt werden, ohne dass es zu Namenskonflikten kommt, die in SysML eigentlich keine wären.

Bei Elementen, auf die nicht aus in das Modell eingebetteten C-Code zugegriffen werden sollte, gilt diese Konvention nicht. Die generierten Variablen werden aus dem Namen des Elements selbst, sowie allen Eltern-Elementen bis zur Wurzel (also dem Deployable) erzeugt und sind somit immer eindeutig.

Opaque-Expressions und Opaque-Behaviors müssen immer für die Sprache „C“ spezifiziert werden. Spezifikationen für andere Sprachen werden ignoriert. Opaque-Expressions müssen gültige C-Ausdrücke sein. Opaque-Behaviors müssen genereller gültiger C-Code sein.

Weitere Anforderungen an das Modell ergeben sich aus den unterstützten Modellelementen, welche im nächsten Abschnitt erläutert werden.

3.2. Abbildung von SysML-Elementen auf C-Code

Da die UML- und SysML-Standards sehr umfangreich und komplex sind, kann nur eine Teilmenge der spezifizierten Modellelemente- und Semantiken im Rahmen dieser Arbeit umgesetzt werden. Im Folgenden werden die implementierten Elemente und Semantiken beschrieben und wie diese auf C-Code abgebildet werden.

Typ-Definitionen

UML/SysML bietet mehrere Möglichkeiten, Typen zu definieren, die beispielsweise Ports zugewiesen werden können. Dabei kann es sich um primitive Datentypen wie

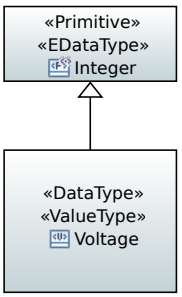

Abbildung 3.1.: Beispiel für einen ValueType-Block

```
1 typedef int Voltage;
```

Abbildung 3.2.: Generierter Code zu Abbildung 3.1

Integer oder Boolean handeln, oder auch um Blöcke, die ein oder mehrere Attribute enthalten (und so auch beliebig tief geschachtelt werden können).

Primitive Datentypen werden einfach wie das nächste C-Äquivalent behandelt. Integer und Boolean werden so zu `int` und Real wird zu `double` (da der Micro-Kernel-6 derzeit beim Kontextwechsel die Gleitkommaregister nicht sichert und wiederherstellt, sollten Gleitkommazahlen zunächst nicht benutzt werden, da dies zu unerwartetem Verhalten führen kann). Analog dazu werden ValueType-Blöcke, die von einem primitiven Datentypen abgeleitet sind, als `typedef` in C umgesetzt. Ein ValueType-Block, der beispielsweise einen Typ für elektrische Spannung in Volt als Integer definieren soll, würde wie in Abbildung 3.1 auf den primitiven Datentyp Integer generalisieren und in den in Abbildung 3.2 zu sehenden C-Code umgesetzt werden.

Enumeration-Blöcke werden in C jeweils zu einem `enum`. Um dabei eventuelle Probleme mit der Reihenfolge von Elementen und dergleichen zu vermeiden, werden die Werte jedem Element explizit zugewiesen. In der Umsetzung ist dabei zu beachten, dass in SysML ein Enumeration-Element ohne explizite Zuweisung implizit den Wert 0 hat. Außerdem muss darauf geachtet werden, dass die Elemente von Enumerationen eindeutige Namen haben.

Komplexe Datentypen können modelliert werden, indem ein Block mit Attributen versehen wird. Dabei kann es sich um einfache Blöcke mit regulären Attributen handeln, oder auch um sogenannte Interface-Blöcke, die beispielsweise zum Typisieren von Proxy-Ports verwendet werden. Insbesondere bei der Typisierung von Ports werden außerdem sogenannte Flow-Properties anstatt regulärer Attribute verwendet.

Die aus diesen verschiedenen Arten von Blöcken modellierbaren Datentypen werden auf C-Strukturen (`struct`) abgebildet. Beispielsweise würde der in Abbildung 3.3 dargestellte Block zum in Abbildung 3.4 zu sehenden C-Code werden.

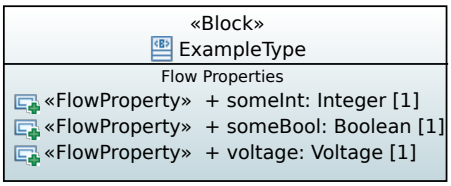

Abbildung 3.3.: Beispiel für Block mit Flow-Properties

```
1 struct ExampleType {
2     int someInt;
3     int someBool;
4     Voltage voltage;
5 };
6 typedef struct ExampleType ExampleType_t;
```

Abbildung 3.4.: Generierter Code zu Abbildung 3.3

Zustandsautomaten

SysML-Zustandsautomaten [3, Kap. 13] werden nicht direkt in ausführbaren Code umgewandelt. Stattdessen werden sie in C-Strukturen übersetzt, die von einem wiederverwendbaren Interpreter ausgeführt werden können, der Teil des Domänenrahmenwerks ist. Die Details dazu und insbesondere auch wie Events kodiert und ausgewertet werden, werden im nächsten Abschnitt behandelt.

Guards müssen als Constraint angegeben werden, die jeweils durch eine Opaque-Expression spezifiziert ist. Actions müssen als Opaque-Behavior definiert werden.

Derzeit werden nur einfache Zustände (*simple States*) unterstützt, also keine verschachtelten Zustandsautomaten, History-Zustände, oder Parallelisierung. Dies führt zu komplexeren Modellen, vereinfacht aber die Implementierung des Interpreters und die Umsetzung des weiter unten beschriebenen Modells zur Abarbeitung von Ereignissen. Desweiteren werden Pseudozustände in Form des Startzustands, sowie Choice- und Junction-Zustände unterstützt.

Ein Zustandsautomat wird jeweils von einer `smSpec_t`-Struktur repräsentiert, dargestellt in Abbildung 3.5, welche den Namen des Automaten, ein Array aller Zustände des Automaten, sowie die Länge des Zustandsarrays enthält. Per Konvention ist der erste Zustand im Array der Startzustand.

Zustände werden in der in Abbildung 3.6 dargestellten Struktur `smState_t` kodiert. Diese enthält neben dem Namen des Zustandes folgende Elemente:

- Ein Flag, welches den Zustand als einfachen bzw. Pseudozustand ausweist.

```
1 typedef struct smSpec {
2     char *stateMachineName;
3     unsigned int numStates;
4     smState_t **sm;
5 } smSpec_t;
```

Abbildung 3.5.: `smSpec_t`-Struktur für Zustandsautomateninterpreter

```
1 typedef struct smState {
2     char *stateName;
3     unsigned int pseudoState;
4     unsigned int stateEnteredAt;
5     void (*entryAction)(void);
6     struct list_head *transListHead;
7 } smState_t;
```

Abbildung 3.6.: `smState_t`-Struktur für Zustandsautomateninterpreter

- Den Zeitpunkt des letzten Betretens des Zustandes, gemäß der vom Domänenrahmenwerk vorgegebenen Zeitmessung. In diesem Fall Millisekunden seit Systemstart.
- Einen Zeiger auf eine C-Funktion, welche die Entry-Action des Zustandes kodiert.
- Das Ankerelement einer doppelt verketteten Liste, welche alle ausgehenden Transitionen des Zustandes enthält.

Exit-Actions werden derzeit nicht unterstützt, sind aber analog zu Entry-Actions trivial zu implementieren. Gleiches gilt für DoActivity-Actions, wobei hier problematisch sein kann, dass gemäß der Spezifikation [6, Abschn. 14.2.3.4.3] DoActivities nebenläufig zu allem anderen Verhalten laufen und potentiell dauerhaft aktiv sind, bis sie beim Verlassen des jeweiligen Zustandes terminiert werden. Bei einer korrekten Umsetzung des Standards müsste für DoActivity-Actions also jeweils ein eigener Thread erzeugt werden. Um dies zu vereinfachen, könnte per Konvention vorgegeben werden, dass DoActivity-Actions immer von sich aus in vorhersagbarer Zeit terminieren müssen.

Derzeit auch nicht unterstützt sind Completion-Events, die ausgelöst werden, sobald die Entry- und DoActivity-Action des jeweiligen Zustands terminiert haben.

Die letzte C-Struktur, die benötigt wird um Transitionen zu kodieren, heißt `trans_t` und ist in Abbildung 3.7 zu sehen.

Die Struktur enthält folgende Elemente:

- Einen Zeiger auf eine C-Funktion, welche die Dauer eines Time-Events zurückgibt.
- Ein Flag, das aussagt, ob die **after**-Zeitdauer bereits abgelaufen ist.
- Einen Zeiger auf eine C-Funktion, die ein Change-Event kodiert.
- Einen Zeiger auf eine C-Funktion, die eine Guard kodiert.
- Einen Zeiger auf eine C-Funktion, die eine Aktion kodiert.
- Den Index des Zielzustandes in der Liste aller Zustände in `smSpec_t`.
- Ein Listen-Element, das auf die nächste und vorherige Transition des jeweiligen Zustands zeigt.
- Ein Zeiger auf die Struktur eines Ports, auf dem ein Signal eintreffen kann, welches ein Signal-Event auslöst.

Zeit-Events werden nur relativ zum Betreten des Zustandes unterstützt (**after**-Schlüsselwort). Wenn kein Zeit-Event für die Transition definiert ist, wird eine vom Domänenrahmenwerk vorgegebene Funktion `noDuration()` verwendet, welche immer 0 zurückgibt. Analog dazu gibt es die Funktion `guardTrue()`, falls keine Guard definiert ist und `emptyAction()` für nicht definiert Actions. `guardTrue()` gibt immer 1 zurück, `emptyAction()` ist einfach eine leere Funktion. Wenn kein Change- oder Signal-Event für eine Transition definiert ist, so müssen die jeweiligen Zeiger auf `NULL` stehen.

Wenn die Transition zu einem Pseudozustand gehört, werden alle Elemente mit Ausnahme der Guard und der Aktion ignoriert. Bei Pseudozuständen ist zu beachten, dass der Interpreter so lange über alle ausgehenden Transitionen iteriert, bis der Zustand verlassen werden kann. Wenn das Modell erlaubt, dass keine der ausgehenden Transitionen eines Pseudozustands aktiv ist, weil beispielsweise die Guards nicht erschöpfend sind, dann führt dies zu einem Deadlock durch Endlosschleife im Interpreter.

Gemäß der Spezifikation in [6, Abschn. 14.2.3.7] wird vom Standard kein Algorithmus vorgegeben, der entscheidet, welche Transition feuert, wenn beispielsweise mehrere Guards eines Choice-Zustandes wahr sind. Der Interpreter für Zustandsautomaten nimmt hier einfach die erstbeste aktive Transition. An derselben Stelle wird empfohlen, dass genau eine ausgehende Transition mit dem Schlüsselwort *else* als Guard versehen werden kann, die sozusagen wahr wird, wenn alle anderen Guards falsch sind. Dies wird derzeit für Zustandsautomaten nicht unterstützt.

Events

UML/SysML-Events sind Ereignisse, die Verhalten auslösen können. Die UML-Spezifikation sieht dabei vor, dass Events in sogenannten Event-Pools gesammelt werden und dann jeweils einzeln aus dem Pool entfernt und abgearbeitet (*dispatched*) werden [6, Abschn. 13.3]. In welcher Reihenfolge die Events in einem

```
1  typedef struct trans {
2      unsigned int (*duration)(void);
3      unsigned int hasElapsed;
4      int (*when)(eventHandler_t e);
5      int (*guard)(void);
6      void (*action)(void);
7      unsigned int tgtState;
8      struct list_head h;
9      struct SignalPort *p;
10 } trans_t;
```

Abbildung 3.7.: `trans`-Struktur für Zustandsautomateninterpreter

Event-Pool abgearbeitet werden, lässt der Standard explizit offen, um Entwicklern Spielräume für die Umsetzung modell- oder domänenspezifischer Anforderungen zu lassen. Die einzige Ausnahme im Kontext von Zustandsautomaten sind Completion-Events, welche laut Standard Priorität vor allen anderen Events haben sollten [6, Abschn. 14.2.3.8.3].

Das Modell der Abarbeitung von Events durch Event-Pools bringt allerdings mehrere potentielle Probleme mit sich.

Im Kontext von Echtzeitsystemen kann es beispielsweise problematisch sein, ein Time-Event in einen Event-Pool einzureihen, aus dem es erst nach unbestimmter Zeit wieder entfernt wird, da dies zum Verpassen von Echtzeit-Deadlines führen kann. Dementsprechend kann es sinnvoll sein, alle aufgetretenen Time-Events gleichzeitig zur Verfügung zu stellen, sodass aktive Zustandsautomaten parallel, so zeitnahe wie möglich, auf bereits eingetretene Time-Events reagieren können.

Weitere Problematiken ergeben sich aus der sequentiellen Bearbeitung von Change-Events. Peleska beschreibt in *Spezifikation eingebetteter Systeme* (03-ME-702.03) zwei Gründe, die dafür sprechen, dass Change-Events ebenfalls simultan bereitgestellt werden sollten:

- Angenommen es gibt zwei Bedingungen *cnd1* und *cnd2*, die jeweils ein Change-Event auslösen und *cnd1* impliziert *cnd2*. Dann gibt es keine „natürliche Ordnung“, nach der sinnvoll entschieden werden kann, ob das von *cnd1* oder von *cnd2* ausgelöste Change-Event zuerst bearbeitet werden soll.
- Angenommen *cnd1* und *cnd2* werden gleichzeitig wahr und lösen dadurch je ein Change-Event aus. Weiterhin angenommen das von *cnd1* ausgelöste Change-Event wird zuerst behandelt und nach dem RTC ist *cnd2* nun wieder falsch. Dann wäre das korrespondierende Change-Event trotzdem noch im Event-Pool, obwohl die auslösende Bedingung zum Zeitpunkt dessen Bearbeitung ggf. gar nicht mehr zutrifft.

Entsprechend dieser von Peleska geführten Argumentation werden alle Change-Events gleichzeitig zur Verfügung gestellt und jeweils am Ende eines RTC gelöscht, sodass alle Zustandsautomaten simultan auf die für sie relevanten Change-Events reagieren können, wodurch die oben beschriebenen Problematiken nicht mehr auftreten.

Events werden vom Zustandsautomateninterpreter in folgender Reihenfolge/Prioritätenordnung behandelt:

1. Time-Event
2. Change-Event
3. Signal-Event

Time-Events

In der Implementierung werden derzeit nur relative Time-Events unterstützt. Diese werden als C-Funktion kodiert, die entweder einen direkt im Modell definierten konstanten Integer-Wert zurück gibt, oder eine Opaque-Expression, in der z.B. Variablen genutzt werden können, die an anderer Stelle im Modell verändert werden. Die Kodierung als C-Funktion wurde gewählt, damit bei der Codegenerierung eine statische Zuweisung in der jeweiligen C-Struktur gemacht werden kann, was mit Opaque-Expressions, die auf Variablen zugreifen, nicht funktionieren würde.

Weiterhin werden keine variablen Zeiteinheiten unterstützt. Alle Zeitwerte werden als Millisekunden interpretiert.

Da Time-Events per Zustand verfolgt werden, ist es trivial diese zu invalidieren, falls sie keine Transition ausgelöst haben. Beim ersten RTC nach dem Eintreffen wird das Flag gesetzt, welches das Time-Event als eingetroffen markiert, unabhängig davon, ob die entsprechende Transition gefeuert hat oder nicht. Ist das Flag gesetzt, wird das Event beim nächsten RTC nicht mehr berücksichtigt.

Change-Events

Change-Events werden als C-Funktionen kodiert, die den Zustand einer Variable oder eines Ausdrucks in Form einer Opaque-Expression überwachen. Das Domänenrahmenwerk definiert hierfür ein Hilfsmakro (siehe Abbildung 3.8), welches aus einem eindeutigen Namen und einem booleschen Ausdruck eine C-Funktion erstellt.

In der statischen Variable `oldCndStatus` wird vermerkt, ob der überwachte Ausdruck zuletzt wahr oder falsch war. Wenn die Funktion mit dem Argument `peekEvent` aufgerufen wird, wird zunächst nur geprüft, ob sich der Zustand des Ausdrucks seit dem

```
1  typedef enum eventHandler { peekEvent, clearEvent } eventHandler_t;
2
3  #define CHANGE_EVENT(name,cond)                                    \
4  static int name(eventHandler_t e) {                                \
5      static int oldCndStatus = 0;                                   \
6      int newCndStatus = (cond);                                     \
7      if ( e == peekEvent ) {                                        \
8          return eventIndicator(oldCndStatus,newCndStatus);          \
9      }                                                              \
10     oldCndStatus = newCndStatus;                                   \
11     return -1;                                                     \
12 }
```

Abbildung 3.8.: C-Makro für Change-Events

letzten RTC von wahr auf falsch oder von falsch auf wahr geändert hat. Ist dies der Fall, ist das Change-Event eingetreten. Mit dem Argument `clearEvent` kann später `oldCndStatus` aktualisiert werden, sodass das Change-Event nicht erneut ausgelöst wird.

Da alle Change-Events gleichzeitig zur Verfügung gestellt werden, werden diese entsprechend alle gleichzeitig verarbeitet, auch wenn sie keine Transition feuern lassen. Dementsprechend werden nach jedem RTC alle Change-Events zurückgesetzt.

Signal-Events

Signal-Events werden durch das Verschicken von Signalen über Ports ausgelöst (siehe auch den Abschnitt zu *Ports und Item-Flows* weiter unten). Im Modell muss hier im Trigger der jeweiligen Transition neben dem Signal-Event, auf das gelauscht werden soll, auch definiert werden, auf welchem Port das zugehörige Signal eintreffen wird. Vom Domänenrahmenwerk wird vorgegeben, wieviele Signale maximal von den Ports aufgenommen werden können. Wenn schneller Signale erzeugt werden, als sie verarbeitet werden können, kann dies zu unerwartetem Verhalten führen.

Nach jedem RTC wird aus allen Signal-Ports das zuletzt bereitgestellte Signal aus jedem Signal-Port verworfen, unabhängig davon, ob es Transitionen hat feuern lassen oder nicht.

Ports und Item-Flows

Ports und Item-Flows (spezifiziert in [6, Kap. 14] und [3, Kap. 9]) können in SysML genutzt werden, um eine Vielzahl von Schnittstellen und Verhaltensweisen zwischen Komponenten zu modellieren. Im Rahmen dieser Ausarbeitung werden Ports und Item-Flows primär dazu genutzt, um zu beschreiben, wie verschiedene Komponenten Daten und Signale austauschen. Die Richtung von Informationsflüssen wird dabei ausschließlich von den Item-Flows abgelesen. Die Richtung von Flow-Properties wird ignoriert.

Dabei wird vom Codegenerator zwischen zwei Arten von Ports unterschieden. Denen, die durch einen Block, bzw. Interface-Block mit Flow-Properties getypt sind, und denen, die durch ein Signal getypt sind. Der Unterschied liegt darin, dass an Ports, die durch Blöcke getypt sind, die Daten kontinuierlich anliegen und so jederzeit darauf zugegriffen werden kann. Bei Ports, die durch Signale getypt sind, liegen die vom Signal übermittelten Daten nur genau zu dem Zeitpunkt an, an dem ein Signal am Port eintrifft.

Von Signalen getypte Ports werden im Folgenden als Signal-Ports bezeichnet und alle anderen einfach als Ports, bzw. „normale“ Ports.

Beide Arten von Ports haben gemeinsam, dass sich durch ihre Verbindung mit Item-Flows jeweils gerichtete Graphen bilden lassen. Um normale Ports in C umzusetzen, wird einfach für die Wurzel eine Variable vom Typ des Ports definiert und dann für alle Ports, einschließlich der Wurzel, damit diese sich gleich verhält, ein Zeiger auf diese Variable. Dadurch sind auf allen Ports innerhalb eines Graphen immer alle aktuellen Daten unter dem Namen der Ports zugreifbar.

Die in Abbildung 3.9 gezeigte Datenstruktur kodiert Signal-Ports, welche mehrere Signale in einem Ringpuffer vorhalten, bis diese verarbeitet wurden. Die Struktur enthält folgende Elemente:

- Den Namen des Signals.
- Die Größe in Bytes des jeweiligen Signals.
- Den Lese-Index des Ringpuffers.
- Den Schreib-Index des Ringpuffers.
- Bis zu `MAX_SIGS` Signale (dies ist ein vom Domänenrahmenwerk vorgegebenes Maximum).
- Die Anzahl an ausgehenden Item-Flows bzw. Ports, an die eingehende Signale weitergeleitet werden müssen.
- Ein Array von Signal-Ports, an die eingehende Signale weitergeleitet werden müssen (gemäß ausgehender Item-Flows des jeweiligen Ports im Modell).

Anders als bei normalen Ports, bekommen alle Signal-Ports ihre eigene Datenstruk-

```
1 struct SignalPort {
2     char *signalName;
3     unsigned int signalSize;
4     unsigned int rIdx;
5     unsigned int wIdx;
6     void *sigs[MAX_SIGS];
7     unsigned int numRecvs;
8     struct SignalPort **recvPorts;
9 };
```

Abbildung 3.9.: C-Struktur für Signal-Ports

tur, die jeweils den Namen des Ports trägt. Anhand der Item-Flows wird auch hier ein gerichteter Graph erzeugt, bei dem vom Wurzel-Port über das `recvPorts`-Array rekursiv alle empfangenden Ports erreicht werden können.

Da Papyrus zum Zeitpunkt dieser Ausarbeitung die in [3, Tab. 13.1] definierten *Receive-Signal-Action* und *Send-Signal-Action* in Zustandsautomatendiagrammen nicht unterstützt, musste insbesondere für das Versenden von Signalen aus Zustandsautomaten eine andere Lösung gefunden werden. Der Codegenerator erzeugt hierfür Hilfsfunktionen für jedes Signal, die beispielsweise in der Entry-Action eines Zustandes durch ein Opaque-Behavior aufgerufen werden können.

Die Funktion hat jeweils den Namen des Signals mit dem Präfix `send` (also beispielsweise `sendTestSignal` für ein Signal mit dem Namen „TestSignal“) und erwartet als Argumente einen Zeiger auf den Port, über den das Signal verschickt werden soll, sowie je ein Argument für jedes im Signal enthaltene Datum.

Da die Datenstruktur für Signal-Ports die Daten der zu verschickenden Signale als void-Zeiger erwartet, also keinen eigenen Speicher alloziert, wird außerdem in jeder dieser generierten Funktionen ein statisches Array erzeugt, welches eine Anzahl von `MAX_SIGS` Elementen mit der Größe des Signals enthält. Dieses wird als Ringpuffer behandelt. Da ein Signal-Port sowieso nur maximal `MAX_SIGS` Signale aufnehmen kann, muss bei diesem Ringpuffer nicht geprüft werden, ob freie Elemente vorhanden sind.

Ein Beispiel für eine generierte C-Funktion zum Versenden eines Signals, das einen Integer-Wert enthält, ist in Abbildung 3.10 zu sehen. Nach einem RTC wird jeweils geprüft, ob neue Signale verschickt wurden. Ist das der Fall, werden sie rekursiv auf alle empfangenden Ports verteilt.

```
1  static void sendTestSignal(struct SignalPort *p, int test) {
2      static TestSignal_t sig[MAX_SIGS];
3      static unsigned int next = 0;
4      unsigned int newIdx = (p->wIdx + 1) % MAX_SIGS;
5      if (newIdx == p->rIdx) {
6          return;
7      }
8
9      sig[next].test = test;
10     p->sigs[p->wIdx] = &sig[next];
11     p->wIdx = (p->wIdx + 1) % MAX_SIGS;
12     next = (next + 1) % MAX_SIGS;
13 }
```

Abbildung 3.10.: C-Funktion zum Versenden von Signalen über Signal-Ports

Operations und Activities

Neben Zustandsautomaten kann Verhalten in UML/SysML via Activity-Diagramm modelliert werden [6, Abschn. 15.2], [3, Kap. 11]. Das Hauptaugenmerk dieser Arbeit liegt auf Zustandsautomaten für die Modellierung von Verhalten, aber um das Spektrum von unterstützten UML/SysML-Elementen zu erweitern, wird auch eine Teilmenge von Activity-Diagrammen umgesetzt. Ziel soll es dabei sein, komplexe Bedingungen, z.B. für Guards, grafisch modellieren zu können.

Daher wird auch kein komplexer Algorithmus, wie der von Ouchani [13] vorgestellte New Activity Calculus (NuAC), angewendet, welcher Activies in ihrem vollen Umfang erfassen. Stattdessen werden Activies in Verbindung mit UML-Operations [6, Abschn. 9.6] umgesetzt, sodass Activity-Diagramme als C-Funktion mit modellierbaren Argumenten und Rückgabewerten aufgerufen werden können. Voraussetzung dafür ist, dass Activity-Diagramme sequentiell sind und garantiert terminieren. Activities zum Modellieren von Verhalten von Blöcken werden explizit nicht unterstützt.

Unterstützte Activity-Elemente sind:

- Activity-Parameter-Nodes
- Object-Flows
- Control-Flows
- Decision-Nodes
- Call-Behavior-Action-Nodes
- Output-Pins
- Input-Pins

Es muss für jede Activity eine Operation geben, welche die Activity als Methode refe-

renziert und die Activity-Parameter-Nodes der Activity als Owned-Parameter besitzt.

Activities müssen mindestens einen Activity-Parameter-Node für das Übergeben von Argumenten haben und genau einen Activity-Parameter-Node für die Rückgabe des Ergebnisses haben. Soll die Activity für die Modellierung einer komplexen Guard verwendet werden, sollte der Rückgabewert vom Typ Boolean sein.

Call-Behavior-Action-Nodes müssen ihr Verhalten als Opaque-Behavior definieren, indem beispielsweise übergebene Argumente aus Activity-Parameter-Nodes behandelt werden.

Decision-Nodes dürfen keine eigene Entscheidungslogik haben. Die Entscheidungslogik wird über Guards in ausgehenden Control-Flows definiert. Guards müssen als Opaque-Expression definiert werden. Das besondere Schlüsselwort `else` (ebenfalls als Opaque-Expression) definiert den Control-Flow, der genommen werden kann, wenn sonst keine Guard wahr ist.

Weiterhin muss es einen definitiven Einstiegspunkt geben, also einen Decision-Node, der eingehende Object-Flows aus Activity-Parameter-Nodes hat, aber keinen eingehenden Control-Flow.

Ein nach diesen Vorgaben modelliertes Activity-Diagramm wird nach folgendem Algorithmus in C-Code umgesetzt:

1. Erzeuge Prototypen für eine statische C-Funktion:

   - Der Name der Opeartion ist der Name der Funktion.
   - Der Typ des Activity-Parameter-Nodes für den Rückgabewert ist der Rückgabetyp. Ist keiner vorhanden, so ist der Rückgabetyp der Funktion `void`.
   - Name und Typ der Funktionsargumente ergeben sich aus den restlichen Activity-Parameter-Nodes.
   - Parameter mit der Richtung „out“ oder „inout“ werden als Zeiger definiert.

2. Ist der Rückgabetyp nicht `void`, erzeuge eine Variable mit Namen des Activity-Parameter-Nodes.

   - Hat der Activity-Parameter-Node einen Default-Wert, weise diesen zu, sonst ist die Variable uninitialisiert.

3. Markiere alle Input-Pins aller Call-Behavior-Action-Nodes und alle in Decision-Nodes eingehende Object-Flows mit `hasVarToken=false`.
4. Markiere alle Control-Flows mit `hasControlToken=false`.
5. Iteriere über alle Activity-Parameter-Nodes mit Richtung „in“ oder „inout“ und markiere Ziele der Object-Flows mit `hasVarToken=true`.

6. Finde Call-Behavior-Action-Node, von dem alle eingehenden Tokens `true` sind, falls vorhanden.
    - Markiere für eingehende Object- bzw. Control-Flows `hasVarToken=false` bzw. `hasControlToken=false`.
    - Erzeuge eine Variable für jeden Input-Pin und initialisiere diese mit dem Wert der Variable bzw. des Parameters des eingehenden Object-Flows.
    - Erzeuge eine Variable für jeden Output-Pin.
    - Füge das Opaque-Behavior des Call-Behavior-Action-Nodes ein.
    - Markiere alle ausgehenden Object-Flows bzw. dessen Ziele mit `hasVarToken=true`.
    - Wenn ein ausgehender Object-Flow auf den Rückgabe-Activity-Parameter-Node zeigt, weise den Wert des zugehörigen Output-Pins der Variable für den Rückgabewert zu und füge ein `return`-Statement ein.
    - Ansonsten verfolge alle ausgehenden Object-Flows und gehe zu 7. bzw. 6. falls beim Ziel alle eingehenden Tokens auf `true` stehen.
7. Finde Decision-Node für den alle eingehenden Tokens `true` sind, falls vorhanden.
    - Markiere für eingehende Object- bzw. Control-Flows `hasVarToken=false` bzw. `hasControlToken=false`.
    - Erzeuge `if`, `else if` oder `else` Anweisung anhand der aktuellen Guard.
    - Markiere ausgehende Control-Flows mit `hasControlToken=true`.
    - Verfolge ausgehende Control-Flows und gehe zu 6. bzw 7., falls beim Ziel alle eingehenden Tokens auf `true` stehen.
8. Gehe zu 6. bzw. 7., sofern es einen Decision- oder Call-Behavior-Action-Node gibt, bei dem alle eingehenden Tokens auf `true` stehen.
9. Füge `return`-Statement ein.

Dabei wird jeweils die Tiefe der Schachtelung vermerkt, sodass Einrückung korrekt generiert werden kann. So wird aus dem in Abbildung 3.11 zu sehenden Activity-Diagramm der in Abbildung 3.12 zu sehende C-Code. Die Zuweisungen `returnFalse = false;` und `returnFalse = false;` sind dabei die Opaque-Behviors, die im jeweiligen Call-Behavior-Action-Node definiert sind.

3.3. Abbildung von Schnittstellen

Die Schnittstellen eines SysML-Modells, in der Regel in Form von Ports, sind sehr abstrakt und somit hardware- und plattformunabhängig. Da diese Details auch generell nicht Teil eines SysML-Modells sind, muss bei der Generierung von Code die Verbindung zwischen abstrakten Ports und konkreten Hard- und Softwareschnittstellen auf andere Art und Weise hergestellt werden.

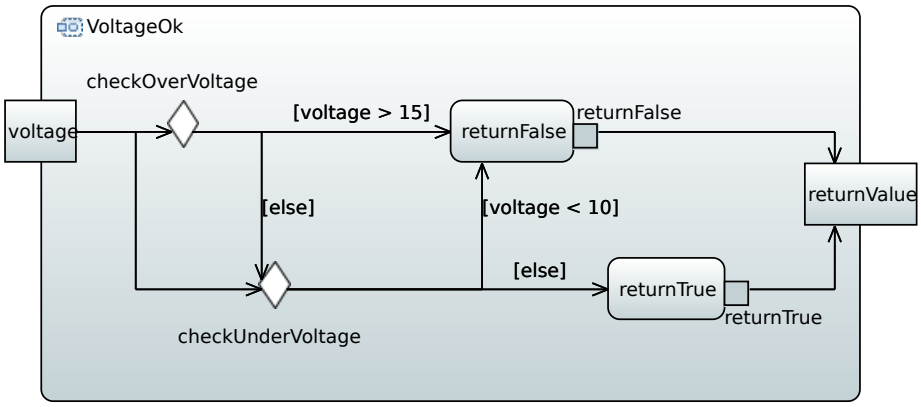

Abbildung 3.11.: Beispiel für ein Activity-Diagramm

```
1  static int VoltageOk(Voltage voltage) {
2      int returnValue = false;
3      if (voltage > 15) {
4          int returnFalse;
5          returnFalse = false;
6          returnValue = returnFalse;
7          return returnValue;
8      }
9      else {
10         if (voltage < 10) {
11             int returnFalse;
12             returnFalse = false;
13             returnValue = returnFalse;
14             return returnValue;
15         }
16         else {
17             int returnTrue;
18             returnTrue = true;
19             returnValue = returnTrue;
20             return returnValue;
21         }
22     }
23     return returnValue;
24 }
```

Abbildung 3.12.: Generierter Code zu Abbildung 3.11

```
1 typedef struct hal_interface_in {
2     unsigned int id;
3     size_t len;
4     int (*driver)(void *buffer);
5     struct list_head* h;
6 } hal_interface_in_t;
```

Abbildung 3.13.: Datenstruktur für Eingabeschnittstellen

```
1 typedef struct input_map_element {
2     unsigned int offset;
3     size_t len;
4     void *target;
5     struct list_head h;
6 } input_map_element_t;
```

Abbildung 3.14.: Datenstruktur für die Zuordnung von Variablen zu Schnittstellen

Dazu muss das Domänenrahmenwerk eine einheitliche Form von Schnittstelle bereitstellen, die von der unterliegenden Hard- bzw. Software abstrahiert. Je nach Richtung des Datenflusses werden C-Funktionen bereitgestellt, die Daten annehmen oder zurückgeben, die mit Signalen und Flow-Properties von Ports im Modell korrespondieren und nach jedem RTC aufgerufen werden.

Zu diesem Zwecke ist im Domänenrahmenwerk je eine Datenstruktur definiert, die für die Definition von Eingabe- und Ausgabeschnittstellen genutzt werden kann. Diese enthält eine eindeutige ID für die Schnittstelle, die hauptsächlich für Debugging-Zwecke relevant ist, die Länge in Bytes eines Datensatzes, die die Schnittstelle verarbeiten soll, einen Zeiger auf eine Treiber-Funktion, die die Daten versendet bzw. empfängt, sowie einen Listenkopf für eine doppelt verkettete Liste von Variablen, die von der Schnittstelle verarbeitet werden. In Abbildung 3.13 ist die Datenstruktur für Eingabeschnittstellen zu sehen. Die für Ausgabeschnittstellen sieht identisch aus. Die Treiberfunktionen erwarten bewusst einen void-Zeiger, sodass beliebige Daten versendet bzw. empfangen werden können.

Die Zuordnung von Variablen zu Schnittstellen erfolgt über die in Abbildung 3.14 zu sehende Datenstruktur. Diese unterscheidet sich für Ausgabeschnittstellen nur darin, dass der void-Zeiger `target` anstatt `source` heißt.

Die Datenstruktur enthält den Offset in Bytes der Variable innerhalb des Datenblocks, der an die Treiber-Funktion der Schnittstelle übergeben wird, die Länge in Bytes der Variable, einen Zeiger auf die korrespondierende Variable im System, sowie ein Listenelement, um die Struktur in in die Liste der Variablen in der jeweiligen Schnittstelle einhängen zu können.

Bei der Ziel- bzw. Quellvariable kann es sich generell um eine beliebige Variable bzw. Speicheradresse handeln. Generell ist es hier aber sinnvoll, entweder ganze Ports (normale oder Signal-Ports) oder einzelne Flow-Properties von Ports zu referenzieren.

Bei der Implementierung der Schnittstellentreiber muss auch beachtet werden, dass sich die Semantiken von normalen und Signal-Ports unterscheiden. Bei normalen Ports reicht es, den Wert in den Ports definierten Flow-Properties auf den jeweils aktuellsten Wert zu setzen, bzw. den aktuellen Wert über eine Schnittstelle zu verschicken. Bei Signal-Ports können mehrere Signale zur Zeit eintreffen, bzw. ausgehen, oder auch gar keine, sodass hier mit Warteschlangen gearbeitet werden muss.

In diesem speziellen Fall werden Signal-Ports, die auf eine Schnittstelle abgebildet werden, jeweils einer IPC-Nachrichtenwarteschlange im Micro-Kernel-6 zugeordnet. Am Ende jeden RTCs werden alle auf einem ausgehenden Signal-Port anliegenden Signale in die zugehörige IPC-Warteschlange übertragen und bei eingehenden Signal-Ports alle in der zugehörigen Warteschlange vorhandenen Signale in den Signal-Port.

Da der lesende Zugriff auf solche IPC-Nachrichtenwarteschlangen in der gegebenen Implementierung immer blockierend war (sprich: wenn keine Nachrichten in der Warteschlange sind, wird der jeweilige Task schlafen gelegt bis eine Nachricht entnommen werden kann), musste eine Schnittstelle für den nicht-blockierenden Zugriff hinzugefügt werden, sodass auf eingehende Signale geprüft werden kann, ohne das jeweilige Deployable auf unbestimmte Zeit zum Stillstand zu bringen.

Normale Ports sind hier einfacher zu handhaben. Da der Micro-Kernel-6 nicht mit virtuellem Speicher arbeitet, können die an den normalen Ports anliegenden Daten einfach zwischen den Datenstrukturen der Ports kopiert werden.

Was nun noch fehlt, ist ein Mechanismus, um Ports aus dem Modell auf die passenden Schnittstellen im Domänenrahmenwerk abzubilden. Da dies, wie gesagt, nicht Teil des eigentlichen SysML-Modells ist, muss dies separate passieren. Man könnte hier verlangen, dass diese Abbildung an irgendeiner Stelle von Hand programmiert wird, was aber entgegen des Ziels liegt, so viel Code wie möglich automatisch zu generieren.

Stattdessen wird die Abbildung in Form von zwei Comma-seperated values (CSV)-Dateien modelliert. In einer davon werden die Schnittstellen definiert und einem Treiber im Domänenrahmenwerk zugeordnet, in der zweiten werden Variablen den Schnittstellen zugeordnet. Um die Erstellung dieser CSV-Dateien zu vereinfachen, bietet es sich an, eine Software für Tabellenkalkulation zu benutzen und die beiden resultierenden Tabellen als CSV zu exportieren.

Die Schnittstellendefinition muss dabei einen eindeutigen Namen für die Schnittstelle enthalten (dieser muss eindeutig und ein gültiger C-Bezeichner sein), die Richtung des Datenflusses, welche Treiberfunktion des Domänenrahmenwerkes für die Schnittstelle verwendet werden soll und welchem Deployable die Schnittstelle zugeordnet

```
1 Name;Direction;Driver;Deployable
2 InputInterface;in;halInDriver;ExampleDeployable
3 OutputInterface;out;halOutDriver;ExampleDeployable
```

Abbildung 3.15.: Schnittstellendefinition in CSV-Format

```
1 Global;Type;Interface;Offset
2 example_var1;int;InputInterface;0
3 example_var2;int;InputInterface;4
4 example_var3;Voltage_t;OutputInterface;0
```

Abbildung 3.16.: Variablenzuordnung zu Schnittstellen in CSV-Format

ist. Ein konkretes Beispiel dafür, welches eine Eingabe- und eine Ausgabeschnittstelle definiert, ist in Abbildung 3.15 zu sehen.

Die Zuordnung der Variablen zu den definierten Schnittstellen muss den Namen der Variable enthalten, die zugeordnet werden soll, welchen Typ diese hat, von dem unter anderem die Länge abgeleitet wird, welcher Schnittstelle sie zugeordnet ist und den Offset in Bytes innerhalb des Speicherblocks, der durch einen void-Zeiger an die Treiberfunktion übergeben wird. Ein zu Abbildung 3.15 passendes Beispiel ist in Abbildung 3.16 dargestellt.

Nicht intuitiv ist hier die Behandlung von Signal-Ports für eingehende Signale. Das Treibermodell sieht vor, dass bei Eingabeschnittstellen ein Zeiger auf einen Speicherblock ausreichender Länge übergeben wird, der vom Treiber gefüllt wird. Nach Aufruf der Treiberfunktion werden dann, gemäß der Variablenzuordnung, die Daten an den entsprechenden Offsets extrahiert und in die zugeordneten Variablen kopiert. Da Signal-Ports aber eine komplexe Datenstruktur sind, die einen Ringpuffer für mehrere Signale enthält, ist dies in diesem Fall nicht einfach anwendbar. Um Signal-Ports für eingehenden Daten zu unterstützen müssen die Schnittstellen somit als Ausgabeschnittstelle definiert werden, damit die Datenstruktur des Signal-Ports an den Treiber übergeben wird. Der Treiber muss dementsprechend so ausgelegt sein, dass er den übergebenen Signal-Port mit eingegangenen Daten füllt, anstatt vorhandene Signale auszulesen und zu versenden, wie es bei einem Treiber für ausgehende Signal-Ports der Fall wäre.

Als Folge davon, dass der Micro-Kernel-6 derzeit keine Implementierung für Kommunikationsprotokolle mit externen Systemen, wie beispielsweise über IP-Netze, hat, gibt es derzeit keinen Grund, Adressierungsinformationen für eingehende oder ausgehende Schnittstellen zu definieren. Dies ließe sich jedoch vergleichsweise einfach durch die Erweiterung der CSV-Daten umsetzen, indem beispielsweise neue Spalten für IP-Adressen und Portnummern ergänzt werden.

Für das Parsieren der CSV-Daten wird in der Implementierung die C-Bibliothek libcsv [9] verwendet.

3.4. Synchronisation

Das RTC-Paradigma sieht vor, dass alle aktiven Zustandsautomaten gleichzeitig laufen, bis sie einen stabilen Zustand erreicht haben. Erst wenn alle Zustandsautomaten einen stabilen Zustand erreicht haben, kann der nächste RTC-Schritt anlaufen. Um dies in der Praxis zu bewerkstelligen, muss es Mechanismen geben, über die die Zustandsautomaten innerhalb der Deployables synchronisiert werden können. Das muss jeweils nur in den Deployables passieren. Die Synchronisation über Deployables hinaus passiert implizit über das Versenden von Signalen am Ende jeden RTC.

Wie bereits erwähnt, gibt es für jedes Deployable einen Minor-Frame bzw. eine Partition im Prioritäten-basierten Scheduler des Micro-Kernel-6 und in jeder Partition läuft, neben den Tasks für die Zustandsautomaten ein *Dispatcher*-Task, der für das Bereitstellen und Löschen von Events zuständig ist.

Für die Synchronisation wird einfach eine Zählvariable eingeführt, welche vom Dispatcher-Task auf die Anzahl der Zustandsautomaten im Minor-Frame gesetzt wird. Danach legt der Dispatcher sich schlafen, bis diese Variable den Wert 0 erreicht hat. Ist dies der Fall, wacht der Dispatcher auf, erledigt seine Aufgaben für den aktuellen Zyklus, setzt die Variable wieder auf 0, legt sich wieder schlafen und so weiter.

Die Tasks der Zustandsautomaten schlafen initial und warten darauf, dass der Wert der Zählvariable die Anzahl der Zustandsautomaten in der Partition annimmt. Sobald dies passiert, wachen alle Zustandsautomaten-Tasks auf, durchlaufen einen RTC-Schritt, dekrementieren die Zählvariable um eins und legen sich wieder schlafen.

Im für Linux ausgelegten Original muss bei dieser Zählvariable auf Nebenläufigkeit geachtet werden. Es wird durch Locking verhindert, dass mehr als ein Thread zur Zeit auf die Variable zugreift. Im Micro-Kernel-6 ist dies nicht notwendig, da das Scheduling innerhalb einer Partition kooperativ ist, und somit kein nebenläufiger Zugriff auf die Zählvariable möglich ist.

3.5. Artefakte der Codegenerierung

Im Rahmen der Codegenerierung entstehen verschiedene Artefakte, die an verschiedenen Stellen des Arbeitsablaufes relevant werden. Dieser Abschnitt betrachtet diese

Artefakte, ihre Beziehung untereinander und an welcher Stelle des Arbeitsablaufes sie entstehen bzw. eingesetzt werden.

Am Anfang der Codegenerierung steht offensichtlich das Modell, das in Code umgesetzt werden soll, und im XMI-Format vorliegen muss. Dieses wird zusammen mit der in Abschnitt 3.3 beschriebenen Schnittstellendefinition, welche im CSV-Format vorliegen müssen, an den Codegenerator übergeben.

Der Codegenerator erzeugt daraus mehrere Dateien mit C-Code. Zunächst wird eine Header-Datei (`GeneratedTypes.h`) generiert, welche alle im Modell verwendeten Datentypen definiert. Dann wird für jedes Deployable eine C-Datei mit Namen des Deployable erzeugt, welche die Datei `GeneratedTypes.h` inkludieren und den für das jeweilige Deployable relevanten Code enthalten. In dieser Datei sind mit einigen Ausnahme alle Variablen und Funktionen als `static` deklariert, sodass sie nur innerhalb der Datei Gültigkeit haben, um Namenskonflikte zwischen den Deployables zu vermeiden.

Um die Deployables mit dem Domänenrahmenwerk zu verbinden, gibt dieses eine Datenstruktur vor, in der alle relevanten Elemente eines Deployables referenziert werden und diese, in diesem speziellen Fall, für den Scheduler des Micro-Kernel-6 aufbereiten (siehe Abbildung 3.17). Diese Datenstrukturen werden in den C-Dateien der Deployables als globale Variable definiert, die den Namen des Deployables enthalten. Um diese in einheitlicher und wiederverwendbarer Weise dem Domänenrahmenwerk zur Verfügung zu stellen, werden diese in einer weiteren generierten Datei, `GeneratedConfig.c`, in einem Array mit festem Namen zusammengefasst.

Zu guter Letzt kann man die so generierten Dateien nun zusammen mit dem Domänenrahmenwerk und dem Micro-Kernel-6 an die ARM-Compiler-Toolchain übergeben, um ein Binary zu generieren, welches auf dem BBB lauffähig ist.

In Abbildung 3.18 ist dieser Zusammenhang grafisch veranschaulicht.

```
1  struct Deployable {
2      smSpec_t **smArray;
3      task_t *smEntry;
4      struct tcb_entry_map *taskMap;
5      conditionHandle_t smStartRunCond;
6      conditionHandle_t smRtcCond;
7      size_t smNum;
8      size_t rtc;
9      hal_interface_out_t *outputArray;
10     size_t numOutputInterfaces;
11     hal_interface_in_t *inputArray;
12     size_t numInputInterfaces;
13     struct SignalPort **signalPortsToDispatch;
14     size_t numPortsToDispatch;
15     ipMessageQueueHandle_t ipipc;
16 };
17
```

Abbildung 3.17.: Datenstruktur für Deployables

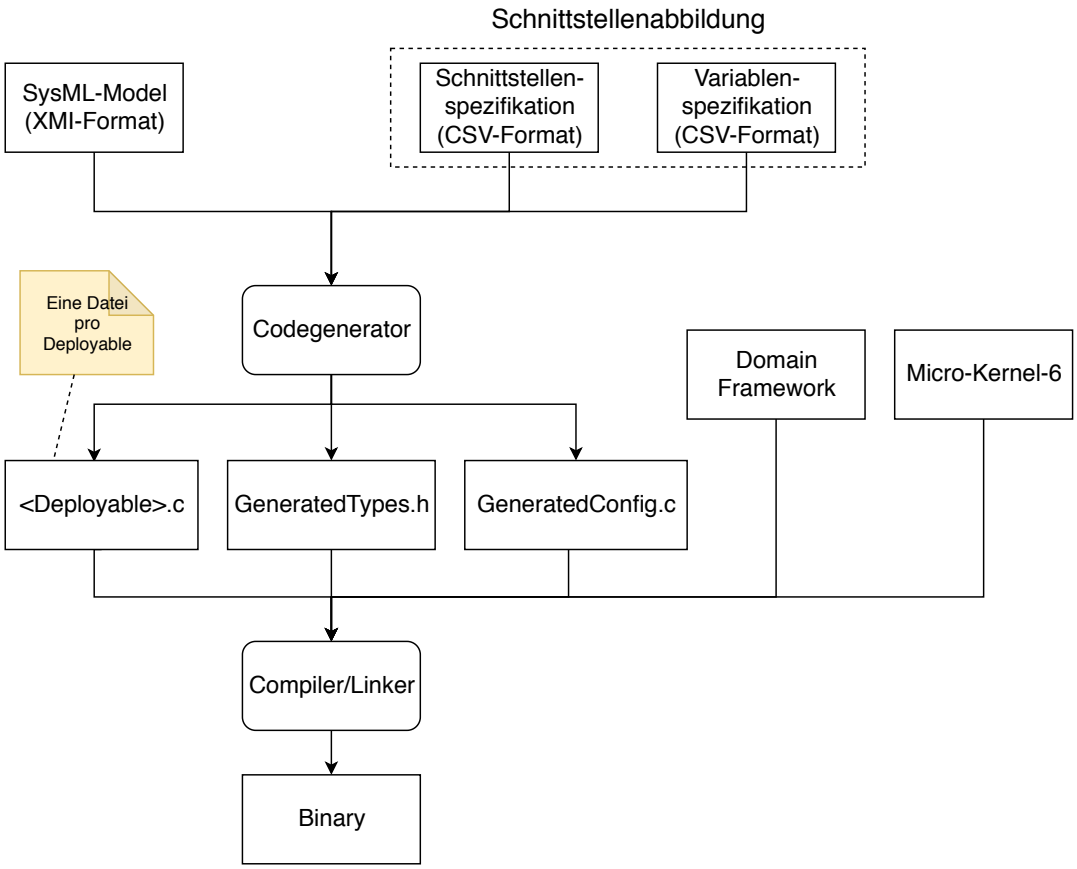

Abbildung 3.18.: Artefakte der Codegenerierung

4. Verifikation

Die Verifikation des Codegenerators und des portierten Domänenrahmenwerkes erfolgt durch eine Fallstudie an einem konkreten Modell. Zu diesem Zweck wird, wie bereits bei der Beschreibung der Vorgehensweise erwähnt, das Modell einer Blinkersteuerung aus dem Automobilbereich herangezogen. Die Schnittstellen zu den Systemen, mit denen die Steuerung interagiert, wie dem Blinkerschalter und den Lampen am Fahrzeugäußeren und Armaturenbrett, werden dabei über die General-Input/Output-Pins (GPIO-Pin) des BBB realisiert. Durch eine Reihe von manuellen Eingaben wird dann getestet, ob das mit dem generierten Code bespielten BBB die jeweils erwarteten Ausgaben produziert.

An dieser Stelle werden auch explizit keine gezielten Tests für den Nachweis der Echtzeiteigenschaften des Systems gemacht. In [12] wurde bereits gezeigt, dass sich das BBB, zusammen mit einer Vorgängerversion des Micro-Kernel-6, als Echtzeitsystem eignet. Darum wird darauf verzichtet, aufwändige Messungen zu machen, um nachzuweisen, dass dies auch im Zusammenhang mit dem hier generierten Code der Fall ist.

4.1. SysML-Modell einer Blinkersteuerung

Bei dem modellierten System (*TurnIndicationFunction*) handelt es sich um die Steuerung der Blinkeranlage, welche in vier Controller unterteilt ist. Ein Front-Controller (*FronController*), der die vorderen Blinker sowie die Anzeigen am Armaturenbrett betreibt, zwei Tür-Controller (*LeftDoorController*, *RightDoorController*), welche die Blinker in den Außenspiegeln in den Türen ansteuern und einen Heck-Controller (*RearController*), welcher die hinteren Blinker ansteuert und die eigentliche Logik enthält.

Der Heck-Controller gibt also vor, welche Blinker wann blinken sollen und teilt dies den restlichen Controllern über eine gemeinsame Schnittstelle mit. In der echten Welt wäre dies in der Regel ein Controller-Area-Network-Bus (CAN-Bus), aber für die Zwecke dieser Ausarbeitung werden die IPC-Mechanismen des unterliegenden Micro-Kernel-6 verwendet.

Die Blinkeranalge hat folgende Eigenschaften/Funktionen:

- Der Blinkerschalter (*TurnIndicationLever*) hat drei Positionen: Links, rechts und neutral.
  - Wird der Schalter auf die rechte bzw. linke Position gesetzt, wird rechts bzw. links geblinkt.
  - Wird der Schalter von der rechten oder linken auf die neutrale Position gesetzt, wird der Blinkvorgang sofort unterbrochen, sofern mindestens drei Blinkzyklen durchlaufen wurden. Ansonsten wird der Blinkvorgang nach dem dritten Zyklus eingestellt (im Folgenden wird dies als *Tipp-Blinken* bezeichnet).
  - Wird der Schalter von rechts nach links oder von links nach rechts umgelegt, wird der Blinkvorgang auf der vorherigen Seite sofort unterbrochen und auf der neuen Seite gestartet.
- Wenn links, rechts oder beidseitig geblinkt wird, so gehen die Blinklampen (*TurnIndicationLamp*) und die Anzeigelampen auf dem Armaturenbrett (*DashIndicator*) auf der jeweiligen Seite zyklisch an und wieder aus.
  - Ist die Zündung (*Iginition*) eingeschaltet, ist ein Blinkzyklus 340ms an und 340ms aus.
  - Ist die Zündung ausgeschaltet, ist ein Blinkzyklus 320ms an und 360ms aus, um Energie zu sparen.
- Liegt die von der Autobatterie (*Battery*) bereitgestellte Spannung außerhalb eines definierten Bereiches, kann nicht geblinkt werden. Wenn die Spannung während des Betriebs den definierten Bereich verlässt bzw. betritt wird der Blinkvorgang sofort unterbrochen bzw. wieder aufgenommen.
- Wird der Schalter für den Warnblinker (*EmergencyFlashingButton*) betätigt, wird beidseitig geblinkt, unabhängig von der Position des Blinkerschalters.
- Wird der Warnblinkschalter abgeschaltet, so wird der Blinkvorgang gemäß der Position des Blinkerschalters wieder aufgenommen.
- Wird während des Warnblinkens der Blinkerschalter auf die linke oder rechte Position umgeschaltet, so wird links bzw. rechts geblinkt, bis der Blinkerschalter auf die neutrale Position gesetzt wird. Ab dann wird wieder beidseitig geblinkt.
- Solange die Zündung abgeschaltet ist kann nur beidseitig, also mit dem Warnblinker, geblinkt werden. Wird die Zündung während des Links- oder Rechtsblinkens abgeschaltet, so wird der Blinkvorgang angehalten.

Die Blinkersteuerung als Gesamtsystem hat Schnittstellen zu anderen Komponenten des Fahrzeugs, die relevant für den Betrieb der Blinkeranlage sind. Diese werden als Blackbox behandelt, für die nur die Schnittstellenbeschreibung bekannt ist. Da das zur Modellierung genutzte Werkzeug Papyrus derzeit keine Kontext-Diagramme unterstützt, werden die Schnittstellen hier in Form eines IBD in Abbildung 4.1 dargestellt (für bessere Lesbarkeit des Diagramms wurden hier die Beschriftungen der Ports ausgeblendet).

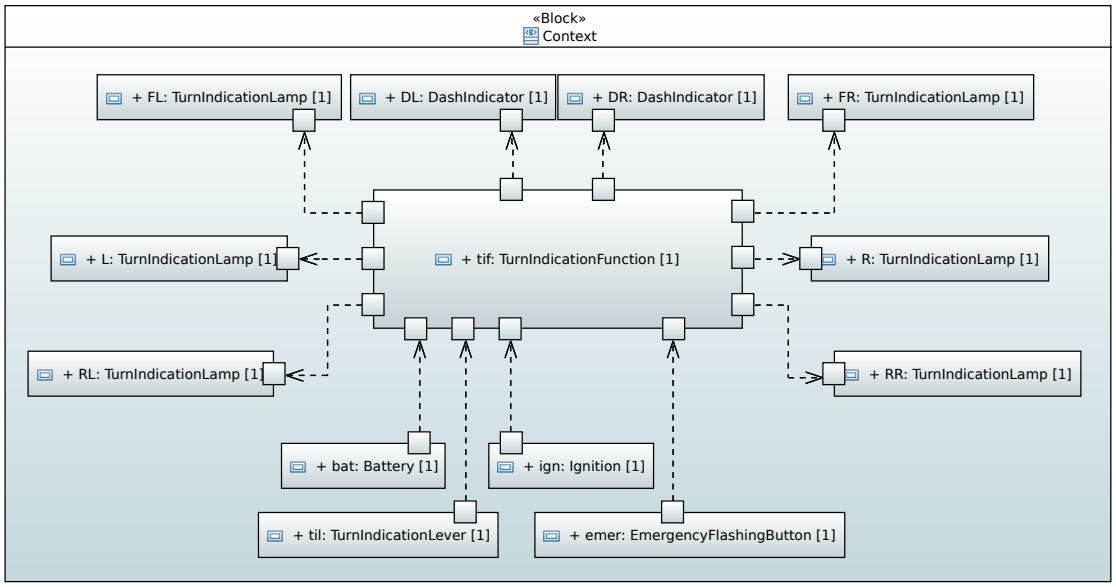

Abbildung 4.1.: Kontext-Diagramm für Blinkersteuerung

Die Schnittstellen der Controller untereinander und mit der Außenwelt sind im IBD in Abbildung 4.2 dargestellt. Zur besseren Lesbarkeit wurden auch hier die Beschriftungen der Ports selbst ausgeblendet und stattdessen die Item-Flows entsprechend ihrer Funktion beschriftet.

Es ist leicht zu erkenne, dass der Heck-Controller der einzige ist, der eingehende Daten von externen Systemen entgegen nimmt. Wie bereits gesagt, enthält dieser Controller die Logik, die entscheidet, wann und wie Lange die Blinker- und Armaturenbrettlampen leuchten sollen. Dies wird in Form von Signalen an die anderen Controller kommuniziert. Das Signal zum Einschalten der Lampen (*OnSignal*) enthält hierbei jeweils ein Flag für die linke und rechte Seite, an der die anderen Controller ablesen können, ob sie links, rechts oder beidseitig blinken sollen. Außerdem ist die Dauer in Millisekunden enthalten, für die die Lampen eingeschaltet bleiben sollen. So muss für jeden Blinkintervall nur ein *OnSignal* geschickt werden, anstatt ein *OnSignal* und ein *OffSignal*. Letzteres ist das Signal zum vorzeitigen Ausschalten der Lampen, beispielsweise weil sich die Position des Blinkerschalters geändert hat. Das *OffSignal* enthält keine zusätzlichen Daten.

Die Logik im Heck-Controller ist in drei Zustandsautomaten dekomponiert. Siehe Abbildung 4.3 für das IBD, diesmal mit Beschriftungen der Ports. Der Zustandsautomat *OnDuration*, der den Zustand der Zündung überwacht und anhand dessen die Dauer für die die Lampen während eines Blinkzykluses eingeschaltet sind definiert, ist beispielsweise in Abbildung 4.4 zu sehen. Die Variablen `durOut` und `ign` sind dabei normale Ports und `isOn` und `duration` Flow-Properties des jeweiligen Ports.

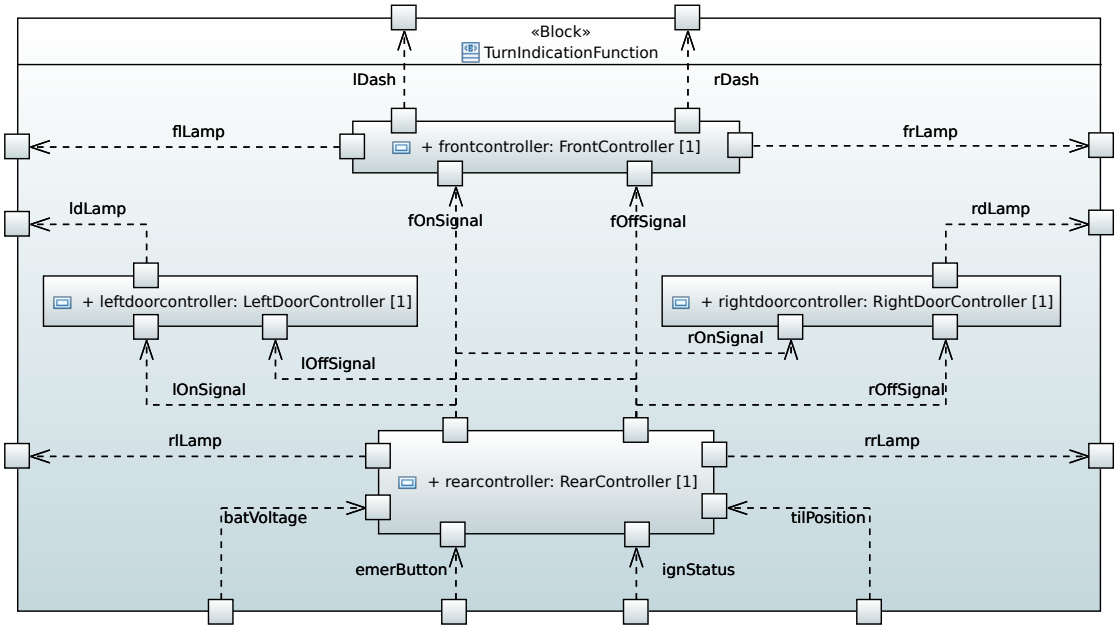

Abbildung 4.2.: Internal-Block-Diagram des Blinkersteuerungs-Blocks

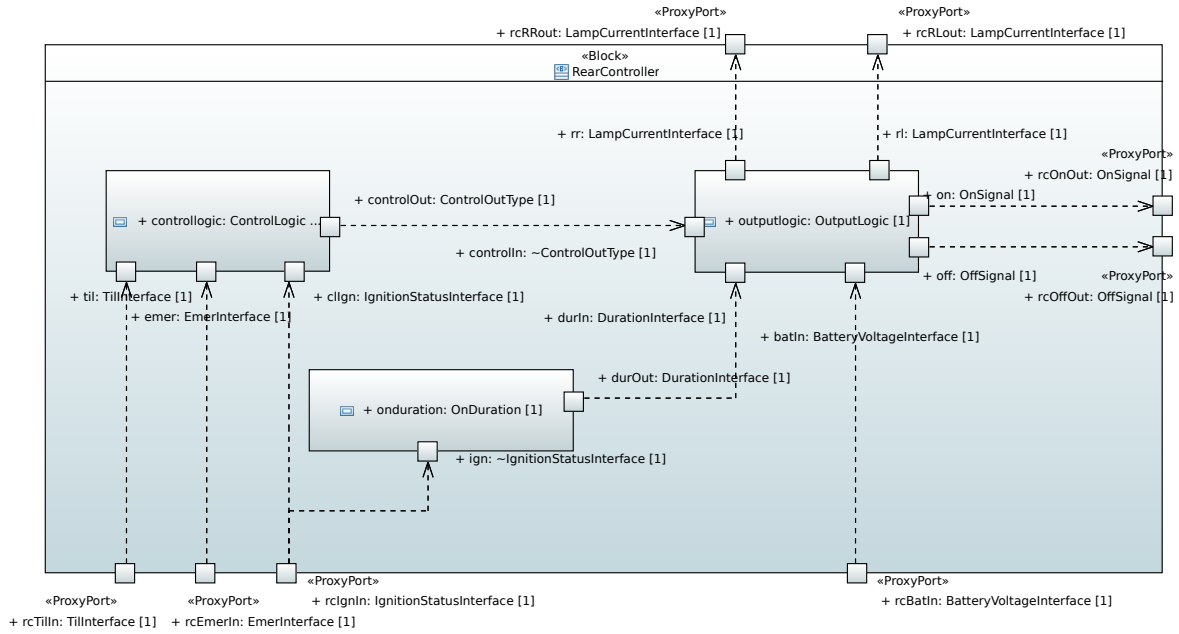

Abbildung 4.3.: Internal Block Diagramm des Heck-Controllers

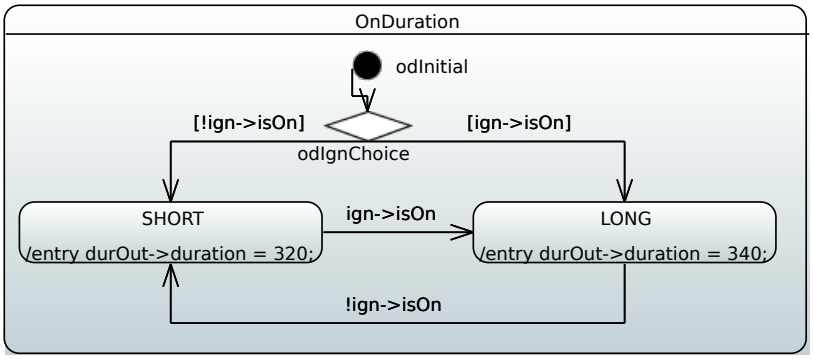

Abbildung 4.4.: Zustandsautomat zum Einstellen der Blinkzeiten

Ein weiterer Zustandsautomat *OutputLogic* entscheidet unter Berücksichtigung von Zündung, Blinkerschalter und Warnblinkerschalter ob derzeit links, rechts, beidseitig oder gar nicht geblinkt werden sollte. Der dritte und letzte Zustandsautomat *BusOutputs* bezieht zu guter Letzt die Ausgaben von *OnDuration* und *OutputLogic* und versendet anhand dieser Signale an die anderen Controller und kümmert sich gleichzeitig um das An- und Abschalten der hinteren Blinklampen. An dieser Stelle wird auch berücksichtigt, ob die verfügbare Spannung im akzeptablen Bereich liegt.

4.2. Die Testumgebung

Um den aus dem Testmodell generierten Code zu testen, wird der in Abbildung 4.5 zu sehende Testaufbau verwendet. Der generierte Code zusammen mit dem BBB wird im Folgenden als das System-Under-Test (SUT) bezeichnet.

Das Blinkermodell sieht acht Lampen vor, zwei Anzeigelampen auf dem Armaturenbrett, sowie je zwei Blinklampen vorne, an den Außenspiegeln an den Türen und hinten. Auf dem BBB sind vier LEDs fest verbaut (auf dem Diagramm mit „LEDs 1-4“ beschriftet, gezählt von oben nach unten), sodass noch vier zusätzliche LEDs die General-Purpose-Input/Output (GPIO)-Pins angeschlossen wurden („LEDs 5-8“ auf dem Diagramm, von links nach rechts), um eine LED für jede Lampe im Modell zu haben.

Die Abbildung der LEDs auf ihre Funktion im Modell ist in Tabelle 4.1 zu sehen.

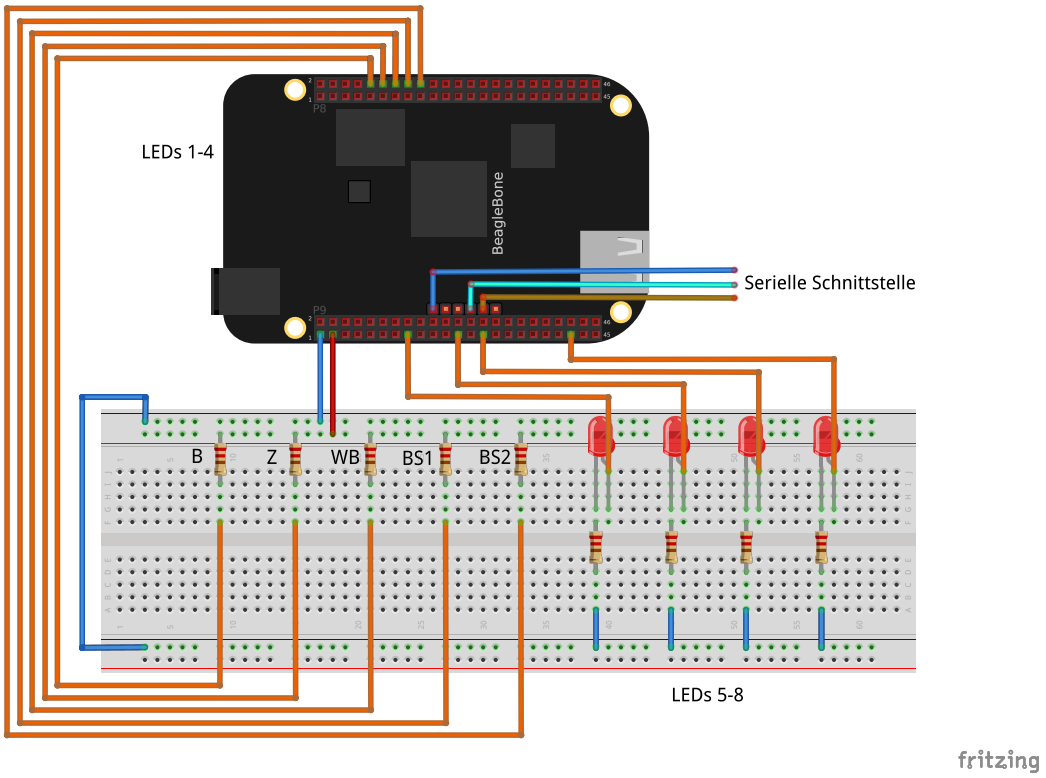

Abbildung 4.5.: Testschaltung

Tabelle 4.1.: Abbildung LEDs auf Blinkerlampen

LED	Funktion
1	Vorne links
2	Hinten links
3	Vorne rechts
4	Hinten rechts
5	Armaturenbrett links
6	Tür links
7	Armaturenbrett rechts
8	Tür rechts

Die Schalter für Zündung („Z“), Blinker („BS1“ und „BS2“) und Warnblinker („WB“) werden durch vier Kabelbrücken dargestellt, die jeweils über einen Widerstand einen 3V-Pin des BBB mit einem GPIO-Pin verbinden. Durch einfaches Stecken und Abziehen der Kabel auf der Testplatine kann so das Umlegen der jeweiligen Schalter simuliert werden. Dabei müssen die Kabel nicht auf Masse gelegt werden, um sie auf den 0-Zustand zu setzen, da das BBB konfigurierbare Pull-Widerstände an den GPIO-Pins hat, welche für diesen Zweck auf Pull-Down konfiguriert sind.

Da der Blinkerschalter drei Positionen hat, wird dieser auf zwei GPIO-Pins abgebildet („BS1“ und „BS2“). Tabelle 4.2 zeigt, wie die auf den Pins aufliegenden Werte in die Position des Blinkerschalters übersetzt werden.

Tabelle 4.2.: Abbildung GPIO-Pins auf Blinkerschalter

BS1	BS2	Blinkerschalter
0	0	*neutral*
0	1	*rechts*
1	0	*links*
1	1	*neutral*

Der Einfachheit halber wird der Wert der verfügbaren Spannungswerte auf die Äquivalenzklassen „Spannung OK“ und „Spannung nicht OK“ reduziert, welche ebenfalls durch eine einfache Kabelbrücke umgeschaltet werden können („B“). Zum Testen von Grenzfällen des konkreten Spannungswerts könnte man entweder für jeden Testwert ein separates Binary mit einem hartkodierten Wert erzeugen, oder auch Spannungswerte vom Treiber zufällig erzeugen lassen.

Die erzeugten Binaries werden jeweils über eine Programmierschnittstelle mit On-Chip-Debugger (OCD) auf das BBB übertragen, sodass gängige Debugmechanismen

angewendet werden können, um im Fehlerfall das Programmverhalten genauer zu analysieren.

Über die serielle Schnittstelle des BBB werden zudem Debugausgaben erzeugt. Insbesondere im Kontext von Echtzeitsystemen hat dies allerdings den Nachteil, dass die Textausgaben die Timings verfälschen können.

Passend zu diesen Ein-/Ausgabemodell wurden entsprechende Treiber in das Domänenrahmenwerk integriert, die wie in Abschnitt 3.3 beschrieben den entsprechenden Ports im Modell zugeordnet wurden.

4.3. Testfälle

Die vorgestellte Testumgebung gibt dem SUT durch seine fünf Eingabevariablen insgesamt 32 mögliche Kombinationen von Eingaben. Wenn man das Ausgabeverhalten des Systems auf die möglichen Werte „Blinker aus“, „links Blinken“, „rechts Blinken“, „Warnblinken“, „Tipp-Blinken links“, „Tipp-Blinken rechts“ zusammenfasst, könnte man das System in einen endlichen Zustandsautomaten mit $32 \cdot 6 = 192$ möglichen Zuständen darstellen. Angenommen jede Transition kann nur genau eine Eingabevariable verändern, dann hätte dieser Automat maximal $192 \cdot 5 = 960$ Transitionen.

Selbst nach Reduktion dieses Zustandsautomaten wären vermutlich trotzdem noch zu viele Zustände und Transitionen übrig, um das SUT mit vertretbarem Aufwand erschöpfend auf seine Korrektheit zu testen. Hinzu kommt, das das eigentliche Ziel der Tests ist, zu zeigen, dass der Codegenerator korrekten Code erzeugt, was an nur einem Beispiel im Allgemeinen sowieso nicht beweisbar ist.

Dementsprechend werden die in Abschnitt 4.1 beschriebenen Eigenschaften des Testmodells, ausgehend von sinnvollen Startzuständen, nach gängigen Anwedungsfällen einer Blinkeranlage getestet.

Sofern nicht explizit Ausgangszustand für einen Testfall angegeben wird, gilt der Folgende, der davon ausgeht, dass das Fahrzeug sich im Straßenverkehr bewegt und Verkehrstüchtig ist.

Ausgangszustand: B = 1, Z = 1, WB = 0, BS = *neutral*, Blinker = aus

Links Blinken:

Eingaben:

1. Mehrere Sekunden vor z.B. einer Kreuzung BS auf *links* schalten.
2. Nach Beendigung des Abbiegevorgangs BS auf *neutral* schalten.

Erwartetes Ausgabeverhalten:

Es wird links geblinkt, bis der Blinkerschalter wieder auf *neutral* geschaltet wird.

Rechts Blinken:

Eingaben:

1. Mehrere Sekunden vor z.B. einer Kreuzung BS auf *rechts* schalten.
2. Nach Beendigung des Abbiegevorgangs BS auf *neutral* schalten.

Erwartetes Ausgabeverhalten:

Es wird rechts geblinkt, bis der Blinkerschalter wieder auf *neutral* geschaltet wird. Dann geht der Blinker sofort aus.

Tipp-Blinken links:

Eingaben:

1. BS kurz auf *links* und sofort wieder auf *neutral* schalten.

Erwartetes Ausgabeverhalten:

Es wird drei Mal links geblinkt. Danach geht der Blinker aus.

Warten am Straßenrand:

Eingaben:

1. BS auf *rechts* schalten und an den Straßenrand fahren.
2. Nach einiger Zeit die Zündung abschalten.

3. Zündung wieder einschalten.
4. BS auf *links* schalten und weiterfahren.
5. BS auf *neutral* schalten.

Erwartetes Ausgabeverhalten:

Es wird rechts geblinkt bis die Zündung ausgeschaltet wird. Dann geht der Blinker sofort aus. Wenn die Zündung wieder eingeschaltet wird, wird wieder rechts geblinkt. Wenn der Blinkerschalter auf *links* geschaltet wird, wird sofort vom Rechtsblinken auf Linksblinken umgeschaltet. Beim letzten Schritt geht der Blinker wieder aus.

Warnblinken im fließenden Verkehr:

Eingaben:

1. Warnblinkschalter einschalten, z.B. beim Zufahren auf stockenden Verkehr.
2. Warnblinkschalter ausschalten.

Erwartetes Ausgabeverhalten:

Beim ersten Betätigen des Warnblinkschalters wird beidseitig geblinkt. Beim Abschalten geht der Blinker sofort aus.

Warnblinken mit Spurwechsel Variante 1:

Eingaben:

1. Warnblinker einschalten, z.B. beim Zufahren auf ein Stauende.
2. BS auf *links* schalten, um z.B. auf eine freie Spur zu wechseln.
3. Nach Beendigung des Spurwechsels BS auf *neutral* schalten.
4. Warnblinker wieder abschalten.

Erwartetes Ausgabeverhalten:

Beim Einschalten des Warnblinkers wird beidseitig geblinkt. Sobald der Blinkerschalter auf *links* geschaltet wird, wird links geblinkt. Nach dem Zrurückschalten des BS auf *neutral* wird wieder beidseitig geblinkt. Beim Abschalten des Warnblinkers geht der Blinker sofort aus.

Warnblinken mit Spurwechsel Variante 2:

Eingaben:

1. BS auf *links* schalten, um einen Spurwechsel anzukündigen.
2. Warnblinker einschalten.
3. Warnblinker wieder abschalten.
4. Nach Beendigung des Spurwechsels BS auf *neutral* schalten.

Erwartetes Ausgabeverhalten:

Es wird zunächst links geblinkt, dann beidseitig. Beim Abschalten des Warnblinkers wird wieder links geblinkt, bis der BS auf *neutral* geschaltet wird.

Warnblinken am Straßenrand:

Eingaben:

1. An den Straßenrand fahren und die Zündung ausschalten.
2. Warnblinker einschalten.
3. Zündung einschalten.
4. Warnblinker abschalten.

Erwartetes Ausgabeverhalten:

Der Warnblinker geht auch mit abgeschalteter Zündung an. Beim Einschalten der Zündung verlängert sich die Dauer, für die die Blinkerlampen an sind, geringfügig. Beim Ausschalten des Warnblinkschalters geht der Blinker sofort aus.

Blinken vor Zünden des Fahrzeugs:

Eingaben:

1. Zündung ausschalten.
2. BS auf *rechts* schalten.
3. Zündung einschalten.
4. Zündung ausschalten.

Erwartetes Ausgabeverhalten:

Blinker geht nicht nach Betätigen des Blinkerschalters an, aber nach Einschalten der Zündung. Blinker geht nach Abschalten der Zündung sofort aus.

Ausfall der Spannungsversorgung:

Wie im Testmodell beschrieben, ist das Blinken nur möglich, wenn die von der Batterie bereitgestellten Spannung in einem akzeptablen Bereich liegt. Um zu Testen, ob das SUT korrekt auf ungültige Spannungswerte reagiert, wurden alle zuvor beschriebenen Testfälle erneut durchgeführt und entweder vor dem Test oder währenddessen zu einem beliebig gewählten Zeitpunkt die Eingabevariable B auf 0 gesetzt.

Das erwartete Ausgabeverhalten ist, dass sobald B auf 0 gesetzt wird, kein Blinken möglich ist und ein laufender Blinkvorgang sofort unterbrochen wird.

4.4. Auswertung des Testergebnisse

Die im vorherigen Abschnitt beschriebenen Testfälle wurden alle von Hand am SUT durchgeführt. Dabei sind zunächst sowohl im Modell als auch im Codegenerator einige Fehler aufgedeckt worden, welche aber alle behoben werden konnten. Im finalen Testlauf waren somit alle Testfälle erfolgreich.

Es wurde jedoch sehr gut aufgezeigt, dass die modellgetriebene Codegenerierung kein Wundermittel zum Vermeiden von Fehlern ist. Tatsächlich wird in einigen Fällen die Fehlersuche erschwert, da nicht immer sofort erkennbar ist, ob ein Fehler im Modell, im Codegenerator, oder sogar in beiden ist. Anders als bei einer textbasierten Programmiersprache, kann es bei einer graphischen Modellierungssprache jedoch einfacher sein, durch „Draufgucken“ Modellfehler zu erkennen, da Unstimmigkeiten unter Umständen schneller ins Auge fallen als bei Text.

Im Folgenden werden ein paar nennenswerten Beispiele von aufgedeckten Fehler betrachtet.

Deadlock durch falsch initialisierte Variable

Ein Problem, welches bei initialen Tests sofort auffiel war, dass der generierte Code sporadisch in ein Deadlock lief. Nach genauerer Untersuchung stellte sich heraus, dass dies daran lag, dass generierte Variablen (also auch die Daten in normalen Ports) vom Codegenerator pauschal mit 0 initialisiert wurden. Dadurch konnte es dazu kommen, dass *OnSignal*-Signale verschickt wurden, bevor der zuständige Zustandsautomat den Wert für die Dauer der An-Zeiten der Lampen definieren konnte. Dadurch wurden dann relative Time-Events für 0ms definiert, welche vom Interpreter so behandelt werden, als wäre kein Time-Event definiert, sodass ein Zustandsautomat in einen Zustand kommen konnte, in dem keine ausgehenden Transitionen mehr feuern konnten.

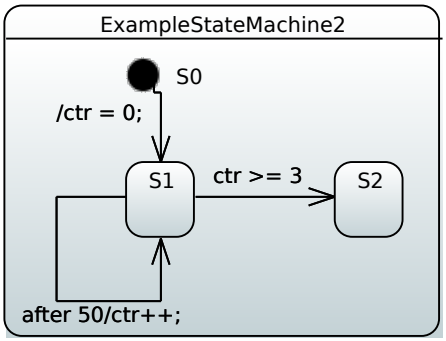

Abbildung 4.6.: Zustandsautomat der eigenes Change-Event auslöst

Dieses Problem wurde gelöst, indem der entsprechenden Flow-Property des normalen Ports, in dem die An-Zeiten der Lampen gespeichert werden, auf einen sinnvollen Initialwert gesetzt wurde. Der Codegenerator musste entsprechend angepasst werden, um solche Initialwerte aus dem Modell zu berücksichtigen.

Hinzu kommt, dass der UML-Standard keine Aussagen darüber macht, ob relative Time-Events mit einer Zeitdauer von 0 ungültig sind. Um solche Time-Events zu erlauben, wurde das in Abschnitt 3.2 vorgestellte Modell zur Repräsentation von Time-Events so abgeändert, dass kein Time-Event also NULL-Zeiger anstatt einer vordefinierten Funktion mit dem Rückgabewert 0 abgebildet werden.

Change-Events die während eines RTC entstehen

Beim Testen des Tipp-Blinkens stellte sich heraus, dass das angewandte Modell für das Bearbeiten von Change-Events eine signifikante Schwäche hat bzw. zu naiv umgesetzt wurde. Der Zustandsautomat, der sicherstellt, dass mindestens drei Mal geblinkt wird, funktioniert in etwa wie im Beispielautomat in Abbildung 4.6. Bei jedem Blinkzyklus wird eine Hilfsvariable inkrementiert und sobald diese den Wert 3 erreicht, wird ein Change-Event ausgelöst (im echten Modell wird hier mit Guards verhindert, dass das Blinken unterbrochen wird, wenn der Blinkerschalter zu diesem Zeitpunkt noch auf der entsprechenden Position steht). Da aber nach jedem RTC alle Change-Events zurückgesetzt werden, war es mit der naiven Umsetzung unmöglich auf Change-Events zu reagieren, die während eines RTC erzeugt werden.

Um unterscheiden zu können, welche Change-Events vor einem RTC und welche währenddessen eingetreten sind, wurde das in Abbildung 3.8 gezeigte Hilfsmakro so angepasst, dass vor jedem RTC alle Change-Events, die derzeit noch nicht eingetroffen sind, durch ein `skipClear`-Flag markiert werden. Beim Zurücksetzen nach einem RTC wird dann zunächst nur das `skipClear`-Flag gelöscht, sodass während des RTC eingetrete-

```
1  typedef enum eventHandler {
2      peekEvent, clearEvent, markEvent
3  } eventHandler_t;
4
5  #define CHANGE_EVENT(name,cond)                                    \
6  static int name(eventHandler_t e) {                               \
7      static int oldCndStatus = 0;                                  \
8      static int skipClear = 0;                                     \
9      int newCndStatus = (cond);                                    \
10     int indicator = eventIndicator(oldCndStatus,newCndStatus);    \
11     if ( e == peekEvent ) {                                       \
12         return indicator;                                         \
13     }                                                             \
14     if ( e == markEvent ) {                                       \
15         if ( indicator == -1 ) {                                  \
16             skipClear = 1;                                        \
17         }                                                         \
18     }                                                             \
19     if ( e == clearEvent ) {                                      \
20         if ( skipClear ) {                                        \
21             skipClear = 0;                                        \
22         } else {                                                  \
23             oldCndStatus = newCndStatus;                          \
24         }                                                         \
25     }                                                             \
26     return -1;                                                    \
27 }
```

Abbildung 4.7.: Angepasstes Makro für Change-Events

ne Change-Events erhalten bleiben und erst am Ende des nächsten RTC zurückgesetzt werden. Siehe Abbildung 4.7 für die angepasste Version.

Dabei fiel jedoch direkt das nächste Problem auf, welches jedoch am Modell selbst lag. Bei der wie in Abbildung 4.6 gestalteten Konstruktion wurde im Zustand analog zu *S1* als Entry-Action das *OnSignal* versendet. Dadurch wurde beim Eintreten des Change-Events immer noch ein viertes *OnSignal* verschickt, direkt gefolgt von einem *OffSignal*, sodass die Blinkleuchten am Ende des Tipp-Blinkens jeweils kurz aufblitzten. Da dies nicht das gewünschte Verhalten ist, wurde die Konstruktion analog zu Abbildung 4.8 angepasst, sodass hier auf ein Change-Event verzichtet werden konnte und stattdessen vor dem Betreten des Zustandes, der das *OnSignal* versendet, geprüft wird, ob der Blinkvorgang abgebrochen werden soll.

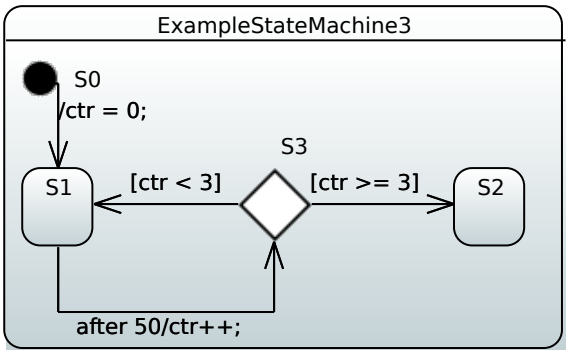

Abbildung 4.8.: Zustandsautomat der einen Choice-Node anstatt Change-Event nutzt

Abschalten der Zündung

Beim Testen, ob durch das Abschalten der Zündung der Blinkvorgang korrekt unterbrochen wird, wurde festgestellt, dass dies erst nach dem dritten Blinkzyklus passiert. Dies ergab sich daraus, dass das Modell an der Stelle, an der sichergestellt wird, dass mindestens drei Mal geblinkt wird, der Zustand der Zündung nicht direkt mit einbezogen wurde, sondern nur die abstrakte Information, ob und auf welchen Seiten gerade geblinkt werden soll. Dadurch konnte nicht entschieden werden, ob der Blinkerschalter umgelegt oder die Zündung ausgeschaltet wurde.

Um dieses Verhalten zu berichtigen, mussten ein entsprechender Port mit Item-Flow ergänzt und die zugehörigen Change-Events angepasst werden.

Textausgaben und Timings

Das folgende Problem liegt weder beim Modell noch beim Codegenerator, sondern ergibt sich aus den Anforderungen an die Echtzeiteigenschaften des SUT.

Wie bereits bei der Erläuterung der Testumgebung erwähnt, hat die Ausgabe von Text über die serielle Schnittstelle des BBB Auswirkungen auf die zeitlichen Abläufe des Systems. Zum Nachvollziehen der Programmabläufe wurden zunächst bei Zustandsübergängen in Zustandsautomaten, sowie Änderungen bei Ein- und Ausgabevariablen entsprechende Ausgaben über die serielle Schnittstelle gemacht. Dabei stellte sich heraus, dass, wenn die Zeitscheiben für die Partitionen so kurz gewählt sind, dass die LEDs augenscheinlich zur selben Zeit angehen, die Textausgaben so ineinander verzahnt sind, dass die unlesbar werden. Wählt man die Zeitscheiben lang genug, sodass keine der Textausgaben durch Präemption unterbrochen werden, dann gibt es eine mit bloßem Auge erkennbar Verzögerung zwischen dem Ein- und Ausschalten der LEDs.

Da die Textausgaben jedoch nur zu Zwecken der Nachvollziehbarkeit und Fehlerdiagnose dienen, ist dies kein wirkliches Problem, veranschaulicht aber sehr gut die Tücken der Materie.

5. Fazit und Ausblick

Ziel der Arbeit war es, einen Codegenerator zu entwickeln, der einen vertikalen Schnitt von SysML in auf dem BBB ausführbaren C-Code umsetzen kann. Dieses Ziel wurde zwar erreicht, allerdings nur durch sehr strikte Auflagen an das Modell. SysML bietet so viele alternative Möglichkeiten, funktional äquivalente Modelle zu erstellen, dass es schwer ist, diese erschöpfend zu erfassen und zu behandeln.

Hinzu kommt die Schwierigkeit, dass SysML fundamental objektorientiert ist, was schwer in einer nicht objektorientierten Sprache wie C abzubilden ist. Eine der Stärken von SysML ist es, dass man wiederverwendbare Komponenten modellieren und an verschiedenen Stellen einfach einbinden kann, um Redundanzen zu vermeiden. Durch die Anforderung, dass bestimmte Namen eindeutig sind, um das Einbetten von C-Code in das Modell zu erleichtern, wird dies stark erschwert.

Nichtsdestotrotz konnten genügend UML/SysML-Modellelemente in den Funktionsumfang des Codegenerators aufgenommen werden, um sinnvolle Modelle, aus Anwendungen in der echten Welt, zu unterstützen.

Weiterführende Arbeiten können die Ergebnisse dieser Arbeit an diversen Stellen aufgreifen und fortführen. Beispiele wären das Umsetzen von verschachtelten Zustandsautomaten, oder auch die Inbetriebnahme der Netzwerkschnittstelle des BBB, sodass Deployables auf verteilten Plattform laufen können. Ein weiterer Punkt wäre die erwähnte Problematik mit den eindeutigen Namen. Hier könnte beispielsweise ein Interpreter für C-Ausdrücke in den Codegenerator integriert werden, der Opaque-Expressions und -Behavior parsiert, darin verwendete Namen von Objekten identifiziert und diese dann durch eindeutige Langnamen ersetzt.

Anhänge

A. Abbildungsverzeichnis

2.1. Beispiel für ein SysML Block Defintion Diagram 5

2.2. Beispiel für ein SysML Internal Block Diagram 5

2.3. XMI-Kodierung eines SysML-Blocks 7

3.1. Beispiel für einen ValueType-Block 12

3.2. Generierter Code zu Abbildung 3.1 12

3.3. Beispiel für Block mit Flow-Properties 13

3.4. Generierter Code zu Abbildung 3.3 13

3.5. `smSpec_t`-Struktur für Zustandsautomateninterpreter 14

3.6. `smState_t`-Struktur für Zustandsautomateninterpreter 14

3.7. `trans`-Struktur für Zustandsautomateninterpreter 16

3.8. C-Makro für Change-Events 18

3.9. C-Struktur für Signal-Ports 20

3.10. C-Funktion zum Versenden von Signalen über Signal-Ports 21

3.11. Beispiel für ein Activity-Diagramm 24

3.12. Generierter Code zu Abbildung 3.11 24

3.13. Datenstruktur für Eingabeschnittstellen 25

3.14. Datenstruktur für die Zuordnung von Variablen zu Schnittstellen . . . 25

3.15. Schnittstellendefinition in CSV-Format 27

3.16. Variablenzuordnung zu Schnittstellen in CSV-Format 27

3.17. Datenstruktur für Deployables 30
3.18. Artefakte der Codegenerierung 30

4.1. Kontext-Diagramm für Blinkersteuerung 33
4.2. Internal-Block-Diagram des Blinkersteuerungs-Blocks 34
4.3. Internal Block Diagramm des Heck-Controllers 34
4.4. Zustandsautomat zum Einstellen der Blinkzeiten 35
4.5. Testschaltung 36
4.6. Zustandsautomat der eigenes Change-Event auslöst 43
4.7. Angepasstes Makro für Change-Events 44
4.8. Zustandsautomat der einen Choice-Node anstatt Change-Event nutzt . 45

B. Tabellenverzeichnis

4.1. Abbildung LEDs auf Blinkerlampen 37

4.2. Abbildung GPIO-Pins auf Blinkerschalter 37

C. Glossar

Controller-Area-Network Ein Bus-System für Microntroller bzw. eingebettete Systeme aus dem Automobilbereich. 53

Object Management Group Ein Industriekonsortium das für die Entwicklung von Standards für objektorierntierte Programmierung gegründet wurde. ii, 53

On-Chip-Debugger Ein Typ von Debug-System für eingebettete Systeme, welches es erlaubt Programmabläufe und -zustände direkt auf dem Chip zu betrachten. 37, 53

D. Abkürzungen und Akronyme

BBB BeagleBone Black ii, 1, 8, 9, 29, 31, 35, 37, 38, 45, 47

BDG Block-Definition-Diagram 4

CAN Controller-Area-Network 31, 53, *Glossar:* Controller-Area-Network

CSV Comma-seperated values 26–29

GPIO General-Purpose-Input/Output 31, 35, 37

IBD Internal-Block-Diagram 4, 32, 33

IPC Inter-Process-Communication 8, 26, 31

MBSE Model-Based Systems Engineering 2

MDA Model-Driven Architecture ii

MOF Meta-Object Facility 7

NuAC New Activity Calculus 21

OCD On-Chip-Debugger 37, 53, *Glossar:* On-Chip-Debugger

OMG Object Management Group ii, 7, 53, *Glossar:* Object Management Group

RTC Run-to-completion 6, 9, 16–18, 20, 25, 26, 28, 43, 44

SUT System-Under-Test 35, 38, 42, 45

SysML System Modeling Language ii, 1–4, 6–8, 10–13, 15, 19, 21, 23, 26, 47

UML Unified Modeling Language ii, 1, 3, 4, 6, 7, 10, 11, 15, 21, 43, 47

XMI XML Metadata Interchange 2, 7, 8, 29

XML Extensible Markup Language 7

E. Literaturverzeichnis

[1] Object Management Group (OMG). *Meta Object Facility*. Standard. Object Management Group (OMG), 2016-10. URL: https://www.omg.org/spec/MOF.

[2] Object Management Group (OMG). *Object Management Group Model Driven Architecture (MDA)*. Guide. Object Management Group (OMG), 2014-06. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[3] Object Management Group (OMG). *System Modeling Language (SysML) Version 1.4*. Standard. Object Management Group (OMG), 2015-08. URL: https://www.omg.org/spec/SysML/1.4.

[4] Object Management Group (OMG). *XML Metadata Interchange*. Standard. Object Management Group (OMG), 2015-06. URL: https://www.omg.org/spec/XMI.

[5] MohammadHossein AskariHemmat. „Model to code generation of UML/SysML activity diagrams for ARM CortexM MCUs“. Magisterarb. Montreal, Quebec, Canada: Concordia University, 2015-04. URL: https://spectrum.library.concordia.ca/979988/1/AskariHemmat_MASc_S2015.pdf.

[6] Steve Cook, Conrad Bock, Pete Rivett u. a. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG), 2017-12. URL: https://www.omg.org/spec/UML/2.5.1.

[7] *Eclipse Papyrus™*. Webseite. Eclipse Foundation. URL: https://www.eclipse.org/papyrus/ (besucht am 2020-01-29).

[8] Andy Evans, Robert B. France, Kevin Lano u. a. „The UML as a Formal Modeling Notation“. In: *CoRR* abs/1409.6919 (2014). arXiv: 1409.6919[1]. URL: http://arxiv.org/abs/1409.6919.

[9] Robert Gamble, Pino Toscano und bobhairgrove. *rgamble/libcsv*. Github. URL: https://github.com/rgamble/libcsv (besucht am 2020-02-19).

[10] D. Harel und E. Gery. „Executable object modeling with statecharts“. In: *Proceedings of IEEE 18th International Conference on Software Engineering*. 1996-03, S. 246–257. DOI: 10.1109/ICSE.1996.493420[2].

[1] http://arxiv.org/abs/1409.6919
[2] https://doi.org/10.1109/ICSE.1996.493420

[11] Janina Hellwege. „Automated C-Code Generation for SysML-based Architectures in the Context of Avionic Software“. Bachelorarbeit. Hamburg, Deutschland: Technische Universität Hamburg, 2016-08. URL: https://www.sts.tuhh.de/pw-and-m-theses/2016/hellwege16.pdf.

[12] Christian Manal. „Eine Plattform für Bare-Metal-Programmierung und die Entwicklung eines Hard-Realtime-Operating-Systems“. Bachelorarbeit. Bremen, Deutschland: Universität Bremen, 2019-01. URL: http://user.informatik.uni-bremen.de/manal/ba.pdf.

[13] Samir Ouchani. „A Security Verification Framework for SysML Activity Diagrams“. Diss. Montreal, Quebec, Canada: Concordia University, 2013-09.

[14] *Rational Rhapsody 8.4.0*. Webseite. IBM. URL: https://www.ibm.com/support/knowledgecenter/SSB2MU_8.4.0/com.ibm.rhp.homepage.doc/helpindex_rhapsody.html (besucht am 2020-01-29).

[15] Daniel Veillard. *The XML C parser and toolkit of Gnome*. Webseite. URL: http://www.xmlsoft.org/ (besucht am 2020-01-29).

[16] *Windchill Modeler (formerly Integrity Modeler)*. Webseite. PTC. URL: https://www.ptc.com/en/products/plm/plm-products/windchill/modeler (besucht am 2020-01-29).