technische universität
dortmund

Masterarbeit

Automatisierte Verfeinerung von Energiemodellen für eingebettete Systeme

Birte Friesel
März 2017

Gutachter:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Markus Buschhoff

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12, Arbeitsgruppe ESS
http://ess.cs.tu-dortmund.de

Zusammenfassung

Bei der Entwicklung und Nutzung von eingebetteten Systemen werden häufig Energiemodelle der einzelnen Systemkomponenten benötigt, um den Energiebedarf des Gesamtsystems abschätzen zu können. Die Erstellung und Verfeinerung solcher Modelle bedeutet meist aufwändige Handarbeit, da die notwendigen Mess- und Auswertungsschritte von den jeweiligen Komponenten abhängen. Diese Arbeit zeigt, dass es trotz der Unterschiede zwischen verschiedenen Peripheriegeräten möglich ist, mit einem generischen Konzept eine automatisierte Modellverfeinerung durchzuführen. Dazu wird anhand eines Gerätetreibers und eines vorgegebenen Automatenmodells des Geräts ein repräsentatives Testprogramm generiert, eine Reihe von Messungen durchgeführt und ausgewertet und das Automatenmodell zu einem Energiemodell verfeinert. Falls das Modell konfigurierbare Parameter wie Sendeleistung oder Datenrate angibt, werden Abhängigkeiten von diesen Parametern automatisch erkannt und analytisch beschrieben. Zusätzlich werden zwei Methoden zur Modellierung der Transitionsenergie verglichen. Eine Evaluation einer prototypischen Implementierung mit verschiedenen Arten von Peripheriegeräten zeigt, dass viele statische und dynamische Modelleigenschaften zuverlässig erkannt und modelliert werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seien es Smartphones, vernetzte Feuermelder oder Funk-Autoschlüssel: Ein Alltag ohne batteriebetriebene eingebettete Systeme ist heutzutage nur noch schwer vorstellbar. Doch auch einige alltägliche Ärgernisse sind erst mit ihnen aufgekommen. Je mehr solcher Systeme im Umlauf sind, desto mehr können plötzlich das Ende ihrer Akkulaufzeit erreichen und so das gesamte System bis zum Austausch oder Wiederaufladen der Batterie unbenutzbar machen.

Zur Vermeidung derartiger Unannehmlichkeiten ist es wünschenswert, die erwartete Laufzeit möglichst gut abschätzen zu können, um zumindest nicht von leeren Batterien überrascht zu werden. Dies gilt insbesondere bei System an abgelegenen Orten, wo ein Batteriewechsel nicht praktikabel ist, wie z.B. GPS-Sendern zur Beobachtung von Zugvögeln. Hier ist das Ziel stattdessen, den Einsatz so zu konzipieren, dass die Laufzeit für die geplanten Tätigkeiten ausreicht.

Um diese Laufzeit zu bestimmen, wird möglichst detaillierte Kenntnis über die im Akku vorhandene sowie die zur Laufzeit vom System genutzte Energie benötigt. Dies erfordert insbesondere ein Energiemodell, welches das energetische Verhalten des Systems beschreibt und eine Bestimmung des Energiebedarfs einzelner Aktionen und Anwendungsfälle erlaubt. Die dazu benötigten Daten können oft nicht vollständig aus Datenblättern entnommen werden, so dass eine individuelle Modellierung mit Hilfe eigens für das System entworfener Messreihen notwendig ist. Messreihenerstellung, Auswertung und Modellverfeinerung sind dabei zu großen Teilen Handarbeit.

Ziel dieser Arbeit ist die automatisierte Erstellung und Verfeinerung von Energiemodellen für Peripheriegeräte wie Funkmodule oder Temperatursensoren. Auf Basis eines Gerätetreibers und einer maschinenlesbaren Hardwarebeschreibung soll automatisch ein Testprogramm generiert, das Gerät vermessen und aus dieser Messung ein Energiemodell erstellt werden. Zudem soll die Güte des Modells beurteilt und bei Bedarf automatisch auf mögliche Fehlannahmen hingewiesen werden. Modelle des Akkuverhaltens und die letztendliche Laufzeitberechnung liegen hier nicht im Fokus.

Zustand	Energie	Zeit-Anteil
Aus	10 µW	99 %
Daten erheben	100 µW	0,8 %
Daten senden	23 mW	0,2 %
Durchschnitt	56,7 µW	100 %

Tabelle 1.1: Ein einfaches Energiemodell mit bekanntem Anwendungsfall.

In den folgenden Abschnitten werden Hintergrund und Ziele dieser Arbeit näher erläutert und der Aufbau des restlichen Dokuments beschrieben.

1.1 Motivation und Hintergrund

Energiemodelle sind bereits seit der Jahrtausendwende ein zentrales Forschungsthema im Bereich eingebetteter Systeme. Sie berühren fast jeden Anwendungsfall batteriebetriebener Systeme und unterscheiden sich lediglich durch ihre Relevanz: Während eine wegen fehlerhafter Energiemodelle falsch geschätzte Laufzeit bei GPS-Sendern oder Sensornetzen ganze Experimente gefährden kann, sind solche Fehler bei Smartphones oder Notebooks eher lästig als kritisch. In jedem Fall ist eine akkurate Modellierung von Energiebedarf und Systemlaufzeit aber wünschenswert. Es ist daher nicht überraschend, dass es mittlerweile eine Vielzahl von Veröffentlichungen gibt, die sich mit dem Vermessen und Modellieren des Energiebedarfs eingebetteter Systeme befassen.

Der Kerngedanke eines Energiemodells ist, den Energiebedarf eines Systems abhängig von Systemeigenschaften wie Laufzeitvariablen oder Hardware-Zuständen zu beschreiben. Variablen können beispielsweise die eingestellte Funkdatenrate, Messfrequenz oder Sendeleistung sein. Zustände können sowohl lediglich zwischen „an“ und „aus“ unterscheiden als auch feingranulare Unterteilungen wie „senden“, „empfangen“, „betriebsbereit“ und „aus“ verwenden. Auch Mischkonzepte, die beispielsweise den Zustand „senden mit Sendeleistung x“ enthalten, sind möglich.

Jeder Variablen und jedem Zustand wird eine Berechnungsvorschrift des zugehörigen Leistungs- oder Energiebedarfs zugeordnet, so dass zu einem vorgegebenen Anwendungsfall das Energieverhalten des Systems abgeschätzt werden kann. Ein einfaches Modell für ein fiktives Sensormodul findet sich in Tabelle 1.1. Aus der durchschnittlichen Leistung des Anwendungsfalls und der Batteriekapazität lässt sich die Lebensdauer des Moduls abschätzen.

In der Praxis gibt es eine Vielzahl verschiedener Modellierungsansätze. Der Detailgrad reicht von einer groben Betrachtung auf System- oder Komponentenebene bis hin zur Modellierung einzelner Transistoren und Leiterbahnen sowie der dort auftretenden Schaltverluste. Das Modell selbst kann wahlweise losgelöst vom modellierten System (*offline*) oder

zur Laufzeit (*online*) eingesetzt werden. Ersteres erlaubt die einmalige Bestimmung des Energiebedarfs und Evaluation von Anwendungsfällen und z.B. Funkprotokollen, zweiteres ermöglicht Buchführung über den Energiezustand und adaptives Verhalten.

All diesen Ansätzen ist gemeinsam, dass die Genauigkeit des erstellten Modells von den getroffenen Annahmen über das Hardwareverhalten und der Qualität der erhobenen Messdaten abhängt. Die Modellierung selbst ist häufig ein iterativer Prozess, in dem zunächst ein einfaches Modell erstellt und dann solange verfeinert wird, bis es die gewünschte Güte erreicht hat. Zusätzlich müssen Modelle für jedes neue System individuell kalibriert werden, da verschiedene Instanzen der gleichen Peripheriekomponente wegen Fertigungstoleranzen unterschiedlichen Energiebedarf haben können [Hur+11].

Neben der mindestens einmaligen Erstellung eines Testprogramms, welches die Hardware durch die zu modellierenden Zustände führt, ist also potentiell eine Vielzahl von Messungen notwendig – üblicherweise mindestens eine Messreihe pro Verfeinerung oder Kalibrierung. All diese Messungen müssen ausgewertet und ihre Ergebnisse in das Modell zurückgeführt werden. Oft geschieht dieser Prozess zu großen Teilen in Handarbeit; teilweise vorhandene automatisierte Auswertungen sind häufig testprogrammspezifisch und daher nicht leicht auf andere Testprogramme oder gar andere Hardware übertragbar.

1.2 Ziele und Konzept

Diese Arbeit soll zeigen, dass es trotz der großen Unterschiede zwischen einzelnen Peripheriekomponenten möglich ist, mit generischen Verfahren und minimalen manuellen Vorgaben praktisch nutzbare Energiemodelle automatisch zu erzeugen. Dazu wird ein Konzept entwickelt, mit dem Energiemodelle für beliebige Peripheriegeräte anhand vorgegebener Hardwaremodelle und Gerätetreiber erzeugt und verfeinert werden können, und anhand einer ebenfalls im Rahmen dieser Arbeit erstellten Referenzimplementierung evaluiert.

Anhand dieses generischen Konzepts werden zudem Auswertungsmethoden zur Modellverfeinerung konzipiert und mit Hilfe verschiedener Peripheriegeräte evaluiert. Der Fokus liegt dabei auf der Erkennung und Beschreibung von Parameterabhängigkeiten in Hardwarezuständen, wie beispielsweise die Abhängigkeit der Sendeleistung von der eingestellten Bitrate oder die in bestimmten Fällen gegebene Unabhängigkeit der Übertragungsdauer von der Paketlänge.

Sekundärziel der Arbeit ist die bessere Reproduzierbarkeit von Messergebnissen und Energiemodellen durch die Verwendung automatisch generierter Testprogramme und automatisierter Auswertungsmethoden.

Das Konzept orientiert sich am typischen Vorgehen zur manuellen Erstellung eines Energiemodells. Dieses besteht im Wesentlichen aus vier Schritten.

1. Identifikation der zu vermessenden Hardwarezustände und -eigenschaften.
2. Erstellung eines dazu passenden Testprogramms.

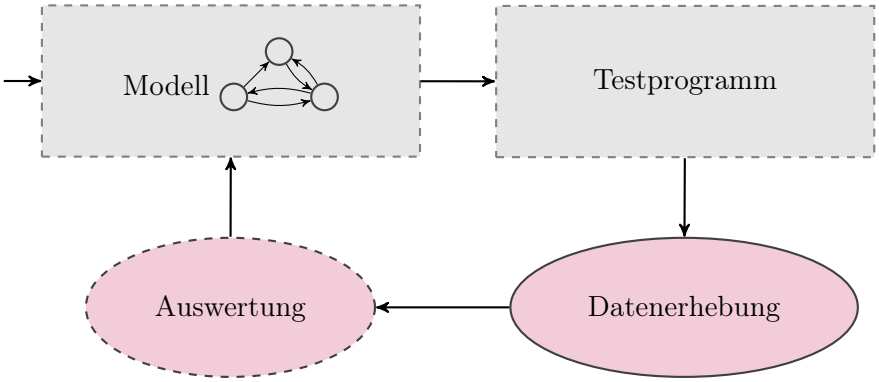

Abbildung 1.1: Ablauf bei Erstellung eines Energiemodells. Die gestrichelten Elemente sind üblicherweise Handarbeit.

3. Vermessung der Hardware bei Ausführung des Programms.
4. Auswertung der Messwerte und Erstellung oder Anpassung des Energiemodells.

Wenn sich das so erstellte Energiemodell als unzulänglich herausstellt, werden die Schritte zwei bis vier mit einem angepassten Modell erneut durchgeführt. Dies wird solange wiederholt, bis eine zufriedenstellende Modellgüte erreicht wurde. Es ergibt sich der in Abbildung 1.1 gezeigte iterative Prozess.

Das dort abgebildete Modell hat zwei verschiedene Bedeutungen. Im ersten Durchlauf beschreibt es lediglich, welche Zustände und sonstigen Eigenschaften der Hardware bei der Erstellung des Testprogramms berücksichtigt werden. Es handelt sich um ein reines Hardwaremodell, welches noch nicht einmal explizit vorliegen muss – das Modellwissen kann auch lediglich implizit im von Hand erstellten Testprogramm enthalten sein. Ein Beispiel für eine solche implizite Annahme ist, dass ein Display entweder ein- oder ausgeschaltet ist, und die Vermessung von Änderungen am Bildschirminhalt nur im eingeschalteten Zustand sinnvoll ist.

Nach der ersten Messdatenauswertung wird dieses Modell um Energiedaten erweitert, das Hardwaremodell wird also zu einem Energiemodell *verfeinert*. Stellt sich heraus, dass dieses noch nicht genau genug ist, da zum Beispiel der Energiebedarf des Displays von der Anzahl heller oder dunkler Pixel abhängt, wird das Modell angepasst. Anschließend werden mit einem ggf. ebenfalls angepassten Testprogramm die dafür fehlenden Daten erhoben und im Energiemodell eingepflegt. Auch dies ist ein Verfeinerungsschritt.

Im Rahmen dieser Arbeit soll die Erstellung des Testprogramms, die Erhebung und Auswertung der Messwerte und die Verfeinerung des Modells automatisiert werden. Lediglich die initiale Vorgabe des Hardwaremodells sowie einzelne Anpassungen daran werden weiterhin manuell durchgeführt. Grund hierfür ist, dass ein verlässliches Hardwaremodell zwingend zur Generierung eines Testprogramms notwendig ist – denn wenn nicht bekannt ist, welche Zustände überhaupt vermessen werden sollen und wie zwischen diesen gewechselt werden kann, kann auch nicht viel ausgewertet werden.

1.3 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die für diese Arbeit relevanten Grundlagen zu Leistung und Energie sowie deren Messung und Modellierung vermittelt. Ebenso werden das später zur Implementierung genutzte Betriebssystem, das bereits im Konzept berücksichtigte Messgerät und die verwendete Hardwareplattform als Beispiele für in diesem Kontext übliche Komponenten vorgestellt.

Nach einer Übersicht über verwandte Arbeiten in Kapitel 3 folgt in Kapitel 4 eine detaillierte Beschreibung des Konzepts zur Modellierung, Testprogrammerstellung, Auswertung und Modellverfeinerung. Die dazu entwickelte Referenzimplementierung wird in Kapitel 5 vorgestellt und in Anhang A vertieft. Kapitel 6 dient schließlich der Bewertung der verschiedenen Teilkonzepte und der Evaluation der für repräsentative Peripheriegeräte erstellten Modelle. Die Arbeit endet mit einem Fazit der erreichten Ergebnisse und offener Fragestellungen in Kapitel 7.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die dieser Arbeit zu Grunde liegenden Konzepte zur Messung und Modellierung von Leistung und Energie sowie allgemeine Methoden zur Modelloptimierung vor. Zusätzlich beschreibt es das Betriebssystem Kratos, ein dafür unter dem Namen *DFA-Treiber* entwickeltes Konzept energiegewahrer Gerätetreiber, die Messplattform MIMOSA und den MSP430FR5969-Mikrocontroller.

Bis auf MIMOSA sind letztere erst im Implementierungsabschnitt relevant. Da Energiemodelle und darauf aufbauende Testprogramme aber eng mit dem sie ausführenden eingebetteten Betriebssystem und dessen Treiberschicht verknüpft sind, welches wiederum von der zu Grunde liegenden Hardware abhängt, werden diese bereits hier vorgestellt. MSP430FR5969 und Kratos dienen damit gleichzeitig als Beispiele für die Hard- und Software-Umgebung eines eingebetteten Systems.

2.1 Leistung und Energie

Zur Bewertung und Verfeinerung von Energiemodellen ist es notwendig, den modellierten Energiebedarf eines Peripheriegeräts mit realen Messwerten zu vergleichen. Dieser Abschnitt dient der Auffrischung der grundlegenden Zusammenhänge zwischen Leistung, Energie und verwandten Messgrößen sowie der am häufigsten genutzten Messmethoden.

Die grundlegende Einheit eines Energiemodells ist die Energie E mit der Einheit Joule. Aus elektrotechnischer Sicht drückt sie die in einem bestimmten Zeitraum verrichtete Arbeit aus. Wird beispielsweise eine 40 W-Lampe für eine Sekunde betrieben oder eine 20 W-Lampe für zwei Sekunden, setzt sie in diesem Zeitraum 40 J elektrischer Energie in Licht und Wärme um. Da die Energie danach nicht mehr in elektrischer Form vorliegt, wurde sie aus Sicht des Energiemodells verbraucht.

Zur Bestimmung der verbrauchten Energie ist also Kenntnis über die umgesetzte Leistung P und die verstrichene Zeit t notwendig. Die Leistung ist im allgemeinen variabel und

wird von einer zeitabhängigen Funktion $P(t)$ beschrieben, so dass die in einem Zeitfenster $[t_1, t_2]$ verbrauchte Energie als Integral der Leistung über die Zeit bestimmt werden kann.

$$E = \int_{t_1}^{t_2} P(t)\,\mathrm{d}t \tag{2.1}$$

Zur Nutzung von Energiemodellen ist dies bereits der wichtigste Zusammenhang. Führt ein Peripheriegerät eine Aktion mit einer festen Dauer aus, kann deren Energie E modelliert und bei jeder Ausführung der Aktion zum bisherigen Energieverbrauch addiert werden. Befindet es sich hingegen für einen variablen Zeitraum in einem bestimmten Zustand, ist die Energie zunächst unbekannt. Hier wird stattdessen die mittlere Leistung P modelliert, so dass zusammen mit der zur Laufzeit bestimmten Aufenthaltszeit t die verbrauchte Energie mittels $E = P \cdot t$ bestimmt werden kann. In der Praxis enthalten Energiemodelle daher meist ausschließlich Energie- und Leistungsangaben.

Bei der Erstellung von Energiemodellen sind jedoch weitere Zusammenhänge wichtig, da die Leistungsaufnahme eines Verbrauchers (welcher in diesem Kontext auch *DUT* für *Device Under Test* genannt wird) nicht direkt gemessen werden kann. Sie hängt wiederum von der am Verbraucher anliegenden Spannung U und dem durch ihn fließenden Strom I ab: es gilt $P = U \cdot I$. Da die Spannung U meist konstant ist, genügt zur Leistungs- und Energiebestimmung in diesen Fällen die Integration des geflossenen Stroms.

$$E = U \cdot \int_{t_1}^{t_2} I(t)\,\mathrm{d}t \tag{2.2}$$

In der Praxis wird die Spannung häufig im Vorfeld auf einen festen Wert eingestellt, so dass die Messung des Stroms I genügt. Dieser ist jedoch im Gegensatz zur Spannung nicht direkt messbar; es können lediglich seine Auswirkungen auf andere Bauteile beobachtet und daraus der Strom berechnet werden.

Eine wegen ihrer Einfachheit häufig verwendete Methode dazu ist die Messung des Spannungsabfalls U_R über einen Messwiderstand (auch *Shunt* genannt) der Größe R. Dieser wird mit dem Verbraucher in Reihe geschaltet, so dass durch beide Bauteile der gleiche Strom I fließt. Der vom Ohmschen Gesetz beschrieben Spannungsabfall am Widerstand ist wiederum proportional zum Stromfluss ($U_R = R \cdot I$), so dass sich der Strom auch als $I = \frac{U_R}{R}$ bzw. die Leistung als $P = \frac{U \cdot U_R}{R}$ bestimmen lässt. Präzise Kenntnis über U_R genügt also, um die verbrauchte Energie zu erfassen.

$$E = \frac{U}{R} \cdot \int_{t_1}^{t_2} U_R(t)\,\mathrm{d}t \tag{2.3}$$

Die Spannung U_R kann schließlich leicht mit Hilfe von Analog-Digital-Wandlern (kurz *ADC* für *Analog Digital Converter*) bestimmt werden. Der Großteil heutiger Mess-Systeme

verwendet daher dieses Prinzip (und somit obige Formel), um den Strom zu messen und Leistung und Energie zu berechnen. Dabei sind aber einige Aspekte zu beachten, die die Genauigkeit der Messdaten beeinträchtigen.

Ein Kernproblem ist die zur digitalen Weiterverarbeitung notwendige Diskretisierung der Messwerte. Einerseits hat der dazu verwendete Analog-Digital-Wandler bauartbedingt eine feste Präzision; so kann ein 12 Bit-Wandler beispielsweise nur $2^{12} = 4096$ verschiedene Spannungswerte unterscheiden. Bei einem Messbereich bis 1 V können also nur Spannungsdifferenzen in Schritten von $\frac{1\,\mathrm{V}}{4095} \approx 244\,\mu\mathrm{V}$ unterschieden werden, bei indirekter Messung von Strömen bis 500 mA analog nur Stromdifferenzen in Schritten von $\frac{500\,\mathrm{mA}}{4095} \approx 122\,\mu\mathrm{A}$.

Andererseits benötigt das Bestimmen und Auslesen der Spannung im Wandler eine gewisse Zeit und auch Bandbreite und Speicherplatz des Messrechners sind begrenzt. Die Funktion $U_R(t)$ kann zwar durch periodische Abtastung approximiert werden, ist jedoch nie exakt bekannt. Insbesondere können Spannungs- bzw. Stromschwankungen, deren Frequenz oberhalb der halben Abtastrate liegt, nicht zuverlässig erfasst werden. In der Praxis liegt diese Grenze je nach Gerät zwischen wenigen Hz und einigen hundert kHz.

Zusätzlich beeinflusst die indirekte Strommessung das vermessene Gerät und somit die Messergebnisse. Durch den stromabhängigen Spannungsabfall am Messwiderstand ist die am Gerät anliegende Versorgungsspannung U_{DUT} nicht mit U identisch, sondern sinkt proportional zum momentanen Stromverbrauch: Es gilt $U_{DUT} = U - U_R = U - R \cdot I$. Je nach Dimension des Widerstands kann dieser Einfluss erheblich sein. Fließen beispielsweise 20 mA durch einen 50 Ω-Widerstand, ergibt dies einen Spannungsabfall von 1 V.

Einige Messgeräte entschärfen diesen Aspekt durch Kompensation des Spannungsabfalls. Wird die Reihenschaltung aus Shunt und DUT beispielsweise nicht mit U, sondern mit der dynamischen geregelten Spannung $U + U_R$ versorgt, so ergibt sich aus Sicht des Peripheriegeräts eine konstante Versorgungsspannung. Der Shunt kann dennoch nicht beliebig groß gewählt werden, da die Spannung $U + U_R$ in der Praxis begrenzt ist.

Dies unterstreicht, wie wichtig die Auswahl eines passenden Shunts ist. Er muss muss einerseits groß genug sein, um die auftretenden Ströme mit einer hinreichenden Auflösung messen zu können, gleichzeitig aber so klein, dass der dabei auftretende Spannungsabfall die Messung nicht zu stark beeinflusst. Es ist daher sinnvoll, für jedes Gerät auf Basis des erwarteten maximalen Stromflusses einen individuellen Shunt auszuwählen. Bei Messungen mit hohem Dynamikumfang, in denen sowohl Ströme im zweistelligen mA- als auch im unteren µA-Bereich auftreten, müssen u.U. Genauigkeitseinbußen im unteren Messbereich in Kauf genommen werden.

Es zeigt sich, dass die Grundlagen zur Modellierung zwar einfach sind, bei der Erhebung der Daten aber einige Fallstricke zu beachten sind. Dies wird auch bei der Betrachtung verschiedener Mess-Systeme in Kapitel 3 deutlich werden.

2.2 Modellierung

Die Modellierung von Energie ist Teil des Themenbereichs Ressourcenmodellierung. Neben der Energie wird dort häufig auch der verfügbare Arbeitsspeicher als Ressource betrachtet; modelliert wird meist auf Basis von Funktionsaufrufen oder des Zustands einzelner Hardwarekomponenten. Es ist jedoch auch eine deutlich detailliertere Energiemodellierung auf der Ebene einzelner CPU-Instruktionen möglich.

Als zu modellierende Ressource weisen Arbeitsspeicher und Energie signifikante Unterschiede auf. Arbeitsspeicher ist eine wiederverwendbare Ressource, die nach jedem Funktionsaufruf wieder freigegeben wird, während verbrauchte Energie dem System später nicht mehr zur Verfügung steht. Dieser Abschnitt beschränkt sich daher auf Methoden, die von vornherein zur Energiemodellierung vorgesehen sind. Nach einem Überblick über die verschiedenen Konzepte werden automatenbasierte und analytische Modelle näher betrachtet.

2.2.1 Übersicht

Zunächst stellt sich die Frage, ob das Modell im Kontext eines eingebetteten Betriebssystems oder losgelöst davon verwendet werden soll. Beides hat Vor- und Nachteile: In das System integrierte Modelle erlauben adaptives Verhalten, können aber nicht beliebig komplex sein, während davon losgelöste Modelle zwar beliebig komplex und somit potentiell beliebig genau werden können, zur Laufzeit aber nicht nutzbar sind.

Für das betriebssystemunabhängige *Offline Modeling* findet sich in der Literatur eine Vielzahl von Methoden, von denen einige auch zur Energiemodellierung geeignet sind.

- *Prozessalgebren* beschreiben interagierende Prozesse. Meist werden sie für wiederverwendbare Ressourcen wie RAM eingesetzt, die Variante P^2ACSR unterstützt aber auch Energiemodelle [LPS07; VS08].
- Auch die UML-Variante *UML/SPT* (*UML for Schedulability, Performance and Time*) beschreibt solche Prozesse und ihre Ressourcenanforderungen. Die Modellierung von Energie ist mit einigen Implementierungen möglich [VS08].
- Zeitautomaten mit Kosten (kurz *PTA* bzw. *Priced Timed Automata*) beschreiben das Zeit- und Energieverhalten eines Systems, welches aus mit Kosten (d.h. Leistung oder Energiebedarf) und Zeitbeschränkungen annotierten Zuständen und Transitionen besteht [VS08].
- Analytische Modelle nutzen Laufzeiteigenschaften wie den Leerlauf-, Sende- oder Empfangsanteil an der Gesamtlaufzeit [Zha+10; Mur+12]. Der Energiebedarf ergibt sich dann z.B. als $t \cdot (idle\_ratio \cdot P_{idle} + send\_ratio \cdot P_{send} + recv\_ratio \cdot P_{recv})$ für die Laufzeit t, Zeitanteile $idle\_ratio$, $send\_ratio$ und $recv\_ratio$ und zugehörige Leistungsdaten P_{idle}, P_{send} und P_{recv}.

Zur Umsetzung des im Betriebssystem integrierten *Online Modeling* bietet die Literatur im Wesentlichen drei Varianten.

- *Macro Modeling* ordnet jedem Funktionsaufruf eines Gerätetreibers eine Formel zu, welche den zu erwartenden Energiebedarf anhand der Argumente bestimmt. Die Formelstruktur (z.B. $send(length = x) = ax + b$) wird vorgegeben und die Parameter (hier a und b) anhand von Messungen bestimmt [TRJ02; Tan+02].
- PTAs können durch Anpassungen am Treiberquelltext zur Laufzeit mitberücksichtigt werden [Fal14]. Eine Synthese von Makromodellen aus PTAs ist ebenfalls möglich [BGS12].
- Wenn Systemeigenschaften wie die Sende- oder Empfangshäufigkeit zur Laufzeit bekannt sind, können analytische Modelle auch online genutzt werden [DZ11].

Für Peripheriegeräte sind insbesondere Zeitautomaten interessant, da diese die verschiedenen Zustände und Zustandsübergänge solcher Hardwarekomponenten auf eine natürliche Art modellieren können. Die zugehörigen Treiber müssen diese Zustände und Transitionen ohnehin kennen und können daher leicht um die Leistungs- und Energie-Angaben aus einem Automaten erweitert werden. Prozessalgebren und UML/SPT sind hingegen primär für interagierende Prozesse vorgesehen und bei Betrachtung von Peripheriegeräten weniger hilfreich.

Auch zur Erstellung von Modellen auf Basis von Laufzeiteigenschaften sind PTAs nützlich. Zur Energieberechnung ist Kenntnis über die mittlere Leistungsaufnahme in jedem Systemzustand notwendig, welche ein Zeitautomat bereits enthält. Tatsächlich unterscheiden sich viele Modelle auf Automatenbasis von analytischen primär dadurch, dass erstere die Zustände der Hardware explizit modellieren, während zweitere sie lediglich implizit durch die Zeitanteile berücksichtigen.

In der Praxis sind beide relevant. Analytische Modelle sind insbesondere im Offline-Bereich nützlich, um schnell verschiedene Arbeitslasten wie Funkprotokolle oder Sensoranwendungen miteinander zu vergleichen. Online sind sie hingegen nur anwendbar, wenn die genutzten Laufzeitvariablen zur Verfügung stehen oder leicht zur Verfügung gestellt werden können – dies ist unter anderem bei Smartphone-Betriebssystemen wie Android der Fall.

Modelle auf Automatenbasis sind sowohl online wie auch offline nützlich. Online können sie entweder direkt oder in Form von Makromodellen genutzt werden, offline erlauben sie zudem beliebig komplexe Modelle, da sie dort nicht durch die Arbeitsspeicher- oder Rechenleistungsbeschränkungen eingebetteter Systeme gebunden sind. Gleichzeitig sind sie flexibler, als eine ausschließlich von Laufzeitparametern abhängige Formel und können je nach Umsetzung sogar selbst parametrisierte Teilformeln enthalten. Es ist daher nicht überraschend, dass sie gegenüber rein analytischen Modellen zur Modellierung komplexer Hardware überlegen sind [McC+11].

Aus diesem Grund wird im Folgenden nur noch Energiemodellierung mit Hilfe von PTAs und Teilformeln betrachtet.

2.2.2 Zeitautomaten mit Kosten

Zeitautomaten sind endliche Automaten, die um Uhren und Uhrbedingungen erweitert wurden. Transitionen können Uhren auf Null zurücksetzen, zudem können Zustände und Transitionen einfache Bedingungen an Uhren stellen. Beispielsweise lässt sich definieren, dass eine Transition nur möglich ist, wenn der Zähler einer Uhr größer als zwei ist oder dass in einem Zustand nur für maximal fünf Zeiteinheiten verweilt werden kann.

Dementsprechend ist die Sprache eines Zeitautomaten eine Menge von Wörtern mit Zeitstempeln. Ein Automat für die Schaltvorgänge einer Ampel könnte beispielsweise das Wort $(\text{grün}, 0) \cdot (\text{gelb}, 15) \cdot (\text{rot}, 17) \cdot (\text{gelb-rot}, 30) \cdot (\text{grün}, 31)$ akzeptieren.

Zeitautomaten mit Kosten sind wiederum eine Erweiterung von Zeitautomaten und können im Laufe der Zeit eine beliebige Ressource verbrauchen – in diesem Fall elektrische Energie. Der Verbrauch kann sowohl von Transitionen als auch von Zuständen abhängen. Für ersteres werden Transitionen um Energie-Angaben erweitert, zweiteres wird durch Leistungsangaben an Zuständen umgesetzt.

In der Literatur gibt es eine Vielzahl von Zeitautomaten- und PTA-Varianten, die sich unter anderem in der Definition der Wörter (Zeitpunkte oder Zeitintervalle) und der Behandlung von Kosten (teils nur für Zustände, teils auch für Transitionen) unterscheiden. Aus diesem Grund werden PTAs hier nicht formal definiert, sondern lediglich beispielhaft erklärt.

Abbildung 2.1 zeigt einen PTA für ein fiktives Funkmodul mit den Zuständen *off*, *idle*, *transmit* und *listen/receive*. Jedem Zustand ist ein Leistungsbedarf zugeordnet, so dass zusammen mit der in ihm verbrachten Zeit die verbrauchte Energie berechnet werden kann. Transitionen sind mit der bei ihrer Durchführung verbrauchten Energie und ihrer Dauer annotiert, wobei die Dauer der Sende-Transition von der Paketlänge pl abhängt.

Der Sendezustand ist zudem zeitlich begrenzt: Sobald das Paket übertragen wurde, wechselt er automatisch zurück in idle. Dazu wird bei Ausführung von *send* eine Uhr zurückgesetzt und die Transition *done* ausgeführt, sobald die konstante Vorbereitungszeit und die von der Paketlänge abhängige Sendezeit verstrichen sind.

Hier ist auch erkennbar, wie sich ein Makromodell für einen zugehörigen Treiber synthetisieren lässt. Ausgehend vom Zustand idle ergibt sich beispielsweise für die Treiberfunktion `int send(char* packet, int packet_length)` die folgende Berechnungsvorschrift.

$$energy\_send(pl) = pl \cdot 4\,\mu\text{J} + (3{,}8\,\text{ms} + pl \cdot 160\,\mu\text{s}) \cdot 122\,\text{mW} = 463{,}6\,\mu\text{J} + pl \cdot 23{,}52\,\mu\text{J}$$

Auf diese Weise können Energiebedarf und Zeitverhalten sowohl offline als auch online modelliert werden. Statische Modelleigenschaften können direkt aus Messdaten bestimmt werden, während für die parameterabhängigen Teilformeln eine Regressionsanalyse notwendig ist. Diese wird im folgenden Abschnitt erklärt.

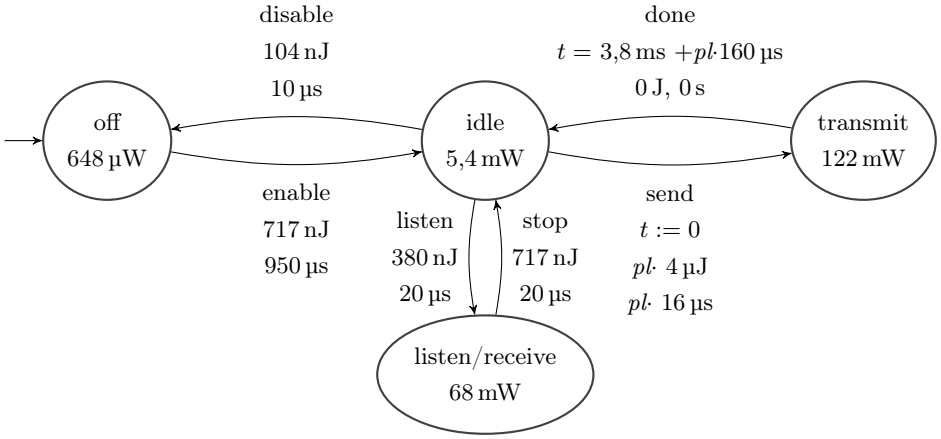

Abbildung 2.1: Ein vereinfachter PTA für ein Funkmodul. Der Parameter *pl* beschreibt die Länge des derzeit übertragenen Pakets.

2.2.3 Regressionsanalyse

In der Praxis gibt es häufig den Fall, dass bestimmte Beobachtungen in Abhängigkeit von Parametern erklärt werden sollen – beispielsweise die Übertragungsdauer des obigen Funkmoduls in Abhängigkeit von der Paketlänge oder die Leistung eines Gesamtsystems in Abhängigkeit von Displayhelligkeit und prozentualer CPU-Last.

Die Art des Zusammenhangs ist dabei meist bereits bekannt oder zumindest schätzbar. So beeinflusst die Sendeleistung die Leistungsaufnahme eines Funkmoduls meist quadratisch oder exponentiell, während der Einfluss der Paketlänge auf die Sendedauer oder der von Helligkeit und CPU-Last auf die Leistung meist linear ist. Die genaue Berechnungsvorschrift, also ob z.B. eine lineare Abhängigkeit nun durch $f(x) = x$, $f(x) = x + 5$, $f(x) = 2x - \frac{1}{3}$ oder andere Zahlenwerte beschrieben wird, ist aber unbekannt.

An dieser Stelle greift die Regressionsanalyse ein. Sie erlaubt, anhand von Messdaten und eines vorgegebenen Modells wie $f(x) = ax + b$ eine Funktion zur Beschreibung der Messwerte zu bestimmen – d.h. Werte für die *Regressionsparameter* a und b, so dass $f(x) = ax+b$ die vorliegenden Daten möglichst gut beschreibt. Die Bestimmung dieser Werte wird als Regressionsanalyse oder auch als *Fitting* der Funktion $f(x)$ bezeichnet.

Hierzu muss zunächst definiert werden, was eigentlich eine „gute“ Funktion ausmacht. Im Rahmen der Regressionsanalyse wird dazu häufig die *Residuenquadratsumme* (kurz *SSR* für *Sum of Squared Residuals*) als Fehlermaß verwendet. Diese beschreibt für Messwerte $X = (X_1, \ldots, X_n)$ und zugehörige Parameterwerte $p = (p_1, \ldots, p_n)$ die Summe der quadratischen Abweichungen zwischen Modellwert $f(p_i)$ und Messwert X_i.

$$SSR(p, X) = \sum_{i=1}^{n} (f(p_i) - X_i)^2 \tag{2.4}$$

Ein ebenfalls verbreitetes Fehlermaß ist der mittlere absolute Fehler (kurz *MAE* für *Mean Absolute Error*), der die mittlere absolute Abweichung zwischen Messwerten und Modellwerten beschreibt.

$$MAE(p, X) = \frac{1}{n} \sum_{i=1}^{n} |f(p_i) - X_i| \tag{2.5}$$

Ziel der Regressionsanalyse ist, das Fehlermaß zu minimieren. Abhängig von der Struktur der Funktion und dem gewählten Fehlermaß existiert eine Vielzahl von Verfahren dazu. So können die Regressionsparameter für eine minimale Residuenquadratsumme bei Verwendung linearer Modelle direkt durch zweifache Ableitung der Residuenquadratsumme bestimmt werden, während zur Minimierung des mittleren absoluten Fehlers oder bei nichtlinearen Modellen aufwändigere Verfahren notwendig sind.

Tatsächlich ist im allgemeinen Fall oft noch nicht einmal garantiert, dass eine eindeutige optimale Lösung existiert. An dieser Stelle werden iterative Verfahren verwendet, die die Regressionsparameter ausgehend von einem Startwert solange verändern, bis ein Abbruchkriterium erreicht wurde.

Als Startwert kann eine Schätzung oder eine Menge von Zufallszahlen genutzt werden; als Abbruchkriterien kommen unter anderem die Gesamtzahl an Iterationen, die relative Änderung des Fehlermaßes im Vergleich zur vorherigen Iteration und die relative Änderung der Regressionsparameter seit der letzten Iteration zum Einsatz. Zudem werden in jeder Iteration nur Veränderungen zugelassen, die das Fehlermaß verringern.

Sofern der vorgegebene Funktionstyp ein brauchbares Modell für das zu beschreibende Verhalten ist, lassen sich mit solchen Verfahren und der Residuenquadratsumme als Fehlermaß verwertbare Modelle erzeugen. Aus diesem Grund setzen viele Statistik-Softwarepakete zur Regressionsanalyse standardmäßig diese Methode (oder eine Variation davon) ein. Auch in dieser Arbeit wird zur Beschreibung von Parameter-Abhängigkeiten nichtlineare Regression mit der Residuenquadratsumme als Fehlerfunktion benutzt. Da sie lediglich als Hilfsmittel dient, wird hier auf eine detaillierte Beschreibung verzichtet.

Zwei Beispiele für mit dieser Methode erzeugte Regressionsmodelle zeigt Abbildung 2.2. Hier sind einige Datenpunkte zur Beschreibung der Leistung abhängig von einem nicht näher benannten Parameter vorgegeben, anhand derer die Regressionsparameter einer linearen und einer quadratischen Funktion optimiert werden. Bei rein visueller Betrachtung ist bereits erkennbar, dass die lineare Funktion augenscheinlich eine bessere Beschreibung der Messdaten erlaubt. Dies zeigt auch die Residuenquadratsumme: Bei der quadratischen Funktion beträgt sie 41, bei der linearen hingegen lediglich 6.

In der Praxis gibt es eine Vielzahl weiterer Regressionsmethoden und Gütemaße, die in dieser Arbeit nicht verwendet werden. Für eine Übersicht darüber und Hintergründe zu den hier vorgestellten Aspekten sei auf [FKL07] verwiesen.

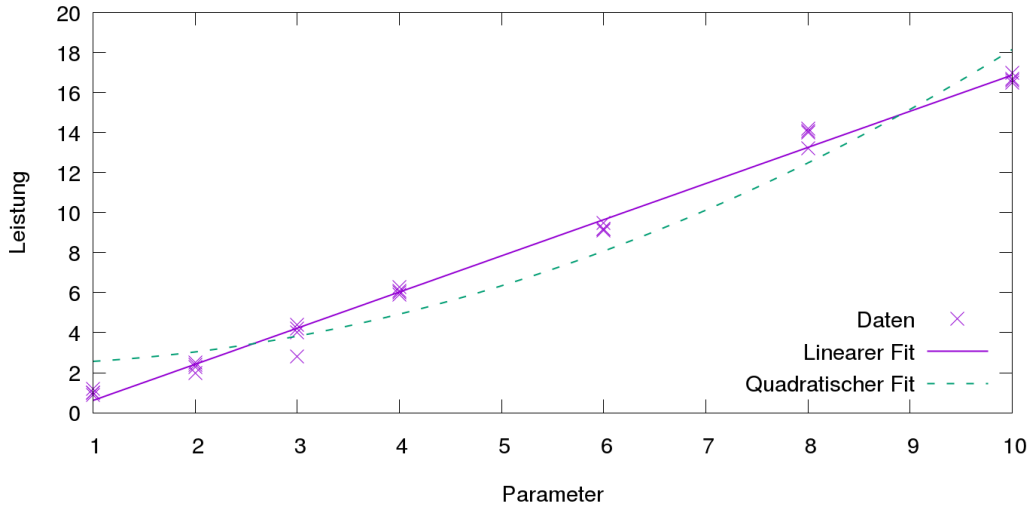

Abbildung 2.2: Fit der Funktionen $f(x) = ax + b$ und $g(x) = cx^2 + d$ mit den Regressionsergebnissen $a = 1,81$, $b = -1,19$, $c = 0,16$ und $d = 2,41$ auf Beispieldaten.

2.3 MSP430FR5969

Der MSP430 ist ein von Texas Instruments entwickelter 16 Bit-Mikrocontroller mit RISC-Architektur. Er verfügt über eine Vielzahl von Ein- und Ausgabepins und Kommunikationsmodulen, ein flexibles Taktsystem und eine klassische Von-Neumann-Architektur. Zudem zeichnet er sich durch sparsame Arbeits- und Tiefschlafmodi aus, weshalb er auch als Ultra Low Power CPU bezeichnet wird.

Das hier verwendete Modell MSP430FR5969 ist mit 2 kB SRAM als Daten- und 64 kB FRAM als kombinierter Daten- und Programmspeicher ausgestattet. Bei FRAM handelt es sich um einen nichtflüchtigen Speicher, welcher beliebig in Programm- und Arbeitsspeicher aufgeteilt werden kann. Hier wird er ausschließlich als Programmspeicher verwendet.

Im Folgenden werden die für diese Arbeit relevanten MSP430-Komponenten näher erläutert. Da der Energiebedarf des MSP430 nicht vermessen wird, beschränken sich diese auf das Taktsystem und die zur Kommunikation mit Peripheriegeräten verfügbaren Schnittstellen.

Taktsystem

Das Taktsystem des MSP430 ist in verschiedene Taktsignale und -quellen aufgeteilt und kann für einzelne CPU-Komponenten individuell konfiguriert werden.

Es stehen fünf mögliche Taktquellen zur Verfügung. Externe Taktgeber im unteren kHz-Bereich können an *LFXT* angeschlossen werden, Taktgeber im unteren MHz-Bereich an *HFXT*. Zusätzlich sind der stufenweise zwischen 1 und 24 MHz konfigurierbare *DCO* (Digitally Controlled Oscillator), der sehr sparsame *VLO* (Very-low-power Low-frequency

Taktquelle	Frequenz	Anmerkungen
LFXT	10 bis 50 kHz	Hier: 32,768 kHz ±25 ppm
HFXT	4 bis 24 MHz	Hier nicht verfügbar
DCO	1 bis 24 MHz	±3,5 %
VLO	9,4 kHz	±35 %
MODOSC	4,8 MHz	±10 %

Tabelle 2.1: Taktsystem des MSP430FR5969.

Oscillator) mit etwa 10 kHz Taktfrequenz und der *MODOSC* (Module Oscillator) mit etwa 5 MHz als interne Taktquellen vorhanden.

Die integrierten Taktquellen haben den Vorteil, dass auch ohne externe Beschaltung hohe Frequenzen erreicht werden können, sind aber mit typischen Fehlern zwischen 3,5 und 35 % ungenau. Der in der hier vorliegenden MSP430-Evaluationsplatine an LFXT angeschlossene Quarz ist mit einem Fehler unterhalb von 25 ppm zwar präzise, aber langsam: Er liefert lediglich einen Puls pro 30,52 µs. Eine detaillierte Übersicht über Frequenzbereiche und erwartete Fehler findet sich in Tabelle 2.1.

Zur Taktung der Systemkomponenten werden diese Taktquellen mit optionalen Frequenzteilern auf die vier Signale *ACLK* (Auxiliary Clock), *MCLK* und *SMCLK* (Master Clock bzw. Sub-system Master Clock), *MODCLK* (Module Clock) und *VLOCLK* (VLO Clock) aufgeteilt. Die Aufteilung ist genau so wie das von einzelnen Teilsystemen verwendete Taktsignal konfigurierbar. Die CPU ist allerdings fest mit MCLK verbunden und unterstützt nur Frequenzen bis 16 MHz.

Schnittstellen

Der MSP430 verfügt über 40 digitale Anschlüsse, die in fünf *Ports* mit je acht *Pins* gruppiert sind. Jeder Pin kann entweder als Ausgang oder als Eingang mit optionalem Pull-Up- oder Pull-Down-Widerstand betrieben werden. Zudem kann jeder Eingang entweder auf Basis des anliegenden Logikpegels oder anhand einer Flanke Unterbrechungen auslösen.

Zusätzlich werden vom MSP430 spezifische Kommunikationsprotokolle unterstützt, welche mit ausgewählten Pins nutzbar sind. Hierzu verfügt er über die Hardwaremodule *eUSCI_A0*, *eUSCI_A1* und *eUSCI_B0*, welche individuell auf spezifische Protokolle eingestellt werden können. Dies sind unter anderem die seriellen Protokolle *UART*, *SPI* und I^2C. Wird ein Pin nicht mit einem spezifischen Protokoll, sondern für herkömmliche Ein-/Ausgabefunktionen genutzt, wird dies auch als *GPIO* (für *General Purpose Input/Output*) bezeichnet.

UART Ein UART (*Universal Asynchronous Receiver Transmitter*) ist eine bidirektionale Kommunikationsschnittstelle mit zwei voneinander unabhängigen Signalleitungen: *Trans-*

mit bzw. TX für Übertragungen von der CPU zum Kommunikationspartner und *Receive* bzw. RX für die Gegenrichtung. Diese können unabhängig voneinander stattfinden, es wird also Vollduplexbetrieb unterstützt. Die Datenrate kann beim MSP430 bis zu 115200 Baud (etwa 115 kbit/s) betragen und muss beiden Kommunikationspartnern bekannt sein, da kein Taktsignal übertragen wird. Datenübertragungen sind Byte-orientiert und unterstützen beliebige Nutzdaten und Anwendungsprotokolle.

SPI Das *Serial Peripheral Interface* SPI verfügt ebenfalls über eine Leitung pro Richtung, die hier MOSI (*Master Out, Slave In*) für von der CPU ausgehende Daten bzw. MISO (*Master In, Slave Out*) für eingehende Daten genannt werden. Zusätzlich stellt die CPU über SCK (*Serial Clock*) einen Takt zur Verfügung, der für beide Richtungen genutzt wird. Daten können also nur übertragen werden, solange die CPU einen Takt vorgibt. Beim MSP430 beträgt dieser maximal 24 MHz.

Datenübertragungen sind Bit-orientiert und häufig von einem zusätzlichen CS-Signal (*Chip Select*) abhängig, welches eine laufende Datenübertragung und ggf. weitere Aktivitäten signalisiert. Das Zusammenspiel aus CS- und SPI-Leitungen ist genau so wie die genutzten Logikpegel peripherieabhängig. Der MSP430 unterstützt eine Vielzahl der möglichen Variationen, überträgt jedoch nur ganze Bytes.

I²C Die *Inter-Integrated Circuits*-Schnittstelle I²C verwendet lediglich zwei Signalleitungen: SDA (*Serial Data*) und SCL (*Serial Clock*). Im Gegensatz zu UART und SPI ist sie zum Betrieb mehrerer Peripheriegeräte (*Slaves*) an einer Datenleitung vorgesehen, die aber nicht untereinander, sondern nur mit einem zentralen *Master* kommunizieren können. Dementsprechend hat jeder Slave eine eindeutige 7 Bit-Adresse und reagiert nur auf an ihn gerichtete Befehle.

Die Datenübertragung ist Byte-orientiert und muss ein spezifisches Protokoll einhalten. Jede Übertragung beginnt mit einem Startsignal, einer Slave-Adresse und einem Schreib/Lese-Bit. Anschließend werden entweder vom Master Daten an den Slave übertragen, oder der Slave schickt (zu dem vom Master vorgegebenen Taktsignal auf SCL) selbst Daten an den Master. Nach jedem Byte ist ein Bestätigungs-Zeitfenster vorgesehen, so dass der Sender feststellen kann, ob ein Gerät mit der gewünschten Adresse existiert und wie viele seiner Daten erfolgreich verarbeitet wurden.

Zur Vermeidung von Kurzschlüssen durch widersprüchliche Signale sind SDA- und SCL-Anschlüsse als offene Kollektoren ausgelegt. Im Ruhezustand und bei Übertragung einer logischen Eins wird der Signalpegel der Leitung nicht beeinflusst, lediglich bei Takt-Generierung und dem Senden einer Null oder eines Bestätigungsbits wird durchgeschaltet und eine Null übertragen.

Um einen eindeutigen Eins-Pegel zu erhalten, sind Pull-Up-Widerstände an SDA und SCL notwendig. Diese betragen üblicherweise wenige kΩ. Die Taktfrequenz liegt im Nor-

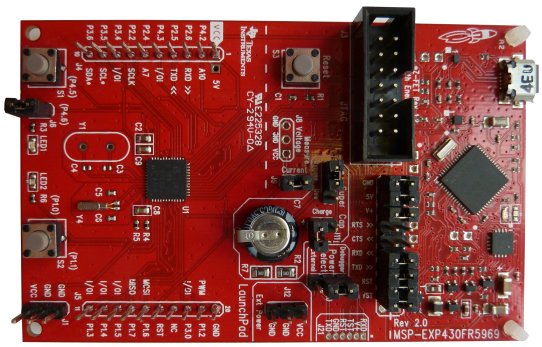

Abbildung 2.3: Das MSP430 Launchpad mit MSP430FR5969-CPU (links) und Programmierschnittstelle (rechts).

malfall bei bis zu 100 kHz, die Datenrate kann entsprechend bis knapp unter 100 kbit/s erreichen. Es gibt allerdings auch I^2C-Varianten mit höherer Datenrate und mit längeren Adressen. Der MSP430 unterstützt wiederum viele davon.

MSP430 Launchpad

Zur einfachen Programmierung der CPU und zur schnellen, nichtpermanenten Anbindung von Peripherie dient die in Abbildung 2.3 dargestellte Evaluationsplattform MSP-EXP-430FR5969, welche ebenfalls von Texas Instruments vertrieben wird. Im Folgenden wird sie kurz als *Launchpad* oder *MSP430 Launchpad* bezeichnet. Diese Plattform besteht aus einer MSP430FR5969-CPU mit leicht zugänglichen GPIO-Pins und einer Programmier- und Debugschnittstelle mit USB-Anschluss.

Der CPU-Teil enthält Stiftleisten, über welche unter anderem eUSCI_B0 (SPI und I^2C) und eUSCI_A1 (SPI und UART) zugänglich sind. Zusätzlich ist ein Uhrenquarz mit einer Frequenz von 32,768 kHz aufgelötet, so dass der präzise LFXT-Taktgeber nutzbar ist. Weiterhin sind ein Reset-Taster, zwei Taster für Nutzereingaben, ein kleiner Akku und zwei LEDs vorhanden.

Mit Hilfe der Debugschnittstelle kann die CPU programmiert, zurückgesetzt und zur Laufzeit analysiert werden. Ebenso werden hier die UART-Leitungen des eUSCI_A0 angebunden, so dass serielle Kommunikation zwischen MSP430-CPU und einem Computer ohne weitere Hilfsmittel möglich ist.

2.4 Kratos

Kratos ist ein an der TU Dortmund entwickeltes Betriebssystem für eingebettete Systeme. Es verfügt über präemptives Multitasking, grundlegende Funktionen wie Semaphoren und zeitbasiertes Warten, eine Unterbrechungsbehandlung und eine grobe Trennung zwischen Treiber- und Anwendungsschicht. Zudem ist es auf individuelle Anforderungen hin

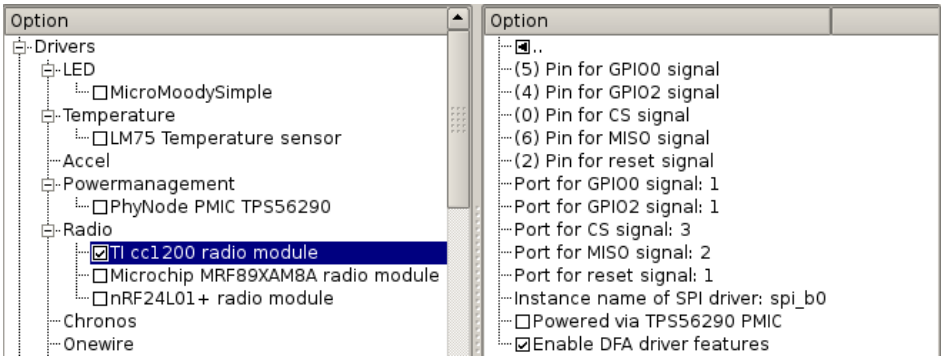

Abbildung 2.4: Ausschnitt aus dem Treiberkonfigurationsmenü von Kratos.

maßschneiderbar, d.h. Anpassungen an Betriebssystemkern und Treiberschicht können mit geringem Aufwand vorgenommen werden. Dies wird auch im Namen ausgedrückt, welcher als rekursives Akronym *Kratos is a Resource-Aware Tailored Operating System* bedeutet. Zur Entwicklung kommt die Programmiersprache AspectC++ zum Einsatz.

Das Betriebssystem unterscheidet zwischen generischen und architekturspezifischen Komponenten, zweitere liegen derzeit für die Architekturen x86, ARM (Raspberry Pi) und MSP430 vor. Mit einem Konfigurationssystem können die Architektur ausgewählt und einzelne Gerätetreiber und Anwendungen konfiguriert und an- oder abgewählt werden. Zudem ist die feingranulare Konfiguration diverser Betriebssystemkomponenten, wie z.B. des Schedulerverhaltens, des Taktsystems oder der Unterbrechungsbehandlung möglich.

Im Folgenden wird auf Konfigurationssystem, AspectC++, Unterbrechungsbehandlung und Treiber- und Anwendungsschicht näher eingegangen.

Konfigurationssystem

Zur Konfiguration von Kratos dient das auch im Linuxkernel verwendete Werkzeug *kconfig*. Dieses verwaltet einen Baum von Konfigurationseinträgen (*Features*) mit optionalen Parametern und zugeordneten Dateinamen. Nach erfolgter Konfiguration wird für den Compiler eine Dateiliste aller verwendeten Features generiert, so dass Quelltextausschnitte für nicht aktivierte Features beim Bauen des Betriebssystems auch nicht berücksichtigt werden. Zudem erzeugt kconfig die Datei `config.h`, welche Präprozessorvariablen für jedes aktivierte Feature und jeden eingestellten Parameter definiert und so die Nutzung von eingestellten Parameterwerten im Quelltext erlaubt.

kconfig erwartet eine Konfigurationsdatei, die alle Features und Parameter sowie ihre Struktur und eventuelle Abhängigkeiten auflistet. Bei Kratos wird diese Datei nicht von Hand gepflegt, sondern aus einzelnen `.feature`-Dateien generiert. Somit kann jede Systemkomponente (z.B. eine Anwendung oder ein Gerätetreiber) ihre eigene Konfigurationsdatei pflegen. Abbildung 2.4 zeigt einen Ausschnitt aus dem Konfigurationsmenü einer kconfig-Oberfläche.

```
aspect SomeDriverLogging {
    advice execution("% SomeDriver::%(...)") : before() {
        uart << "Starting " << JoinPoint::signature() << endl;
    }
};
```

Listing 2.1: Ein Logging-Aspekt. Vor jedem Aufruf einer Funktion von *SomeDriver* wird die Signatur der Funktion per UART ausgegeben.

AspectC++

Die Programmiersprache AspectC++ unterstützt das Konfigurationssystem. Sie erlaubt die Definition von *Aspekten*, die zur Übersetzungszeit u.a. den Kontrollfluss von Funktionen und Funktionsaufrufen verändern können [SG16].

Durch einem Feature zugeordnete Aspektheader können bei dessen Auswahl Veränderungen an anderen Betriebssystemkomponenten vorgenommen werden, ohne dass diese von den Komponenten explizit unterstützt werden – die Umsetzung querschneidender Belange ist also leicht möglich. Dieses Entwicklungskonzept wird auch *aspektorientierte Entwicklung* genannt.

Änderungen am Kontrollfluss werden von *Pointcuts* vorgenommen. Diese können beispielsweise jeden Aufruf einer Gruppe von Funktionen aus einer bestimmten Klasse abfangen und davor, danach oder auch stattdessen im Aspektheader definierten Quelltext ausführen. Die Zuordnung von Quelltextausschnitten zu bestehenden Funktionen und Variablen erfolgt durch *Advice*.

Ein Beispiel für einen Aspekt findet sich in Listing 2.1. Dieser definiert eine Log-Ausgabe, welche vor jedem Aufruf einer beliebigen Funktion x aus der Klasse *SomeDriver* den Text „Starting x“ auf dem UART ausgibt. Änderungen an der Klasse SomeDriver sind dazu nicht notwendig.

In Kratos werden Aspekte routinemäßig sowohl zur Initialisierung und Anpassung von Treibern und Kernkomponenten des Betriebssystems als auch zur Aktivierung von Anwendungen verwendet.

Treiberschicht

Kratos verfügt über zwei verschiedene Treiberschichten: eine architekturspezifische, die Treiber für die CPU-Komponenten wie SPI, UART und I^2C definiert, sowie eine architekturunabhängige, die Treiber für Peripheriegeräte enthält und meist SPI- oder I^2C-Treiber zur Kommunikation mit diesen nutzt. Die Trennung zwischen den Schichten erlaubt die Verwendung eines Peripherietreibers auf verschiedenen Architekturen, solange jede Architektur über einen kompatiblen Treiber für die verwendete Kommunikationsschnittstelle verfügt.

```
aspect ActivateCC1200IRQ {
  advice execution("void␣ initialize_devices ()") : after() {
    radio.plugin();
  }
};
```

Listing 2.2: Initialisierung des globalen CC1200-Treiberobjekts mit dem Namen *radio*.

Treiber werden als C++-Klassen implementiert. Jeder Treiber besteht dementsprechend aus einer Header- und mindestens einer C++-Datei. In der Regel legt ein Treiber zudem ein globales Objekt seiner Klasse an, welches von anderen Anwendungen genutzt werden kann – beispielsweise ein globales Objekt *radio* der Klasse *CC1200*, damit Anwendungen ein Funkmodul nutzen können. Dieses wird in der C++-Datei erzeugt und im zugehörigen Header per *extern*-Deklaration für andere Übersetzungseinheiten nutzbar gemacht.

Eine Feature-Datei erlaubt die An- oder Abwahl und Konfiguration des Treibers. Konfigurationsoptionen können beispielsweise die Anschlussnummern für GPIO-Pins, der Name des verwendeten SPI- oder I^2C-Treiberobjekts oder die Größe eines Displays sein.

Falls vor Nutzung des Treibers spezielle Initialisierungsroutinen auszuführen sind, werden diese mit Hilfe eines Aspektheaders eingebunden. Dieser webt die Routinen nach dem Aufruf der vom Betriebssystem bereitgestellten *initialize_devices*-Funktion ein, so dass der Treiber bei Ausführung der ersten Anwendung bereits nutzbar ist. Im Gegensatz zu einer Initialisierung im Klassenkonstruktor findet die Initialisierung hier zu einem wohldefinierten Zeitpunkt statt, an dem grundlegende Systemeinstellungen wie Taktrate und Initialisierung der CPU bereits vorgenommen wurden.

Ein Beispiel für eine solche Initialisierung findet sich in Listing 2.2. Hier wird der Treiber für einen Funkchip bei der Unterbrechungsbehandlungsschicht von Kratos angemeldet, so dass er auf vom Funkmodul ausgelöste Interrupts reagieren kann.

Unterbrechungsbehandlung

Die Unterbrechungsbehandlung erfolgt bei Kratos zweistufig nach dem Pro-/Epilogmodell. Dazu wird das Betriebssystem zur Laufzeit in drei Ebenen geteilt: die *Anwendungsebene* für unkritische Anwendungen und Treiberaufrufe, die *Prologebene* zur sofortigen Behandlung von Unterbrechungen und die *Epilogebene* zur verzögerten Ausführung zusätzlicher Unterbrechungsbehandlungsroutinen sowie für geschützte Systemaufrufe. Dabei hat die Prologebene die höchste und die Anwendungsebene die niedrigste Priorität.

Die Ebenen interagieren wie folgt miteinander.

- Elemente der Anwendungsebene können durch Pro- und Epiloge unterbrochen und auch (vom Scheduler) durch andere Anwendungen verdrängt werden.

- Epiloge können durch Prologe unterbrochen, jedoch nicht durch andere Epiloge verdrängt werden. Stattdessen werden Epiloge seriell abgearbeitet.
- Prologe sind weder unterbrech- noch verdrängbar.

Ziel dieser Aufteilung ist die schnelle Behandlung von Unterbrechungen bei möglichst kurzen Zeiträumen mit gesperrten Unterbrechungen. Jedem Treiber mit Unterbrechungsunterstützung sind dazu eine Prolog- und eine optionale Epilogfunktion zugeordnet. Der Prolog dient zur sofortigen Ausführung zeitkritischer Aktionen nach einer Unterbrechung, wie beispielsweise dem Auslesen eines Statusregisters, während der Epilog für weniger kritische zusätzliche Arbeit genutzt wird und vom Prolog explizit angefordert werden muss. Im Epilog können beispielsweise Daten an die Anwendungsebene weitergegeben oder ein Kontextwechsel durchgeführt werden.

Bei der Anpassung von Treiberfunktionen zur Protokollierung von Energiedaten sollten nur Funktionen der Anwendungs- und Epilogebene, nicht aber solche der Prologebene berücksichtigt werden. Andernfalls würde die u.U. lange Wartezeit auf Zeit- und Energieberechnungen das Systemverhalten beeinträchtigen.

Anwendungsschicht

Ähnlich wie bei der Treiberschicht werden auch Anwendungen als einzelne Klassen implementiert. Jede Anwendung erbt zudem von der abstrakten Klasse *Thread* und implementiert dessen *action*-Methode. Diese Methode dient als Einstiegspunkt für den Kontextwechsel: Der erste Ausführungszeitraum einer Anwendungen beginnt immer in *action*. Sofern sich eine Anwendung nicht mit Hilfe des Schedulers beendet, darf sie aus dieser Funktion nicht zurückkehren.

Anwendungen haben Zugriff auf alle globalen Betriebssystemobjekte, d.h. insbesondere alle einkompilierten Treiber. Zusätzlich können sie sich mit Hilfe von Semaphoren mit anderen Anwendungen synchronisieren und mit *Buzzern* für einen einstellbaren Zeitraum schlafen legen.

Per Konvention liegt in Kratos jede Anwendung in einem separaten Verzeichnis. Dieses enthält eine Headerdatei zur Definition der Klasse, eine C++-Datei zur Erzeugung des Anwendungsobjekts und zur Implementierung der action-Methode, ein Feature zur Aktivierung/Deaktivierung und Konfiguration und einen Aspektheader zum Anmelden der Anwendung beim Scheduler. Die Erzeugung des Objekts beinhaltet auch die Erstellung des Anwendungsstacks. Die Anmeldung beim Scheduler erfolgt ähnlich wie die Initialisierung von Treibern durch einen Aspekt, der den entsprechenden Aufruf per Advice in die Betriebssystemfunktion *ready_threads* einbettet.

2.5 DFA-Treiber

Bei den DFA-Treibern handelt es sich um ein Konzept für energiegewahre Treiber auf Automatenbasis für Kratos [Fal14]. Ein Teil davon ist in der vorliegenden Kratosversion implementiert und soll die Entwicklung energiegewahrer Treiber mit Datenstrukturen zur Darstellung von Zuständen und Transitionen und Buchführung über die verbrauchte Energie sowie Hilfsmitteln zur Erhebung von Messdaten unterstützen. Zudem enthält das Konzept die Definition eines XML-Formats, welches die Zustände, Transitionen und energetischen Eigenschaften eines Geräts definiert und so als maschinenlesbares Gegenstück zum energiegewahren Gerätetreiber dient. Alle dabei verwendeten Leistungs- und Energieangaben sind statisch, parametrisierte Angaben werden nicht unterstützt.

Kern des Modells sind die Klasse *DFA_Driver*, welche als Basisklasse für energiegewahre Treiber Datenstrukturen und Methoden zur Verwaltung von Transitionen und Zuständen bereitstellt, und die Klasse *ElectricityMeter*, welche Buchführung über Zeit- und Energieverbrauch beliebig vieler Treiber erlaubt.

Im Folgenden werden diese beiden Klassen, die zusätzlich vorhandene Unterstützung bei der Modellerstellung und das für die Modelle definierte XML-Format näher erläutert.

Die Basisklasse DFA_Driver

Ein energiegewahrer Treiber auf Automatenbasis muss Kenntnis über den PTA haben, den er modellieren soll. Dies umfasst die Liste aller Zustände mit zugehöriger mittlerer Leistung, die Liste aller Transitionen mit zugehöriger Energie und den aktuellen Zustand. Der Zustand muss zudem bei Transitionen aktualisiert werden.

Die Klasse DFA_Driver enthält dazu die Variable *state* für den aktuellen Automatenzustand, der mit den Methoden *setInternalState* und *getState* gesetzt und abgefragt werden kann. Das Setzen des Zustands ist dabei nur durch die Klasse selbst oder von ihr abgeleitete Klassen möglich. Zudem definiert sie das Array *statepower*, in welches jeder Treiber die Leistungsdaten seiner Zustände einträgt. Die Zustände selbst werden wiederum als *enum* definiert.

Transitionen und Zustands-Aktualisierungen werden von der Methode *passTransition* umgesetzt. Diese erhält ein von *transition_t* abgeleitetes Transitionsobjekt als Argument und bestimmt anhand dessen den neuen Zustand. Sie enthält ebenso die Implementierung der zur Transition zugehörigen Treiberfunktion. Nutzdaten und sonstige Funktionsargumente werden dabei im Transitionsobjekt vermerkt, so dass sämtliche Treiberfunktionen unter passTransition gekapselt sind. Sie werden durch Angabe des zugehörigen Transitionsobjekts aufgerufen.

Schließlich werden noch die Typen *power_uW_t* und *energy_pJ_t* definiert, um bei Berechnungen von Leistung und Energie immer denselben Variablentypen verwenden zu können. Bei der MSP430-Plattform sind dies 32 Bit-Ganzzahlen ohne Vorzeichen.

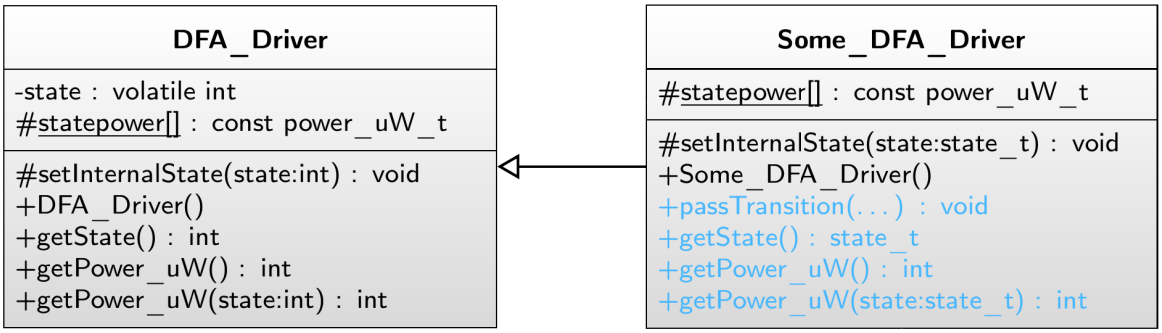

Abbildung 2.5: Die Basisklasse DFA_Driver (links) und ein sie implementierender energiegewahrer Treiber (rechts). Ausschnitt aus [Fal14, Abb. 4.2]

Das DFA-Treiberkonzept sieht vor, dass jeder energiegewahre Treiber von der Klasse DFA_Driver erbt und die Hardware-spezifischen Teile implementiert. Dazu definiert er die Namen der Zustände und die zugehörigen Leistungswerte sowie die verschiedenen Transitionsobjekte mit optionalen Parametern. Ebenso implementiert er die verschiedenen passTransition-Aufrufe als Ersatz für die einzelnen Treiberfunktionen.

Ein Ausschnitt aus der DFA_Driver-Klassendefinition findet sich in Abbildung 2.5. Eine energiegewahre Implementierung einer Kommunikationsschnittstelle würde hier beispielsweise die Zustände UNINITIALIZED, OFF, IDLE und RXTX unterscheiden. Zugehörige Transitionsobjekte sind z.B. trEntry zur Initialisierung der Hardware, trOFF2IDLE zum Einschalten oder trIDLE2RXTX mit den Parametern rxbyte und txbyte zum bidirektionalen Übertragen je eines Bytes.

Stromzähler

Eine Buchführung über die verbrauchte Energie wird vom Treiber selbst nicht umgesetzt. Stattdessen stellt ein globales *ElectricityMeter* eine systemweite Schnittstelle zum Auslesen der seit der letzten Aktivierung vergangenen Zeit und des dabei protokollierten Energiebedarfs zur Verfügung. Treiber können durch einen *ElectricityContractConclusion*-Aspekt um die dazu notwendigen Variablen, Funktionen und Klassenzugehörigkeiten erweitert werden. Diese sind insbesondere eine Variable für den Zeitpunkt des letzten Zustandswechsels, ein „Stromzähler“ für die genutzte Energie und korrespondierende Zugriffsmethoden.

Das Aktualisieren von Zeit- und Energiedaten geschieht per Aspekt vor jeder Ausführung von passTransition. Durch Multiplikation der im aktuellen Automatenzustand benötigten Leistung mit der seit dem letzten Zustandswechsel vergangenen Zeit wird die in diesem Zustand verbrauchte Energie berechnet. Diese wird zusammen mit der im Transitionsobjekt angegebenen Transitionsenergie dem Gesamtenergiebedarf hinzugefügt. Schließlich wird der Zeitpunkt der letzten Zustandsänderung aktualisiert und die eigentliche Transition (d.h. die Kommunikation mit der Hardware) ausgeführt.

Die Zeiterfassung erfolgt mit Hilfe der SMCLK des MSP430. In der Standardkonfiguration von Kratos wird diese mit einem Teiler von 32 aus dem auf 16 MHz eingestellten DCO gespeist, was einen Takt von 500 kHz und somit eine zeitliche Auflösung von 2 µs ergibt.

Unterstützung bei der Modellerstellung

Für eine erfolgreiche Berechnung des Energiebedarfs muss für jeden Zustand die Leistungsaufnahme und für jede Transition der Energiebedarf bekannt sein. Diese Daten werden in der Praxis üblicherweise durch Vermessung der Zustände und Transitionen und Mittelwertbildung der Leistungs- und Energiedaten erhoben. Dabei ist eine Synchronisierung zwischen Treiberverhalten und Messdaten notwendig, um Zustände und Transitionen zuverlässig zu identifizieren.

Die Synchronisierung wird von DFA-Treibern durch die Klasse *PinTrigger* unterstützt. Diese erlaubt die Konfiguration eines CPU-Pins als Trigger, der mit dem digitalen Eingang eines Messgeräts verbunden werden kann. Mit Hilfe eines *AutoTrigger*-Aspekts kann ein energiegewahrer Treiber Beginn und Ende seiner Transitionen durch Flanken oder Pulse auf dem Triggerpin signalisieren, so dass bei einem bekannten Programmablauf die Synchronisierung zwischen Messdaten und Zustands- bzw. Transitionsnamen möglich ist.

Zur DFA-Treiber-Implementierung gehört zudem ein Skript, welches aus vorgegebenen Zuständen und Transitionen ein Testprogramm zum Vermessen der zugehörigen Komponente generiert. Dieses läuft alle Zustände und Transitionen einmal ab und muss lediglich noch um ggf. für einzelne Transitionen notwendige Argumente oder Nutzdaten erweitert werden. Anschließend kann der Anwender manuell ein Auswertungsskript erstellen, welches jede mit diesem Testprogramm aufgezeichnete Messung auslesen und in Leistungs- und Energiewerte für Zustände und Transitionen überführen kann.

Automatendefinition in XML

Zur automatischen Generierung von Treibergerüsten und Unterstützung bei der Anpassung von Treibern ist jedem DFA-Treiber eine XML-Datei zugeordnet, die den Namen der Treiberklasse und das Energiemodell in maschinenlesbarer Form enthält. Sie beschreibt die Namen der Zustände mit zugehörigen Leistungswerten in µW sowie die Transitionen mit zugehörigen Energiewerten in pJ. Ebenfalls wird bei jeder Transition angegeben, ob diese auf der Anwendungsschicht (*user level*) oder der Epilogebene (*epilogue level*) ausgeführt wird.

Transitionen auf Epilogebene enthalten zusätzlich eine Angabe zum *Timeout* in µs. Dieses beschreibt die Wartezeit zwischen Betreten des Ausgangszustands und dem Ausführen der Transition – effektiv also die Wartezeit auf die Unterbrechung, mit der der Ausgangszustand der Transition verlassen und die Transition ausgelöst wird. Andere Zeiten, wie beispielsweise die Dauer von Transitionen, werden nicht modelliert.

```
<?xml version="1.0">
<data>
  <driver name="DFA_SPI">
    <states>
      <state name="STANDBY" power="0"/>
      <state name="TRANSFERRING" power="0"/>
    </states>
    <transitions>
      <transition name="trSTANDBY2TRANSFERRING" energy="0">
        <src>STANDBY</src>
        <dst>TRANSFERRING</dst>
        <level>user</level>
      </transition>
      <transition name="trTRANSFERRING2STANDBY" energy="0">
        <src>TRANSFERRING</src>
        <dst>STANDBY</dst>
        <level>epilogue</level>
        <timeout>8/CONFIG_eUSCI_B0_DFA_SPI_BITRATE</timeout>
      </transition>
    </transitions>
  </driver>
</data>
```

Listing 2.3: XML-Definition der Zustände und Transitionen eines DFA-Treibers für das SPI-Modul des MSP430.

Als Besonderheit können Timeout-Angaben auch von Konfigurationsparametern wie beispielsweise der eingestellten Bitrate abhängen. Eine Abhängigkeit von Laufzeitparametern, wie zum Beispiel der Länge der übertragenen Daten, wird aber auch hier nicht unterstützt.

Ein Beispiel für eine XML-Modelldefinition findet sich in Listing 2.3. Dieses beschreibt einen energiegewahren Treiber für die SPI-Schnittstelle des MSP430, der über die Zustände STANDBY und TRANSFERRING verfügt. Daten werden Byte-weise übertragen, dazu wird für jedes Byte die Transition trSTANDBY2TRANSFERRING ausgeführt. Die Wartezeit zwischen Start der Übertragung und Rückkehr in den STANDBY-Zustand wird im Timeout von trTRANSFERRING2STANDBY beschrieben und hängt von der eingestellten SPI-Bitrate ab.

Da das obige XML-Modell alle von den DFA-Treibern vorgesehenen Aspekte eines Energiemodells enthält, dient es gleichzeitig als Syntaxdefinition des XML-Formats.

2.6 MIMOSA

Das in der Arbeitsgruppe Eingebettete Systemsoftware entwickelte MIMOSA („Messgerät zur integrativen Messung ohne Spannungsabfall“) kann zur Erstellung präziser Energiemodelle genutzt werden [BGS13]. Zur Kommunikation mit dem Messgerät stehen die C++-Bibliothek *libmimosa*, das Kommandozeilenprogramm *MimosaCMD* und die grafische Oberfläche *MimosaGUI* zur Verfügung [Ton14].

Hardware

MIMOSA kann Ströme von bis zu 50 mA mit einer Auflösung von 16 Bit messen. Zudem verfügt es über einen digitalen *Buzzer*-Eingang, welcher eine Synchronisation zwischen Messwerten und Aktionen des vermessenen Geräts erlaubt. Messdaten werden mit einer Frequenz von 100 kHz erhoben, was einer zeitlichen Auflösung von 10 µs entspricht.

Im Gegensatz zu klassischen Multimetern verfügt MIMOSA über einige Besonderheiten. Das Messprinzip beruht zwar wie üblich auf dem an einem Messwiderstand auftretenden Spannungsabfall, eine Messung wird aber nicht durch Anbringen von MIMOSA zwischen einer Spannungsquelle und dem zu untersuchenden Verbraucher durchgeführt. Stattdessen versorgt MIMOSA den Verbraucher selbst und kompensiert dabei den am Messwiderstand auftretenden Spannungsabfall, so dass das Gerät immer eine konstanten Spannung erhält. In dieser Arbeit werden alle Messungen mit einer Versorgungsspannung von 3,6 V durchgeführt, da das MSP430 Launchpad ebenfalls mit dieser Spannung arbeitet.

Zudem werden die Messwerte nicht durch periodische Abtastung, sondern durch Integration des Spannungsabfalls bestimmt. Hierzu verfügt MIMOSA über drei analoge Integratoren auf Basis von Operationsverstärkern und Kondensatoren, die den vollständigen Spannungsverlauf zwischen zwei Messpunkten integrieren. Da der Spannungsverlauf proportional zum Stromverbrauch ist, wird letztendlich das Integral des Stroms bestimmt, was zusammen mit der eingestellten Versorgungsspannung und dem gewählten Shunt die Energie ergibt (vgl. Gleichung 2.3).

Die drei Integratoren werden im Wechsel in drei Zuständen betrieben. Zuerst wird der Spannungsabfall am Shunt integriert, anschließend wird die Energie des vorherigen Zeitfensters ausgelesen und zuletzt wird der Kondensator des Integrators zurückgesetzt, so dass eine neue Integration beginnen kann. Durch einen gestaffelten Betrieb der Integratoren kann so alle 10 µs die in den vorherigen 10 µs verbrauchte Energie ausgelesen werden. Da die Integration komplett analog stattfindet, fließen auch sehr kurze Stromverbrauchsschwankungen in die Energiewerte ein.

Auf Hardwareseite besteht MIMOSA aus einer Shunt- und einer Integratorplatine (siehe Abbildung 2.6). Das zu vermessende Gerät wird an die Shuntplatine angeschlossen und die am Messwiderstand anliegende Spannung über ein Kabel an die Integratorplatine wei-

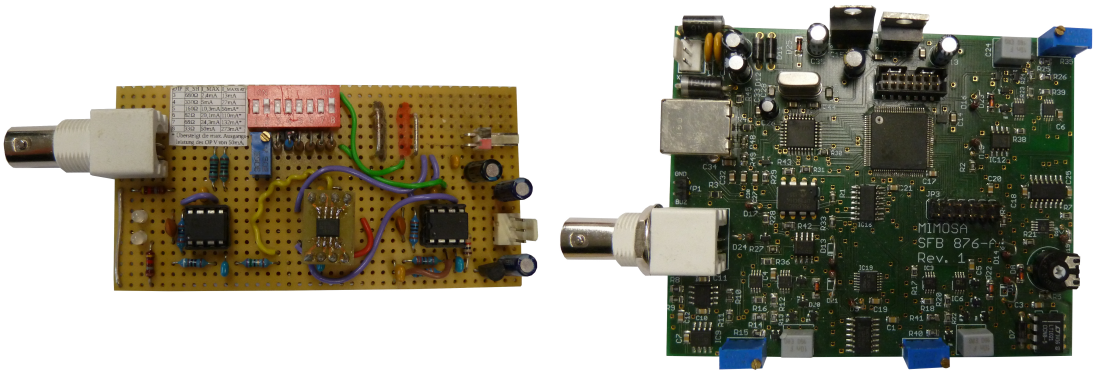

Abbildung 2.6: MIMOSA. Links ist die Shuntplatine mit einstellbarem Messwiderstand und per Potentiometer einstellbarer Spannungsversorgung, rechts die Integratorplatine mit USB-Schnittstelle und einem Kalibrierpotentiometer je Integrator.

Shunt	680 Ω	330 Ω	160 Ω	82 Ω	68 Ω	33 Ω
Messbereich	2,7 mA	5,6 mA	11,5 mA	22,4 mA	27 mA	55,6 mA
Auflösung	41 nA	85 nA	175 nA	342 nA	412 nA	869 nA

Tabelle 2.2: Messbereiche und Auflösungen von MIMOSA.

tergeleitet. Laut MIMOSA-Software darf diese Spannung maximal 1,836 V betragen, da andernfalls der Messbereich überschritten wird.

Der Messbereich und die nominelle Messgenauigkeit hängen vom Shunt ab, welcher vor jeder Messung mit DIP-Schaltern fest eingestellt werden muss. Eine Übersicht über die verfügbaren Widerstände und entsprechende Messbereiche findet sich in Tabelle 2.2.

Software

libmimosa dient zur direkten Kommunikation mit MIMOSA. Hiermit kann eine Messung gestartet, beendet und gespeichert sowie eine bereits durchgeführte Messung geladen werden. Dazu wird das Verzeichnis `/tmp/mimosa` verwendet, welches unter anderem die Datei `mimosa_scale_1.tmp` mit den Rohdaten der aktuellen (oder alternativ der zuletzt geladenen) Messung enthält. Gespeicherte Messungen sind letztendlich nur komprimierte Archive von `/tmp/mimosa` mit der Dateiendung `.mim`.

Messungen werden binär mit vier Byte je Mess-Intervall gespeichert, wobei ein Intervall in `mimosa_scale_1.tmp` eine feste Dauer von 10 µs hat. Vermerkt wird die Nummer des für die Messung verwendeten Integrators, das Buzzersignal und ein zum mittleren Stromfluss im Mess-Intervall proportionaler ADC-Wert. Eine Detailansicht einer solchen Vier-Byte-Folge zeigt Tabelle 2.3.

Bit(s)	31 … 4	3	2	1 … 0
Bedeutung	Messwert	Buzzer	Reserviert	Integratornummer

Tabelle 2.3: Rohdatenformat eines einzelnen MIMOSA-Messpunkts.

Unter Hinzunahme der ADC-Maximalspannung U_{max} und des Shuntwiderstands R berechnet libmimosa wie folgt aus einem Messwert x den mittleren Stromfluss.

$$I = \frac{x}{2^{16} - 1} \cdot \left(\frac{U_{max}}{R} \right)$$

Optional kann dabei anstelle von 65535 durch eine kleinere Zahl geteilt werden, um Messbereichsverschiebungen durch Hardware-Ungenauigkeiten zu kompensieren. Die so bestimmten Messdaten mit Stromwerten können von libmimosa bzw. MimosaCMD im CSV-Format exportiert und von MimosaGUI grafisch dargestellt werden.

MimosaCMD dient lediglich als Kommandozeilenschnittstelle zu libmimosa und bietet keine weiteren Funktionen, während MimosaGUI neben dem Durchführen von Messungen auch die visuelle Inspektion der Messergebnisse erlaubt. Zudem kann anhand von Buzzer- oder Stromverbrauchsflanken halbautomatisch die mittlere Leistung oder der Energiebedarf in einem bestimmten Zeitbereich bestimmt werden.

Auch eine einfache Kalibrierung von MIMOSA wird unterstützt. Hierzu enthält MimosaGUI einen Kalibrierungsbildschirm, bei dem ein Sollstrom vorgegeben und mit den Ist-Werten der einzelnen Integratoren verglichen werden kann. Den Integratoren zugeordnete Präzisionspotentiometer auf der MIMOSA-Platine können dann von Hand eingestellt werden, bis die Differenz zwischen Soll und Ist vernachlässigbar wird. Für den Sonderfall eines Sollstroms von 0 A kann zudem der aufgetretene Messfehler direkt in MimosaGUI eingetragen und an libmimosa weitergereicht werden.

Abbildung 2.7 zeigt die MimosaGUI-Oberfläche bei der Auswertung von Messdaten.

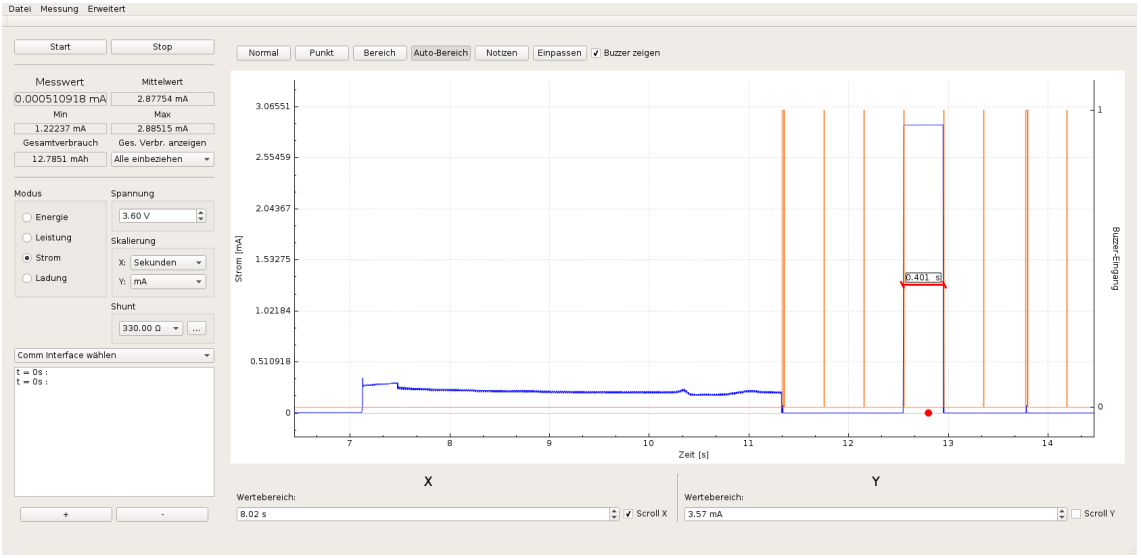

Abbildung 2.7: Darstellung einer Messung mit MimosaGUI. Im linken Teil werden Zeit- und Energiedaten zum rechts ausgewählten Messbereich angezeigt.

Kapitel 3

Verwandte Arbeiten

Sowohl zu Modellen allgemein als auch zur automatischen oder halbautomatischen Modellerstellung gibt es bereits eine Vielzahl an Arbeiten, die allesamt sehr unterschiedliche Ziele verfolgen. Vor der Vorstellung des in dieser Arbeit entworfenen Modellverfeinerungskonzepts lohnt es sich daher, verwandte Literatur aus diesem Themenbereich zu betrachten.

Die Literatur wird dazu in diesem Kapitel auf drei Kategorien aufgeteilt:

- die Entwicklung von zur Modellerstellung geeigneter Messtechnik,
- Modellierungsmethoden und ihre Genauigkeit und
- die (teilweise automatisierte) Datenauswertung und Modellsynthese.

3.1 Messtechnik

Neben zahlreichen ad hoc-Auswertung mit Hilfe von Messwiderständen an digitalen Multimetern und Oszilloskopen gibt es mittlerweile eine zunehmende Anzahl an eigens für die Analyse eingebetteter Systeme entwickelter Mess-Hardware. Der Fokus reicht von der Beobachtung von Ein- und Ausgaben des Systems und bidirektionaler Kommunikation mit dem Messgerät bis hin zur gleichzeitigen Vermessung mehrerer Dutzend Knoten eines Sensornetzes. Ein durchgehendes Thema ist dabei allerdings die Synchronisation zwischen vermessenem System und Messgerät. Diese wird benötigt, um erhobenen Messdaten einzelne Ereignisse oder Systemzustände zuzuordnen.

Eine deutliche Abwendung von klassischer Messtechnik mit Messwiderstand und Analog-Digital-Wandler ist *Energy Bucket* [AH09]. Dieses System verwendet zwei Kondensatoren, die im Wechsel ge- und entladen werden und die zu vermessende Hardware versorgen. Ladespannung und Entladegrenzwert sind konstant, so dass bei jedem Entladevorgang die gleiche Ladung und dank eines vor das Zielsystem geschalteten Spannungswandlers auch die gleiche Energie übertragen wird. Durch Zählen der Lade-Entlade-Zyklen kann somit der Energiebedarf in einem beliebigen Zeitfenster bestimmt werden. Diese Methode umgeht genau so wie MIMOSA die Beschränkungen zeitdiskreter Strommessung und erreicht

einen Messfehler von maximal 2 % für Ströme bis 50 mA. Die zeitliche Auflösung hängt allerdings vom aktuellen Stromverbrauch ab.

Zur Synchronisierung mit dem vermessenen System dienen drei digitale Eingänge, so dass durch Annotationen im Quelltext insgesamt acht Zustände unterschieden werden können. Die Auswertung der Messwerte findet an einem mit dem Energy Bucket verbundenen PC statt und ist für das getestete System nicht sichtbar.

Die Plattform *iWEEP_HW* nutzt klassische Messwiderstände, verfügt aber über mehrere Kanäle [ZO13]. Somit kann der Energiebedarf eines Systems auf verschiedene Komponenten aufgeschlüsselt werden, wofür sonst zeitlich getrennte Messläufe mit aufwändiger Synchronisation und potentiell unterschiedlichen Umgebungseinflüssen notwendig wären. Eine digitale Schnittstelle zum System ist allerdings nicht vorhanden, so dass die Zuordnung von Messdaten zu Hardwarezuständen und -ereignissen von Hand erfolgen muss.

Ein simulationsbasierter Mess-Ansatz wird in [GP11] vorgestellt. Hier wird eine Reihe von drahtlosen Sensoren untersucht, wobei jeder Sensor mit einer eigenen Messplatine ausgestattet ist, die nicht nur seinen Energiebedarf messen, sondern auch Sensorenwerte wie Temperatur oder Luftfeuchtigkeit simulieren kann. Die Sensoren führen dieselbe Software wie auch im realen Einsatz aus und sind untereinander per Funk vernetzt, während die Messplatinen mittels CAN-Bus an einen zentralen PC angebunden werden.

Somit kann das Energieverhalten der Sensoren in diversen Umgebungen gemessen werden. Die aufwändige Erstellung von Testprogrammen entfällt, da die Messplatinen über eine direkte Schnittstelle zu den Sensoren verfügen und der auf den Sensoren laufenden Standardsoftware lediglich verschiedene Messdaten vorgeben.

FlockLab verfügt über eine ähnlich umfangreiche Schnittstelle zwischen Messplatine und Gerät, ist jedoch deutlich flexibler [Lim+13]. Hier stehen sowohl binäre Eingänge als auch Ausgänge zur Verfügung, so dass bidirektionale Kommunikation möglich ist. Zusätzlich können UART-Ausgaben protokolliert und die momentane Temperatur und Luftfeuchtigkeit ausgewertet und z.B. zur Belüftung der Testumgebung bei zu hoher Temperatur genutzt werden. Auch das Zurücksetzen oder Neuprogrammieren einzelner Geräte wird unterstützt

Der Stromfluss wird mit einem Messwiderstand von 150 mΩ in Kombination mit einem Operationsverstärker gemessen. Der Messfehler beträgt im Mittel nur 0,4 %, steigt für Ströme unterhalb von 1 mA aber auf bis zu 10 % an. Abbildung 3.1 zeigt das Konzept dieses Systems.

Auch *Powerbench* ist in der Lage, UART-Ausgaben mitzuschneiden und zeitlich den gemessenen Leistungsdaten zuzuordnen [Har+08]. Der Fokus dieser Plattform liegt auf der Vermessung vernetzter Sensoren des gleichen Typs: Bis zu 24 Sensoren können gleichzeitig vermessen werden. Zudem wird die Messung nicht von einem vollwertigen PC, sondern von einem eingebetteten System durchgeführt, so dass ein kompaktes Gesamtsystem entsteht. Die Auflösung der Messdaten beträgt allerdings nur 200 µs und 30 µA.

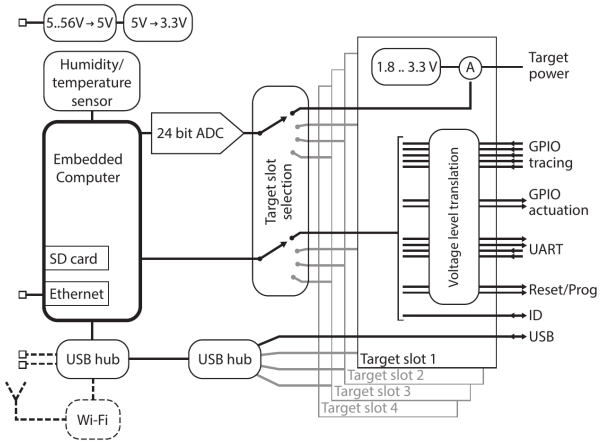

Abbildung 3.1: Architektur einer FlockLab-Messplatine [Lim+13].

Die bereits im vorherigen Kapitel erwähnte MIMOSA-Plattform ist nicht unmittelbar in der Lage, UART-Ausgaben aufzuzeichnen, verfügt dafür aber über integrative Strommessung und die Kompensation des am Messwiderstand auftretenden Spannungsabfalls [BGS13]. Die Auflösung liegt bei 10 µs und 41 bis 869 nA. Es ist allerdings nur ein einziger digitaler Eingang vorhanden, so dass Messungen ihren aktuellen Zustand lediglich binär ausgeben können. Bei komplexen Messverläufen sind entsprechend aufwändige Synchronisierungsroutinen notwendig.

3.2 Modellierung

Eine Kernfrage fast jedes Modellierungskonzepts ist der Detailgrad des Modells. Ein zu abstraktes Modell erlaubt keine genauen Vorhersagen, während ein zu detailliertes für einen praktischen Einsatz zu pflegeintensiv oder bei Verwendung als Onlinemodell in einem eingebetteten Betriebssystem schlicht zu komplex sein kann.

Zu diesem Aspekt wird in [Hur+11] gezeigt, dass zu starke Abstraktion aufgrund von Fehlannahmen über das Hardwareverhalten der Modellgüte abträglich ist. Hier wird das zum Veröffentlichungszeitpunkt verbreitete *Three States Model* untersucht, welches Funkmodule in Sensornetzen auf Automatenbasis beschreibt. Dazu weist es dem Funkchip die drei Zustände *aus*, *senden* und *empfangen* mit jeweils konstanter Leistungsaufnahme zu und modelliert Transitionen zwischen den einzelnen Zuständen ohne Zeit- oder Energiebedarf. Beim Test mit verschiedenen Kommunikationsprotokollen weist es bis zu 21 % Abweichung zwischen modelliertem und tatsächlichem Energiebedarf auf, weshalb die Gründe für diese Ungenauigkeit gesucht werden.

Zunächst wird festgestellt, dass sich Energie- und auch Zeitverhalten verschiedener Sensorknoten trotz identischer Hard- und Software unterscheiden. Als Grund dafür werden

Schwankungen bei der Fertigung der Hardware angenommen. Dennoch zeigen sich auch bei Modellierung nur eines einzelnen Knotens noch Modellfehler von durchschnittlich 3,00 %.

Diese können durch eine Unzulänglichkeit im Modell erklärt werden: Der Wechsel in den Sende- und Empfangszustand findet beim betrachteten Funkmodul nicht sofort statt, sondern im Rahmen eines wenige Millisekunden andauernden Übergangs mit geringem Energiebedarf. Durch Berücksichtigung dieses Übergangszustands im neu erstellten *Three States Model with State Transitions* kann der mittlere Fehler auf 1,13 % reduziert werden; wenn Modellsynthese und -evaluation auf die Nutzung eines festen Funkprotokolls beschränkt werden, sinkt er sogar auf 0,42 % ab. Dies zeigt, dass sowohl detaillierte Messungen als auch eine individuelle Kalibrierung pro Gerät und Analyse der Modellfehler notwendig sind, um ein präzises Modell zu erhalten.

Als nächstes stellt sich die Frage, welche Art von Modell sinnvoll ist. Von den in Abschnitt 2.2 vorgestellten Modellansätzen haben in der aktuellen Literatur nur automatenbasierte und analytische signifikante Verbreitung erfahren.

Analytische Modelle auf Basis von Laufzeitparametern sind insbesondere auf Smartphoneplattformen wie Android beliebt. Hier werden die einzelnen Komponentenzustände inklusive ihres Anteils an der Gesamtlaufzeit bereits vom Betriebssystem bereitgestellt, so dass sie leicht zur Modellierung nutzbar sind. In [Mur+12] wird die Erstellung und Optimierung eines solchen Modells mit Hilfe linearer Regression beschrieben.

Bei zunehmend komplexer Hardware ist Regression allgemein und lineare Regression im Besonderen allerdings nicht immer akkurat [McC+11]. Die Abstraktion weg von dem tatsächlichen Hardwareverhalten hin zu groben Zustandsparametern lässt zwangsläufig einen Teil der Komplexität aus und büßt daher Genauigkeit ein. Auch die fehlende Modellierung von Transitionen ist hier problematisch.

Einen Schritt in Richtung automatenbasierter Modelle mit mehr als nur Leistungsannotationen an Zuständen macht [Zho+11]. Hier werden auch Transitionen zwischen Zuständen berücksichtigt und mit einer festen Dauer versehen. Es wird ihnen aber keine Energie zugeordnet, sondern stattdessen die Energie aus der Transitionsdauer und dem Leistungsmittelwert von Start- und Zielzustand berechnet.

Ein ähnlich detailliertes und explizites Transitionsmodell liefern die bereits in Abschnitt 2.5 vorgestellten DFA-Treiber [Fal14]. Hier werden Transitionen als instantan angenommen, haben aber einen eigenen Energiebedarf, der zum bisherigen Energieverbrauch hinzuaddiert wird. Kurzfristige Verbrauchsspitzen während einer Transition können so modelliert werden. Transitionen mit geringerer mittlerer Leistung als die beteiligten Zustände sind allerdings nicht ohne weiteres möglich. Ein Beispiel hierfür ist der Übergangszustand im Three States Model with State Transitions, bei dem es sich genau genommen um eine Transition (mit verhältnismäßig geringem Energiebedarf) handelt.

Als Brücke zwischen komplexen Offline- und einfachen Online-Modellen sei schließlich [BGS12] genannt. Hier wird beschrieben, wie sich aus einem detaillierten Automa-

tenmodell einfache Onlinedaten für spezifische Nutzungsszenarien berechnen lassen. Dazu wird der Automat für das gewählte Szenario simuliert, die Zeit in den einzelnen Zuständen erfasst und so der durchschnittliche Stromverbrauch bestimmt.

Dies kann sowohl auf Ebene des Gesamtsystems als auch auf Ebene von Funktionsaufrufen geschehen, so dass z.B. für jeden Funktionsaufruf einer Treiberfunktion anhand ihrer Parameter die benötigte Energie bestimmt werden kann. Falls im praktischen Einsatz nur wenige Parameterkombinationen vorkommen, können sogar vorberechnete Energiewerte in einer Tabelle vorgehalten werden. Zwar wird der Energiebedarf von Transitionen hier nicht berücksichtigt, das Konzept lässt sich dahingehend aber leicht anpassen.

3.3 Auswertung

Manuelle Messdatenauswertung ist zeitaufwändig – das Erstellen von Programmen zur automatischen Auswertung allerdings ebenfalls. Abhängig davon, ob nur ein Modell für ein bestimmtes Stück Hardware benötigt wird, oder die Entwicklung eines Mess-Systems zur Vermessung und Modellierung einer Vielzahl von Komponenten im Vordergrund steht, sind in gängiger Literatur daher beide Varianten anzutreffen. Auch Mischkonzepte mit teilautomatisierter Auswertung kommen vor.

Der zur manuellen Modellerstellung notwendige Aufwand wird in [Shn+04] sichtbar. Dort wird die Simulationsplattform *PowerTOSSIM* vorgestellt, welche präzise Angaben zum Energiebedarf von Anwendungen im eingebetteten Betriebssystem TinyOS liefert. Zur Modellierung dient ein Automat pro Hardwarekomponente, wobei die einzelnen Automaten anhand von auf die jeweilige Komponente zugeschnittenen Microbenchmarks mit Leistungswerten versehen werden. Sowohl die Benchmarkerstellung als auch die Auswertung der Ergebnisse erfolgt in Handarbeit.

Dennoch ist dieser Aufwand häufig gerechtfertigt, da die Arbeit mit eingebetteten Systemen ohne akkurate Energiemodelle in vielen Fällen nicht denkbar ist. Besonders gut auf den Punkt bringt dies die in Abschnitt 3.1 vorgestellte Publikation über die Messplattform iWEEP_HW. Nach Erstellung einiger Energiemodelle mit dieser Plattform resümieren die Autoren: „Although real-world hardware measurements is time consuming, complex and repetitive [...] the accurate results from real measurements are worth the efforts and time“ [ZO13].

Bei der Recherche nach automatisierten Mess- und Auswertungsmethoden fällt auf, dass ein großer Anteil an Literatur auf Mobiltelefone und Laptops entfällt. Hierfür gibt es zwei Hauptgründe. Einerseits bietet Smartphone- und Laptop-Hardware oft die Möglichkeit, die aus dem Akku entnommene Ladung mit Bordmitteln zu protokollieren, so dass keine zusätzliche Messtechnik notwendig ist. Andererseits stellt das Betriebssystem zur Laufzeit sogenannte *Performance Counter* mit Angaben über Bildschirmhelligkeit, durchschnittliche Funkaktivität, derzeitigen CPU-Zustand und Ähnliches zur Verfügung. Sofern

diese hinreichend akkurat sind, kann mit ihnen leicht der aktuelle Systemzustand bestimmt werden.

Zusätzlich handelt es bei sich Laptops und insbesondere Smartphones um weitgehend homogene Software-Ökosysteme mit nur wenigen verbreiteten Betriebssystemen. Sie sind daher ein lohnendes Ziel zur Entwicklung von Automatismen, da sich diese leicht auf andere Hardware mit gleichem Betriebssystem übertragen lassen.

Aus dem Smartphonebereich sei zunächst *DevScope* genannt [Jun+12]. Hierbei handelt es sich um ein Softwarewerkzeug, welches ohne externe Hardware Energiemodelle für die einzelnen Komponenten eines Telefons erstellt. Anhand einer vorgegebenen Liste von Hardwarezuständen wird jede Komponente durch ihre Zustände geschickt und dabei der Energiebedarf jedes Zustands ermittelt. Auch variierbare Parameter wie Displayhelligkeit oder Funkdatenrate werden im Modell berücksichtigt.

Die Struktur des analytischen Modells wird vorgegeben und daraus mittels Regressionsanalyse ein Modell für das gerade verwendete Smartphone erstellt. Transitionskosten werden nicht berücksichtigt. Bei Komponenten mit einer großen Anzahl an Parameterwerten, wie einem Display mit 256 Helligkeitsstufen oder einem Funkmodul mit variabler Paketrate, werden nur ausgewählte Stufen vermessen. Auch versteckte Teilzustände, wie kurzzeitige Bereitschaftszustände vor dem endgültigen Ausschalten einer Peripheriekomponente, können unter Umständen erkannt werden. Hier sind jedoch teilweise Heuristiken zur Modellierung notwendig.

PowerBooter und *PowerTutor* gehen ähnlich vor [Zha+10]. Jeder Komponente werden eine oder mehrere Systemvariablen zugeordnet, die wiederum als Parameter in eine lineare Regression fließen. In einigen Fällen beschreiben diese Variablen auch den Zustand (z.B. „an“ / „aus“ für ein GPS-Modul), so dass ein gewisser Bezug zu automatenbasierten Modellen vorhanden ist. Transitionen werden aber auch hier nicht berücksichtigt.

Die Arbeit ist eingeteilt in PowerTutor, welcher Messungen durchführt, und PowerBooter, welcher anhand des erstellten Modells Energieprofiling erlaubt. Das Modell berechnet dabei anhand der aktuellen Systemvariablen die momentane Leistung, so dass die Genauigkeit auch von der Berechnungshäufigkeit abhängt. Die Modellsynthese erfolgt automatisch, es ist aber unklar, ob dies auch auf die Erstellung der zur Datenerhebung benötigten Testprogramme zutrifft.

Einen explizit automatenbasierten Ansatz bietet *eprof* [Pat+11]. Auch hier wird der Energiebedarf von Smartphones modelliert, allerdings durch Nutzung eines DFAs pro Komponente und Zuordnung von Automatentransitionen zu Systemaufrufen.

Hier wird zunächst gezeigt, dass eine rein variablenbasierte Lösung ohne Automatenhintergrund der Komplexität heutiger Smartphonehardware nicht gerecht wird. Module können beispielsweise über sogenannte *Tail States* verfügen, in denen sie nach getaner Arbeit nicht sofort in einen Stromsparzustand verfallen, sondern in Erwartung weiterer

Aktivität für einige Zeit in einem Bereitschaftszustand verbleiben. Solches Verhalten kann nur modelliert werden, wenn einzelne Transitionen berücksichtigt werden.

Bei Transitionen, die über keinen zugeordneten Systemaufruf verfügen, wird lineare Regression zur Modellierung der Wartezeit bis zum Transitionsaufruf genutzt. Dies betrifft beispielsweise das Ende des Übertragungszustands eines Funkchips, welches von der Menge zu übertragender Daten abhängt und nicht explizit aufgerufen wird. Die allgemeine Testprogramm- und Modellerstellung ist jedoch zu großen Teilen Handarbeit. Insbesondere die Annotation der Systemaufrufe zur Aktualisierung der Automaten wird nicht automatisiert.

PowerProf arbeitet ebenfalls auf der Systemaufrufebene, nutzt aber eine vollkommen andere Methode zur Energiemodellierung [KB12]. Jedem Funktionsaufruf werden genau vier energetische Zustände zugeordnet: zwei Zustände zwischen Beginn und Ende des Funktionsaufrufs und zwei weitere Zustände zwischen Ende des Aufrufs und Zurückfallen des Stromverbrauchs auf das Hintergrundniveau.

So können Tail States und auch Komponenten mit verzögertem Einschaltvorgang modelliert werden. Messdaten werden ohne externe Hardware mit Hilfe des Akku-Entladecontrollers erhoben und von genetischen Algorithmen ausgewertet. Diese bestimmen für jede Komponente den Übergangszeitpunkt zwischen Zustand eins und zwei sowie zwischen Zustand drei und vier und für jeden Zustand die mittlere Leistungsaufnahme.

Unterstützung für Parameter wie die Displayhelligkeit ist allerdings nicht vorhanden. Auch wird angenommen, dass jede Interaktion mit der Peripherie nur aus einem einzigen Treiberaufruf besteht – so kann zwar die Energie zum Bestimmen der aktuellen Position bei bereits aktivem GPS-Modul modelliert werden, nicht jedoch der Unterschied zwischen ein- und ausgeschaltetem GPS.

Das Werkzeug *Sesame* unterstützt als erstes hier genanntes neben Smartphones auch Laptops, verwendet aber ausschließlich Laufzeitstatistiken und lineare Regression zur Modellierung [DZ11]. Die Datenerhebung findet während realer Nutzung und somit unter realistischen Bedingungen statt, erfordert aber auch, dass ein repräsentativer Nutzer für das getestete System zur Verfügung steht. Da sie ohne externe Messgeräte auskommt, kann sie vollautomatisch im Hintergrund stattfinden. Auch die Modellerstellung findet automatisch statt und erzeugt bei individueller Kalibrierung Modelle mit einem mittleren Fehler von etwa 5 %.

Eine aktuelle Arbeit mit Bezug zu Sensorknoten ist das kombinierte Mess- und Simulationsframework *EMrise* [ZV16]. Dieses basiert auf Automatenmodellen und nutzt den Simulationsteil, um Hardwarezustände ohne externe Vorgaben zu unterscheiden: Der aktuelle Zustande einer Sensorenkomponente ist schlicht der aktuelle (simulierte) Registerinhalt. Auch Transitionskosten werden unterstützt, zudem kann die Energie sowohl abstrakt auf Transitions- als auch detailliert auf Prozessorinstruktionsebene modelliert werden. Die Durchführung und Auswertung von Messungen erfolgt allerdings manuell mit externer

Mess-Hardware. Zwar können mit Hilfe eines genetischen Algorithmus Optimierungen anhand verschiedener vom Nutzer spezifizierter Parameter durchgeführt und z.B. der Energiebedarf gegen Paketverluste abgewägt werden, im Rahmen dieser Arbeit sind diese jedoch uninteressant.

PowerTOSSIM ist ebenfalls ein Energiesimulator mit eingebauten Energiemodellen für bestimmte Sensorknoten. In [Per+08] wird dessen Portierung auf einen neuen Sensorknoten und die Erstellung des zugehörigen Energiemodells beschrieben.

Dieses ist recht simpel und unterscheidet bei den meisten Komponenten lediglich zwischen „an“ und „aus“ – nur CPU und Funkmodul bekommen feingranulare Zustände. Transitionskosten werden hier ebenso wenig unterstützt wie Parameter, so dass das Modell nur für die Parameter gilt, mit denen es erstellt wurde. Der Grund hierfür sind offenbar Einschränkungen im Simulator. Nennenswerte Automatisierungen bei Modellerstellung und Auswertung werden nicht erwähnt.

Auch [Pra+10] beschreibt die manuelle Vermessung und Modellierung von Sensorknoten, in diesem Fall auf MSP430-Basis. Hier wird mit eigens für die Hardware erstellten Benchmarks je eine Komponente in einer Endlosschleife durch ihre Aktionen und Zustände geführt. Dabei wird neben Peripheriegeräten auch die CPU selbst modelliert, für welche zudem neben verschiedenen Betriebszuständen auch Operationen wie Multiplikation und Division vermessen werden. Die Auswertung findet von Hand statt und ergibt ein zustandsbasiertes Energiemodell.

Einen Fokus auf automatisierte Modellsynthese beim Entwurf eingebetteter Systeme legt Prospector [YF09]. Ziel ist, den Entwurf in fünf Schritte einzuteilen: Erstellung eines einfachen Energiemodells, Vermessung, Modellinferenz, erneute Vermessung zur Modellverifikation und schließlich Optimierung.

Hierzu wird zunächst für jede Komponente und jeden Zustand dieser Komponente von Hand ein eigenes Testprogramm entworfen, welches sie in genau diesen Zustand bringt. Dieses wird automatisch mit Zeiterfassungs- und Signalisierungscode erweitert, so dass durch Programmierung und Ausführung der einzelnen Testprogramme die zugehörigen Zustände vermessen werden. Zudem findet jede Messung mit einer Reihe von Messwiderständen statt, so dass sowohl Ströme im unteren µA- als auch im oberen mA-Bereich messbar sind. Eine Kompensation des Spannungsabfalls ist allerdings ebenso wenig vorhanden wie eine automatische Auswertung der Messergebnisse.

Kapitel 4

Konzept

Die Literaturübersicht zeigt, dass zwar eine Vielzahl von Arbeiten zur Modellierung und Modellsynthese existiert, Automatisierung jedoch primär im Laptop- und Smartphonebereich verbreitet ist und häufig von Unterstützung durch das Betriebssystem oder einem Simulator abhängt. Die Erstellung von betriebssystemunabhängigen Modellen für eingebettete Systeme erfolgt zu großen Teilen in Handarbeit, da die Modelle oft nur Mittel zum Zweck sind – sie dienen beispielsweise der Demonstration eines neu entwickelten Messgeräts oder der Widerlegung anderer Modellannahmen. Auch bieten viele solche Modelle lediglich statische Angaben über das Hardwareverhalten und müssen, sobald ein Parameter wie z.B. die Sendeleistung verändert wird, neu erstellt werden.

In dieser Arbeit wird daher ein Konzept zur automatisierten Erstellung und Verfeinerung von parametrisierten Energiemodellen für eingebettete Systeme ohne spezifische Betriebssystemeigenschaften entwickelt. Insbesondere wird untersucht, wie (und wie gut) Parameter-Abhängigkeiten bei automatenbasierten Modellen erkannt und modelliert werden können.

Die Wahl von Automaten als Modellgrundlage begründet sich darin, dass diese im Allgemeinen eine höhere Güte als analytische Modelle erreichen [McC+11]. Zudem erleichtern sie die Modellierung ohne Performance Counter oder ähnliche Laufzeitvariablen. Es muss lediglich jeder Transition des Energiemodells ein Funktionsaufruf des Treibers für das modellierte Gerät zugeordnet werden und schon kann zur Laufzeit der aktuelle Hardwarezustand und Energieverbrauch protokolliert werden.

Zur Messdatenerhebung wird hier MIMOSA vorgesehen, da es eine erprobte Methode zur Synchronisierung zwischen Messdaten und Hardwaretreibern bietet und sowohl im Mikro- als auch im Milliamperebereich messen kann. Es kann daher eine Vielzahl verschiedener Peripheriegeräte zur Evaluation vermessen und modelliert werden. Bis auf Details bei der Datenerhebung ist das Konzept allerdings von der Wahl der Messtechnik unabhängig.

Das Gesamtkonzept ist an das bei manueller Modellsynthese übliche Vorgehen angelehnt und besteht aus vier Schritten.

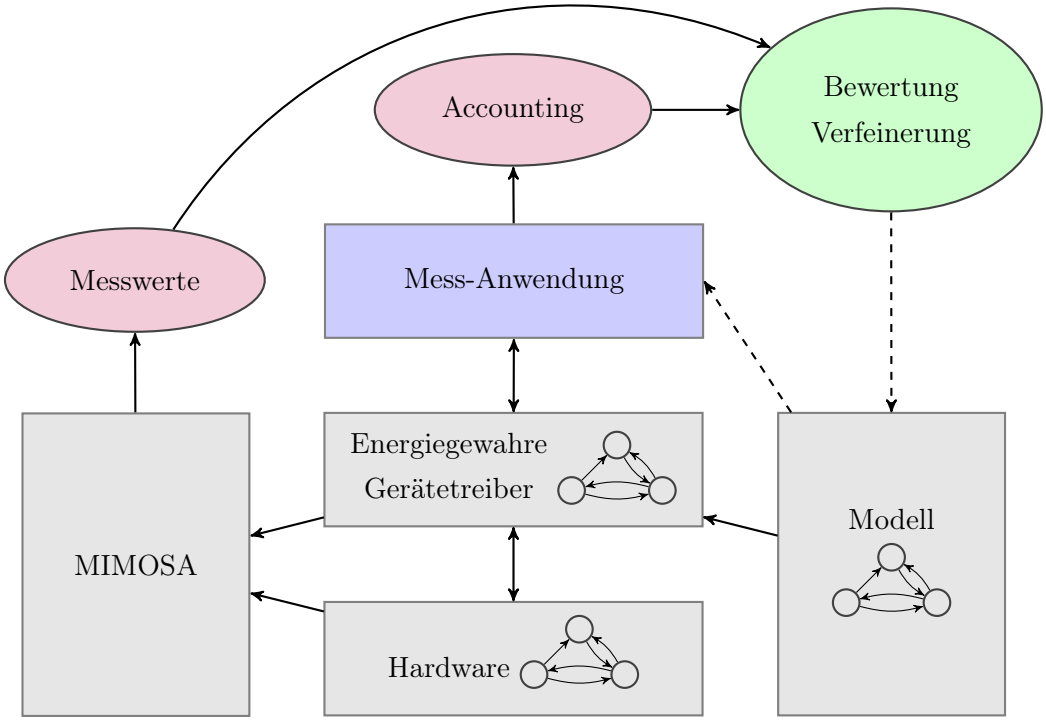

Abbildung 4.1: Konzept zur Modellverfeinerung. Durchgezogene Linien entsprechen vollautomatischen Abläufen, gestrichelte erlauben manuelle Eingriffe durch den Anwender.

1. Der Anwender gibt einen Treiber und ein initiales Modell der Hardware vor. Dieses muss alle zu testenden Zustände, Transitionen und Parameter enthalten; Angaben zu Zeit- und Energieverhalten sind optional.
2. Aus dem Modell wird eine Mess-Anwendung generiert und der Gerätetreiber um Zustands- und Transitionsangaben erweitert. Die Anwendung läuft sämtliche Zustände und Transitionen ab und gibt die protokollierten Abläufe aus.
3. Dieses Programm wird auf den Mikrocontroller geladen. Während seiner Ausführung werden der Energiebedarf der Hardware und die Synchronisierungssignale der Treiber von MIMOSA aufgezeichnet.
4. Aus den Messwerten werden Energie-, Leistungs- und Zeitdaten bestimmt und im Modell eingepflegt. Zudem wird die Güte des Modells untersucht und auf mögliche Unzulänglichkeiten hingewiesen. Diese können beispielsweise fehlende Parameter oder nicht modellierte Parameter-Abhängigkeiten sein.

Mit dem aktualisierten Modell können dann so lange neue Iterationen gestartet werden, bis eine zufriedenstellende Modellgüte erreicht wurde. Die Modellaktualisierung findet überwiegend automatisch statt, lediglich Änderungen an der Zustandsmenge muss der Nutzer selbst umsetzen. Abbildung 4.1 zeigt das Gesamtkonzept und die iterative Modellverfeinerung.

Die folgenden Abschnitte gehen im Detail auf das genutzte Energiemodell und die Umsetzung der Schritte zwei bis vier ein.

4.1 Modellierung

Das Modell dient sowohl als Informationsquelle für Anwender als auch zur Generierung des Testprogramms und der Zustands- und Energiedaten für den Gerätetreiber. Dementsprechend muss es sowohl menschen- als auch maschinenles- und -schreibbar sein. Um dies zu gewährleisten, dient hier das XML-Modell der in Abschnitt 2.5 vorgestellten DFA-Treiber als Grundlage.

Dieses umfasst bereits Angaben über die verschiedenen Hardwarezustände mit zugehöriger Leistung sowie die zwischen den Zuständen möglichen Transitionen mit zugehöriger Energie. Ebenso wird für jede Transition angegeben, ob sie durch Treiberaufrufe aus Anwendungen oder durch eine Unterbrechung ausgelöst wird, und im zweiten Fall zusätzlich das Transitionstimeout (d.h. die Wartezeit bis zu dieser Unterbrechung). Da all diese Angaben für Energiemodelle interessant sind, werden sie beibehalten.

Um auch dem Zeitaspekt von PTAs gerecht zu werden, wird hier zusätzlich die Transitionsdauer berücksichtigt. Weiterhin werden für alle Transitionen Energieangaben in zwei Varianten vorgehalten: Einerseits als absolute Energie, was den Energieangaben in anderen Arbeiten entspricht, und andererseits als relative Energie, aus der die mittlere Leistungsaufnahme des Ausgangszustands herausgerechnet wurde. Wir nehmen an, dass sich durch die Verwendung relativer Transitionsenergie sowohl der Aufwand zur Nutzung von Online-Modellen als auch der mittlere Modellfehler reduzieren lassen. Eine Definition der relativen Energie und eine Begründung dieser Annahme folgen in Abschnitt 4.5.

Zuletzt sollen Zeit- und Leistungsangaben nicht nur statisch, sondern auch in Abhängigkeit von dynamischen Hardware-Parametern wie Bitrate oder Sendeleistung möglich sein. Auch muss das Modell optionale Angaben über Hardware-Eigenheiten machen können, da es hier nicht nur zur Energiemodellierung, sondern auch zur Testprogrammgenerierung verwendet wird. Diese beiden Aspekte werden in den nächsten Abschnitten erläutert. Als Abschluss folgt eine formale Definition des in dieser Arbeit verwendeten Automatenmodells.

Parameter

Sobald ein Peripheriegerät konfigurierbar ist, kann sein Verhalten von in der Konfiguration gesetzten Parametern abhängig sein. Wenn ein Modell für mehr als nur eine feste Konfiguration nutzbar sein soll, muss es diese Abhängigkeiten kennen und zur Verhaltensbestimmung nutzen.

Dazu muss zunächst bekannt sein, welche Parameter es gibt und wie sie verändert werden. Diese Veränderung kann implizit als Nebeneffekt einer anderen Funktion wie bei-

spielsweise einer Initialisierungsroutine erfolgen; eine explizite Änderungen durch eine zugeordnete Konfigurationsfunktion wie *setBitrate(4)* ist ebenfalls möglich. Das Modell enthält dazu eine globale Parameterliste und Annotationen an jeder Transition, welche Parameter implizit oder explizit verändert.

Zudem müssen Parameter zur Bestimmung von Energie-, Leistungs- und Zeitangaben nutzbar sein. Hierzu können jeder solchen Modelleigenschaft drei verschiedene Berechnungsvorschriften zugeordnet werden:

- eine statische Angabe ohne Parametereinfluss,
- eine manuell vorgegebene Berechnungsfunktion und
- eine bei der Auswertung automatisch bestimmte Berechnungsfunktion.

Dabei ist wichtig, dass die Funktionen in einem wohldefinierten Format vorliegen, so dass sie sowohl als Teil des Modells zur Bestimmung des Hardwareverhalten genutzt als auch im Rahmen der Auswertung generiert oder optimiert werden können. Das spezifische Format ist allerdings ein Implementierungsdetail.

Angaben zur Testprogrammgenerierung

Die grundlegende Idee zur Erstellung eines Testprogramms ist einfach: Es müssen lediglich ausgehend vom aktuellen Zustand der Hardware alle erreichbaren Transitionen und Zustände abgelaufen werden bis jeder Zustand und jede Transition mindestens einmal besucht wurde. Durch Ersetzen jeder Transition durch die korrespondierende Treiberfunktion und jedes Zustands durch eine ausreichend hohe Wartezeit ergibt sich ein Testprogramm, welches zusammen mit dem Treiber eine Vermessung aller Hardwarezustände und Transitionen erlaubt.

In der Praxis wird dies zunächst dadurch erschwert, dass der aktuelle Hardwarezustand zu Beginn der Programmausführung nicht bekannt ist. Zur Generierung wird aber ein eindeutiger Ausgangszustand benötigt.

Aus diesem Grund enthalten alle in dieser Arbeit verwendeten Modelle per Konvention den Zustand UNINITIALIZED (kurz „?“), welcher als Startzustand dient. Von diesem aus führt mindestens eine Transition, wie z.B. eine Initialisierungsroutine, in einen wohldefinierten Zustand, der alle weiteren Hardwarezustände erreichen kann.

Als nächstes müssen Funktionsargumente berücksichtigt werden. Dies ist notwendig, um die Hardwarekonfiguration verändern und so parametrisierte Modelle erstellen zu können. Doch auch ohne Parameter sind Argumente wichtig: Viele Treiberfunktionen, die beispielsweise mit einem Funkmodul Daten verschicken oder auf einem Display eine neue Zeile anzeigen, benötigen Argumente mit Nutzdaten. Die Signatur dieser Argumente kann zwar mittels statischer Analyse aus dem Treiberquelltext entnommen werden, ihre Bedeutung und der erlaubte Wertebereich jedoch nicht.

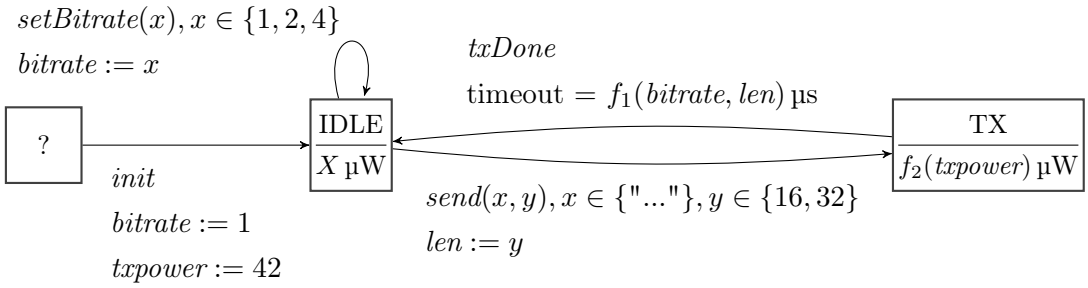

Abbildung 4.2: Beispiel eines Energiemodells für ein Funkmodul mit den globalen Parametern *bitrate*, *len* und *txpower*. Zustände sind mit einer Leistungsangabe annotiert, Transitionen mit Funktionsargumenten, Timeouts und veränderten Parametern. Auf die Angabe von relativer und absoluter Transitionsenergie und -dauer wird aus Platzgründen verzichtet.

Transitionen für Funktionen mit Argumenten erhalten daher zusätzliche Angaben über die Werte, welche für diese Argumente genutzt werden sollen. Hier werden bewusst feste Werte und keine Wertebereiche verwendet, um die zur Modellsynthese genutzten Werte festzuhalten und Anwendern somit zu erlauben, die Übertragbarkeit des Modells auf ihre eigenen Werte zu beurteilen.

Zuletzt sind noch besondere Hardware-Anforderungen zu berücksichtigen. Beispielsweise muss bei eingeschalteten LC-Displays regelmäßig eine sogenannte Polaritätsumkehr durchgeführt werden, da das Display sonst durch den Aufbau statischer Ladungen beschädigt werden kann. Hierzu enthält das Modell einen vom Automaten losgelösten Block mit Sonderanforderungen, in dem konditionelle Zusatztransitionen und Quelltextausschnitte definiert werden können. Zusatztransitionen dienen für Fälle wie die oben genannte Polaritätsumkehr, während die Quelltextausschnitte zur Umsetzung von hier nicht berücksichtigten Anforderungen genutzt werden können.

Ein Beispiel für ein Energiemodell mit Funktionsargumenten zeigt Abbildung 4.2. Dieses verfügt über globale Parameter für Sendeleistung, Paketlänge und Bitrate, die teils explizit durch die Funktionen *send* und *setBitrate* und teils implizit durch die Initialisierungsfunktion *init* eingestellt werden. Zusätzlich benötigt die Funktion *send* Nutzdaten, welche in diesem Modell aber nicht als Parameter behandelt werden.

Automatenmodell

Die formal saubere Umsetzung der oben angesprochenen Testprogrammgenerierung benötigt ein eindeutig definiertes Automatenmodell der Hardware. Dieses basiert auf dem Energiemodell, enthält allerdings keine Energie- und Leistungsangaben, da diese zur Testprogrammgenerierung nicht relevant sind. Auch Zeitangaben werden ausgelassen, da sie bei bei der ersten Mess- und Auswertungsiteration meist noch nicht bekannt sind. Wo nötig, werden stattdessen konfigurierbare Schätzwerte verwendet.

Mit diesen Einschränkungen genügt ein deterministischer endlicher Automat als Hardwaremodell für das Testprogramm. Dies erleichtert das weitere Vorgehen enorm, da im Gegensatz zu Zeitautomaten weder Zeiten noch Nichtdeterminismus berücksichtigt werden müssen. Im Rahmen dieser Arbeit wird das folgende, syntaktisch eingeschränkte Automatenmodell verwendet.

Definition 4.1.1. Ein deterministischer endlicher Automat (kurz *DFA* für *Deterministic Finite Automaton*) ist ein Tupel $\mathcal{A} = (Q, \Sigma, \delta, s, F)$. Er besteht aus der Zustandsmenge Q, dem Alphabet Σ, der partiellen Transitionsfunktion $\delta : Q \times \Sigma \to Q$, dem Startzustand s und der Menge der akzeptierenden Zustände F. Weiterhin gilt $s = \text{UNINITIALIZED} \in Q$ und $F = Q$, so dass $\mathcal{A}$ äquivalent als $\mathcal{A} = (Q, \Sigma, \delta)$ definiert werden kann.

Diese Definition weicht geringfügig von der üblichen Literaturdefinition ab. Die Vorgabe des Startzustands $s = \text{UNINITIALIZED}$ folgt aus den im vorherigen Abschnitt angestellten Überlegungen. Die Vorgabe von $F = Q$ resultiert daraus, dass Peripherietreiber das Konzept akzeptierender Zustände nicht verwenden: Jeder Zustand ist gültig, solange er bekannt und wohldefiniert ist. Daher ist bei der Testprogrammgenerierung jeder Zustand ein akzeptabler Endzustand.

Schließlich ist δ hier eine partielle Funktion, während sie in der Literatur meist total ist. Dies ist eine rein kosmetische Änderung, da jede Transition, für die δ undefiniert ist, als Transition in einen globalen Fehlerzustand betrachtet werden kann. Da dieser aber nicht im Energiemodell vorkommt und das Hardwareverhalten dort undefiniert ist, ist er für die Testprogrammgenerierung ohnehin nicht relevant.

Aus einem Energiemodell wird ein DFA $\mathcal{A} = (Q, \Sigma, \delta)$ wie folgt erzeugt.

- Die Zustandsmenge Q entspricht den Zuständen im Energiemodell.
- Das Alphabet Σ enthält alle aus dem Modell erzeugbaren Funktionsaufrufe. Ist `fun` eine Transition, deren korrespondierende Funktion keine Argumente hat, so wird `fun()` in das Alphabet aufgenommen. Sind die Funktionsargumente $x, y, \ldots$ mit zugehörigen Wertemengen $X, Y, \ldots$ vorgegeben, so wird Σ um die Funktion mit dem Kreuzprodukt aller Argumente erweitert: $\forall x \in X, y \in Y, \ldots : \texttt{fun}(x, y, \ldots) \in \Sigma$.
- Die Transitionsfunktion $\delta : Q \times \Sigma \to Q$ wird analog erzeugt. Wenn das Energiemodell die Transition x mit Startzustand q und Endzustand q' definiert, gilt $\delta(q, \sigma_x) = q'$ für jedes $\sigma_x \in \Sigma$, welches für die Transition x erzeugt wurde. Wenn zu x eine Funktion mit Argumenten gehört, enthält $\mathcal{A}$ also eine Transition für jede Kombination der modellierten Argumente.
- Ist q kein gültiger Startzustand für den Funktionsaufruf σ, so gilt $\delta(q, \sigma) = \perp$.

Aus dem in Abbildung 4.2 gezeigten Energiemodell ergibt sich beispielsweise der in Abbildung 4.3 dargestellte DFA. Zusammen mit den im Energiemodell enthaltenen Angaben zu besonderen Hardware-Anforderungen kann nun ein Testprogramm erstellt werden.

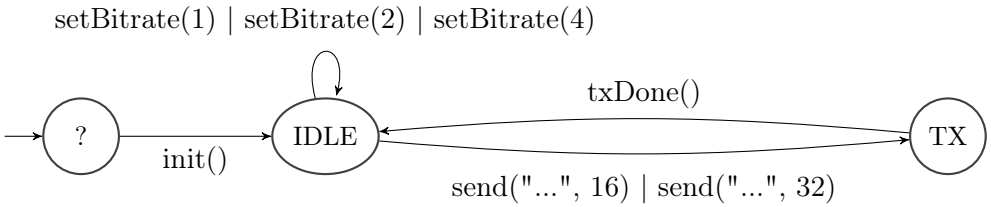

Abbildung 4.3: DFA zur Testprogrammgenerierung für ein beispielhaftes Funkmodul. Das Trennsymbol | gibt an, dass eine Kante für mehrere Transitionen steht.

4.2 Testprogrammgenerierung

Zur Generierung des Testprogramms dient die Sprache $L(\mathcal{A})$ des aus dem Energiemodell erzeugten DFAs $\mathcal{A}$. Diese enthält alle vom Modell erlaubten Folgen von Funktionsaufrufen und führt die Hardware somit durch alle erreichbaren Transitionen und Zustände. Da $\mathcal{A}$ deterministisch ist, ist zudem jedem Wort (d.h. jeder Folge von Funktionsaufrufen) $w \in L(\mathcal{A})$ ein eindeutiger Lauf zugeordnet, welcher die dabei besuchten Zustände angibt. Zu dem Wort $w = \text{init()}\cdot\text{setBitrate(2)}\cdot\text{send("...", 16)}\cdot\text{txDone()}$ aus dem DFA in Abbildung 4.3 gehört beispielsweise der folgende Lauf ρ.

$$\rho = \text{UNINITIALIZED} \xrightarrow{\text{init()}} \text{IDLE} \xrightarrow{\text{setBitrate(2)}} \text{IDLE} \xrightarrow{\text{send("...", 16)}} \text{TX} \xrightarrow{\text{txDone()}} \text{IDLE}$$

Allerdings ist die Sprache $L(\mathcal{A})$ in den meisten Fällen unendlich und daher nicht direkt nutzbar. Stattdessen wird das Testprogramm aus einer Teilmenge von $L(\mathcal{A})$ erzeugt, die nur Wörter enthält, deren Läufe keinen Zustand häufiger als k mal besuchen. Diese Einschränkung nutzt aus, dass sich Hardware bei Wiederholung gleicher Aktionen üblicherweise reproduzierbar verhält, und daher nicht beliebig viele Wiederholungen getestet werden müssen.

Ebenso kann die Menge getesteter Wörter durch Entfernen aller Präfixe weiter verkleinert werden. Grund hierfür ist, dass die Präfixeigenschaft für Wörter wegen des deterministischen Automatenmodells auch für deren Läufe gilt. Alle Zustände, die ein Präfix eines Worts w abläuft, werden also auch von w selbst abgelaufen. Entsprechend können alle Messdaten des Präfixes auch mit dem Wort w selbst erhoben werden. Es ergibt sich die eingeschränkte Sprache $L_k(\mathcal{A})$.

Definition 4.2.1. Die präfixfreie Sprache $L_k(\mathcal{A})$ ist die Teilmenge aller Wörter von $L(\mathcal{A})$, bei deren zugehörigen Läufen kein Zustand häufiger als k mal besucht wird. Die Präfixfreiheit wird so umgesetzt, dass für jedes Wortpaar $w, w' \in L(\mathcal{A})$, bei dem w Präfix von w' ist und beide Wörter die obige Bedingung an ihre Läufe erfüllen, nur w' in $L_k(\mathcal{A})$ aufgenommen wird.

Diese Sprache ist endlich und kann leicht per Tiefensuche in $\mathcal{A}$ und anschließender Entfernung von Präfixen bestimmt werden.

An dieser Stelle besteht die Möglichkeit, $L_k(\mathcal{A})$ durch Herausfiltern von Zuständen oder Beschränken auf Wörter mit einer bestimmten Folge von Transitionen weiter einzuschränken, so dass das Testprogramm nur besonders interessante Zustände und Folgen von Funktionsaufrufen abläuft. Zudem werden jetzt ggf. für einzelne Zustände definierte zusätzliche Funktionsaufrufe aus dem Energiemodell ausgelesen und in die entsprechenden Wörter eingefügt. Die so erzeugte Sprache heißt $L_{k,\star}(\mathcal{A})$.

Schließlich wird ein Testprogramm erzeugt, welches die Hardware initialisiert, eine Pause zum Kalibrieren des Mess-Systems enthält und anschließend die Wörter $w \in L_{k,\star}(\mathcal{A})$ in alphabetischer Reihenfolge abarbeitet. Zwischen je zwei Funktionsaufrufen enthält es eine konfigurierbare Wartezeit, währenddessen der Leistungsbedarf des aktuellen Zustands gemessen werden kann. Ebenso wird jeder Funktionsaufruf, der zu einer Unterbrechungstransition gehört, durch dieselbe Wartezeit ersetzt. Zuletzt werden noch die optionalen zusätzlichen Quelltextausschnitte aus dem Energiemodell eingefügt.

Zusätzlich wird der Treiber des Peripheriegeräts instrumentiert, um mit Hilfe eines Digitalpins Beginn und Ende von Funktionsaufrufen zu signalisieren und in einem internen Puffer die genutzten Transitionen und besuchten Zustände mit Zeit- und Energie-Angaben zu protokollieren. Der Inhalt dieses Puffers wird nach jedem Wort w über die serielle Schnittstelle ausgegeben und anschließend gelöscht. Auch Beginn und Ende des Testprogramms werden dort signalisiert.

Zu jedem Testprogramm wird zudem ein Testplan erzeugt, welcher die Zustände und Funktionsaufrufe angibt, die das Testprogramm gemäß $L_{k,\star}(\mathcal{A})$ ablaufen soll. Ebenso werden zu jedem Zeitpunkt die derzeit gültigen globalen Parameter und zu jedem Funktionsaufruf zusätzlich die genutzten Argumente vorgehalten.

Dieser Plan erlaubt es, Modellfehler wie ausbleibende oder zu spät auftretende Unterbrechungen zu erkennen. Zusammen mit den Ausgaben des Testprogramms ist nun das Durchführen von Messungen und die Auswertung der erhobenen Daten möglich.

4.3 Datenerhebung

Die Vermessung eines Peripheriegeräts besteht aus dem Kompilieren und Übertragen des Testprogramms und des Treibers, gefolgt von einer beliebigen Anzahl von Messläufen. In jedem Messlauf wird der Mikrocontroller zurückgesetzt, das Messgerät kalibriert und anschließend $L_{k,\star}(\mathcal{A})$ wie oben beschrieben komplett durchlaufen und dabei Messdaten erhoben.

Ein Digitalpin des Mikrocontrollers ist währenddessen mit dem Buzzer-Eingang von MIMOSA verbunden, so dass der instrumentierte Treiber den aktuellen Hardwarezustand signalisieren kann. Zudem werden nach jedem Wort w die erhobenen Logdaten ausgegeben. Den zeitlichen Ablauf eines einzelnen Messlaufs zeigt Abbildung 4.4.

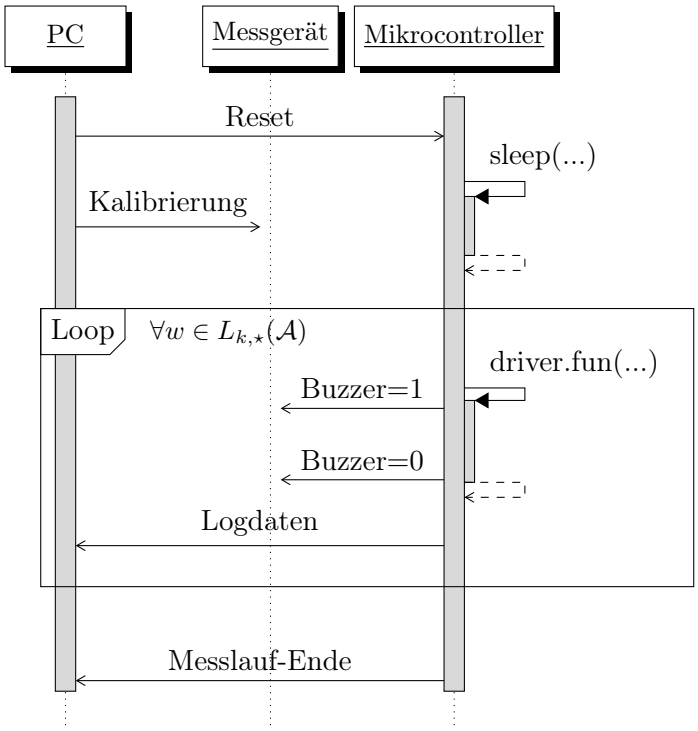

Abbildung 4.4: Vorgänge während eines Messlaufs. Nicht eingezeichnet sind die permanent vom Messgerät an den PC gesendeten Messdaten und weitere Funktionsaufrufe der einzelnen Wörter.

Nach jedem Messlauf findet zudem ein erster Konsistenztest statt. Auf Kalibrierung und Konsistenztest wird nun näher eingegangen.

Kalibrierung

Da die MIMOSA-Komponenten neben einem Grundrauschen auch einer temperaturabhängigen Drift unterliegen, ist vor jedem Messlauf eine Kalibrierung erforderlich [Fal14]. Bei länger andauernden Messungen empfiehlt es sich zudem, alle 15 bis 30 Minuten erneut zu kalibrieren.

Mit Hilfe der MimosaGUI und verschiedener Kalibrierwiderstände kann eine manuelle Kalibrierung durchgeführt werden [Ton14]. Diese Methode wird hier aber nicht genutzt, da sie im Widerspruch zur automatischen Messdatenerhebung steht. Stattdessen wird mit Hilfe einer vom PC angesteuerten Vorplatine zwischen MIMOSA und Peripheriegerät eine automatische Kalibrierung mit bekannten Kalibrierwiderständen durchgeführt.

Hierzu werden an drei verschiedenen Punkten Messungen durchgeführt und die Rohdaten mit dem Sollstrom verglichen. Diese sind der Nullpunkt, bei dem kein Strom fließt, ein Punkt im unteren Messbereich mit einigen zehn µA und einer im oberen Messbereich mit einigen mA.

Anhand dieser Messungen wird eine Kalibrierungsfunktion erzeugt, die Rohdaten in Strom-Angaben umwandelt. Alle Rohdaten, die unterhalb des Nullpunkts liegen, werden

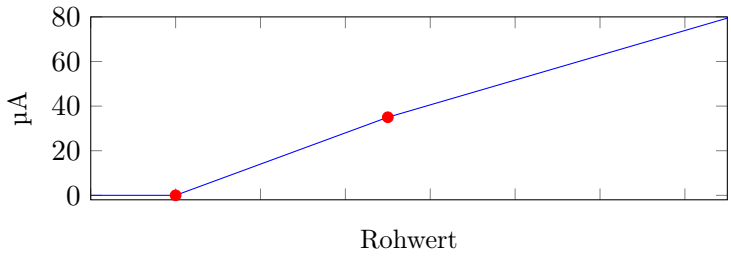

Abbildung 4.5: Beispiel des Kalibrierungskonzepts mit Messpunkten bei 0, 35 und 3500 µA und der daraus erstellten Kalibrierungsfunktion. Der dritte Kalibrierungspunkt ist hier nicht eingezeichnet.

als Rauschen betrachtet und mit 0 µA übersetzt. Rohdaten zwischen Nullpunkt und erstem Kalibrierungspunkt werden anhand der aus diesen beiden Punkten definierten Geraden in Strom-Angaben übersetzt, Rohdaten oberhalb des ersten Punkts entsprechend anhand der Geraden aus erstem und zweiten Kalibrierungspunkt. Abbildung 4.5 zeigt einen Ausschnitt aus einer so erstellten Kalibrierungsfunktion.

Diese Kalibrierung wird automatisch vor jedem Messlauf durchgeführt und die in einem Messlauf erhobenen Rohdaten anhand der vorhergehenden Kalibrierung in Stromverbrauchsdaten umgerechnet.

Konsistenztest

Während eines Messlaufs werden die anfallenden Daten zunächst nur gespeichert. Sobald das Ende des Messlaufs signalisiert wurde, folgt ein Konsistenztest.

Hierbei wird für jedes Wort w der geplante Testablauf mit den vom Mikrocontroller protokollierten Zuständen und Funktionsaufrufen verglichen. Wenn eine geplante Transition oder ein geplanter Zustand im realen Ablauf fehlt oder eine nicht im Plan vorgesehene Transition ausgeführt wurde, weist dies auf einen Fehler im Modell hin und die Messung wird sofort abgebrochen.

Bei Unterschieden im Zustandsnamen gilt dies ebenso, allerdings mit Ausnahme des Zustands UNINITIALIZED. Da dieser lediglich als Startzustand dient und, wie der Name bereits sagt, tatsächlich uninitialisiert und somit undefiniert ist, werden Diskrepanzen hier ignoriert. Da die Leistung dieses Zustands ohnehin nicht Teil des Modells ist, wird auch die Modellqualität nicht davon beeinflusst.

Wenn nach dem Abarbeiten aller geplanten Transitionen noch zusätzliche, als Unterbrechungen bekannte Transitionen protokolliert wurden, ist dies allerdings kein Fehler. Dieser Fall bedeutet lediglich, dass die letzte Transition des Plans die Hardware in einen Zustand brachte, in dem sie eine Unterbrechung auslöst, und die zu dieser Unterbrechung gehörende Transition vor der ersten Transition des nächsten Worts im Testprogramm auftrat. In die-

sem Fall wird der Plan um diese Transition und den ihr vorhergehenden Zustand erweitert und die Parameterwerte von der letzten geplanten Transition übernommen.

Hat eine Messung diesen ersten Test überstanden, kann sie zunächst aufbereitet und anschließend ausgewertet werden.

4.4 Datenaufbereitung

Die Vorverarbeitung wertet Testplan und Messdaten aus, um mit Hilfe der Buzzersignale jedem Zustand und jeder Transition des Testplans die im zugehörigen Zeitraum erhobenen Messdaten zuzuordnen. Ebenso wandelt sie die vom Messgerät gelieferten Rohdaten anhand der aufgezeichneten Kalibrierungswerte in Stromverbrauchsdaten um. Sie findet für jeden Messlauf individuell statt.

Dazu prüft sie zunächst, ob die Buzzersignale mit einer steigenden Flanke beginnen und mit einer fallenden Flanke enden, d.h. ob die Messung tatsächlich in einem Zustand begann und auch in einem Zustand endete. Als nächstes wird sichergestellt, dass die Anzahl an Paaren aus steigender und darauf folgender fallender Flanke der Anzahl an geplanten Funktionsaufrufen entspricht. Ist dies nicht der Fall, kann nicht eindeutig festgestellt werden, welche Messdaten zu welcher Transition bzw. zu welchem Zustand gehören. In diesem Fall wird der Messlauf nicht weiter ausgewertet.

Anschließend wird anhand des Buzzers für jede Transition und jeden Zustand der Start- und Endpunkt in den erhobenen Messdaten bestimmt. An diesem Punkt wird bei Zuständen, die nicht durch eine Unterbrechungstransition beendet werden, zusätzlich die tatsächliche Dauer mit der geplanten Dauer verglichen. Weichen diese um mehr als 50 % voneinander ab, liegen wahrscheinlich Störeinflüsse oder Wackelkontakte in den Signalleitungen vor, so dass der Messlauf ebenfalls nicht weiter ausgewertet wird.

Für jeden Zustand und jede Transition wird nun die Dauer t, die mittlere Leistungsaufnahme P und der Anteil an Messwerten mit Messbereichsüberschreitung bestimmt und protokolliert. Die Daten der einzelnen Messläufe werden hier noch nicht miteinander kombiniert. Nach einer aus vier Messläufen bestehenden Messung werden also u.a. vier mittlere Leistungsaufnahmen pro Element des Testplans gespeichert.

Für jede Transition wird zusätzlich ihr Timeout T (d.h. die Dauer des vorhergehenden Zustands) und ihre relative mittlere Leistungsaufnahme ΔP gespeichert. Dabei handelt es sich um die mittlere Leistungsaufnahme der Transition minus der mittleren Leistungsaufnahme des vorherigen Zustands.

Schließlich werden den Zuständen noch die auf dem Mikrocontroller selbst bestimmten Zeit- und Energiedaten zugeordnet. Falls bei Erstellung des Testprogramms bereits ein Energiemodell vorlag, kann somit auch die Güte der Modellimplementierung auf dem Mikrocontroller beurteilt und auf Probleme z.B. bei der Zeitberechnung hingewiesen werden.

Zustandsschlüssel		Transitionsschlüssel	
Mittlere Leistung	P_i	Dauer	t_i
Dauer	t_i	ggf. Timeout	T_i
Online-Dauer		Absolute Energie	E_i
Online-Energie		Relative Energie	ΔE_i
Messbereichsüberschreitung	$r_{clip,i}$	Messbereichsüberschreitung	$r_{clip,i}$
Parameter	$\vec{p}_i$	Parameter	$\vec{p}_i$

Tabelle 4.1: Aggregierte Messdaten für Zustände und Transitionen.

4.5 Auswertung

Die aufbereiteten Messdaten orientieren sich immer noch am Testplan. Zwar wurden jedem Planelement die zugehörigen Messwerte (je ein Wertetupel pro durchgeführtem Testlauf) zugeordnet, es fand aber noch keine Aggregation nach Zuständen oder Parametern statt. Jeder Zustand und jede Transition ist im Plan mehrfach vorhanden.

Zur Modellverfeinerung werden diese Daten nach drei verschiedenen Schlüsseln aggregiert. Zunächst findet dies ausschließlich nach dem Namen der Transition bzw. des Zustands statt, so dass sich aus den Sammeldaten statische Modellangaben bilden lassen. Als nächstes werden zusätzlich die zum Messzeitpunkt eingestellten globalen Parameter berücksichtigt, so dass mit diesen Daten auch parametrisierte Modelleigenschaften bestimmt und optimiert werden können. Im dritten Fall wird schließlich neben Namen und Parametern noch die Folge aller bis zum jeweiligen Zustand bzw. bis zur Transition abgelaufenen Transitionen verwendet – also das Präfix des zugehörigen Wortes bis zu diesem Punkt. Dies dient zur Untersuchung des Modells im Hinblick auf Abstraktionsfehler und fehlende Parameter.

An dieser Stelle wird für Transitionen zusätzlich aus mittlerer Leistung und Dauer die umgesetzte Energie bestimmt. Wie schon bei der Leistung wird hier sowohl die absolute Energie E als auch die relative Energie $\Delta E = \Delta P \cdot t$ berechnet.

Insgesamt ergeben sich die in Tabelle 4.1 gelisteten Werte. Jeder Wert entspricht dabei einem Mess-Ereignis, d.h. einer einzigen Vermessung eines Zustands oder einer Transition. Beispielsweise enthält der Zustandsschlüssel „TX“ alle Messdaten des TX-Zustands, der Schlüssel „(TX, bitrate=100, txbytes=64)“ nur diejenigen zur Übertragung von 64 Byte bei 100 kbit/s und der Schlüssel „(TX, bitrate=100, txbytes=64, trace=init · setBitrate · setBitrate · send)“ nur solche, bei denen die Bitrate zweimal in Folge eingestellt wurde.

Anhand dieser Daten werden unabhängig voneinander verschiedene Auswertungen vorgenommen.

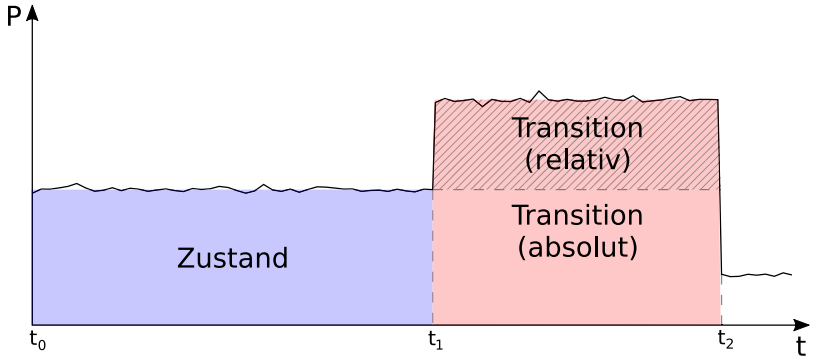

Abbildung 4.6: Unterschied zwischen absoluter Transitionsenergie (rechts, gesamter Bereich) und relativer Transitionsenergie (rechts, schraffierter Bereich).

4.5.1 Statische Modellierung

Zunächst werden lediglich die nach Zustands- bzw. Transitionsnamen aggregierten Daten ausgewertet und so statische Modellwerte ohne Parameter-Unterstützung bestimmt.

Für jeden Zustand wird als neuer Modellwert der Leistungsaufnahme der Median $\tilde{P}$ der Leistungsdaten dieses Zustands eingetragen. Analog wird für jede Transition die Dauer mit $\tilde{t}$, die absolute Energie mit $\tilde{E}$ und die relative mit $\widetilde{\Delta E}$ modelliert. Ein statischer Wert für Timeouts wird nicht vorgegeben, da parameterunabhängige Interruptwartezeiten kein typisches Hardwareverhalten sind.

Der Unterschied zwischen absoluter und relativer Energie ist in Abbildung 4.6 aufgezeichnet. Im Gegensatz zum Absolutwert berücksichtigt die relative Angabe nur die Energie, die nicht plausibel durch den statischen Verbrauch des vorherigen Zustands erklärt werden kann, nämlich nur die darüber hinausgehende Momentanleistung im schraffierten Bereich. Dieser Unterschied hat unmittelbare Folgen für den Berechnungsaufwand für Online-Modelle und potentiell auch für die Güte des Energiemodells.

Bei Nutzung von absoluten Energiewerten muss in einem Online-Modell sowohl zu Beginn (t_1) als auch am Ende (t_0 und t_2) einer Transition die aktuelle Systemzeit bestimmt werden, um mittels $P_{state} \cdot (t_1 - t_0)$ die vom Zustand benötigte Energie zu berechnen. Diese Zeitbestimmung ist u.U. sehr aufwändig und kann Transitionen daher verlangsamen.

Bei Modellierung mit Hilfe der relativen Transitionsenergie sind nur halb so viele Zeitmessungen notwendig: Es genügt der Messpunkt t_0 bzw. t_2 nach dem Ende jeder Transition. Die mit $P_{state} \cdot (t_2 - t_0)$ bestimmte Energie ist zwar höher als die Zustandsenergie, da ein zu langer Zeitraum zur Berechnung verwendet wird, die Differenz zur Zustandsenergie entspricht aber gerade dem Betrag, der in der relativen Transitionsenergie ausgelassen wurde. Dies ist der nicht-schraffierte Bereich zwischen t_1 und t_2.

Bei gemeinsamer Betrachtung von Zustand und der darauf folgenden Transition lässt sich sowohl mit absoluter als auch mit relativer Transitionsenergie die Gesamtenergie korrekt bestimmen, wie die folgende Berechnung zeigt. Diese nutzt E_{tot} als Gesamtenergie

eines Zustands- und Transitionspaars sowie E_{state} und E_{tran} für Zustand bzw. Transition alleine. Analog werden Leistung P und Zeit t verwendet. Die relative Transitionsenergie lässt sich mit diesen Variablen als $E_{rel} = (P_{tran} - P_{state}) \cdot t_{tran} = E_{tran} - P_{state} \cdot t_{tran}$ ausdrücken. Für die Gesamtenergie gilt folgendes:

$$
\begin{aligned}
E_{tot} &= E_{state} + E_{tran} \\
&= P_{state} \cdot t_{state} + E_{tran} && \text{Absolute Energiewerte} \\
&= P_{state} \cdot t_{tot} - P_{state} \cdot t_{tran} + E_{tran} && t_{state} = t_{tot} - t_{tran} \\
&= P_{state} \cdot t_{tot} + E_{rel} && \text{Relative Energiewerte}
\end{aligned}
$$

Beide Varianten sind also äquivalent. Dies gilt auch für $E_{rel} < 0$, d.h. wenn die mittlere Transitionsleistung kleiner als die mittlere Zustandsleistung ist.

Damit ist die statische Modellerzeugung bereits abgeschlossen. Es folgt lediglich noch eine automatische Gütebestimmung mit Hilfe der mittleren absoluten Abweichung vom Modellwert (kurz *MAE* für *Mean Absolute Error*). Für einen Modellwert M und Messdaten $X = (X_1, \ldots, X_n)$ ist diese analog zur parametrisierten MAE aus Abschnitt 2.2.3 wie folgt definiert.

$$MAE(M, X) = \frac{1}{n} \sum_{i=1}^{n} |M - X_i|$$

Zur besseren Veranschaulichung der Abweichung kommt zusätzlich die symmetrische prozentuale Abweichung (kurz *SMAPE* für *Symmetric Mean Absolute Percentage Error*) zum Einsatz.

$$SMAPE(M, X) = \frac{1}{n} \sum_{i=1}^{n} \frac{2 \cdot |M - X_i|}{|M| + |X_i|}$$

Im Gegensatz zur herkömmlichen prozentualen Abweichung, welche über $\left|\frac{M - X_i}{X_i}\right|$ summiert, zeichnet sich diese durch definierte Grenzen aus: Die Abweichung reicht von 0 % (optimal) bis 200 % (maximaler Fehler). Dadurch wird sie von einzelnen Ausreißern wenig beeinflusst. Zudem ist sie auch dann noch nutzbar, wenn einzelne Messwerte Null sind – erst, wenn dies auch auf den Modellwert zutrifft, schlägt die Berechnung fehl.

Im Widerspruch zum Namen ist die SMAPE (wie auch die herkömmliche prozentuale Abweichung) aber nicht vollständig symmetrisch. Eine negative Abweichung vom Modellwert führt zu einem höheren prozentualen Fehler als die gleiche Abweichung in positiver Richtung. Da sie hier lediglich als abschließendes Gütemaß und nicht intern zur Modellauswahl verwendet wird, führt dies aber nicht zur Verfälschung der erzeugten Modelle.

Für die Zustandsleistung $\tilde{P}$, die Transitionsdauer $\tilde{t}$ und die Transitionsenergien $\tilde{E}$ und $\widetilde{\Delta E}$ wird so die Modellgüte bestimmt. Dabei bezieht sich die Menge X jeweils auf die

zugehörigen Messwerte. Zur Gütebewertung der modellierten Leistung des Zustands TX werden beispielsweise MAE und SMAPE von $M = \tilde{P}$ und $X = (P_{\mathrm{TX},1}, \ldots, P_{\mathrm{TX},n})$ berechnet. Auch in den folgenden Abschnitten steht X immer für Messwerte einer beliebigen (aber festen) Metrik eines einzelnen Zustands oder einer einzelnen Transition.

An dieser Stelle ist anzumerken, dass MAE und SMAPE in erster Linie zeigen, wie gut sich das Hardwareverhalten ohne Parameterberücksichtigung beschreiben lässt. Sie sind nicht geeignet, um andere statische Modellierungsmethoden zu evaluieren, da der Median der Messwerte per Definition die geringste mittlere Abweichung aufweist [FKL07]. Zur Erstellung eines Modells mit minimalem mittleren Modellfehler ist der Median der Messwerte also die optimale Wahl.

4.5.2 Erkennung von Parameter-Abhängigkeiten

Sofern im Modell globale Parameter eingetragen sind, werden diese als nächstes betrachtet. Es soll für jede Modelleigenschaft X und jeden bekannten Parameter bestimmt werden, ob (und, falls ja, wie) sie von dem Parameter beeinflusst wird.

Um zu bestimmen, ob überhaupt eine Abhängigkeit vorliegt, nutzen wir die unkorrigierte Standardabweichung σ_X, welche die Streuung von Messwerten $X = (X_1, \ldots, X_n)$ mit arithmetischem Mittel $\overline{X}$ beschreibt.

$$\sigma_X = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (X_i - \overline{X})^2}$$

Durch Partitionierung der Messdaten nach verschiedenen Kriterien und Vergleich der zugehörigen Streuungsmaße können wir abschätzen, von welchen Parametern bestimmte Modelleigenschaften abhängen. Ist beispielsweise die Streuung der Leistungsaufnahme eines Zustands ohne Parameterberücksichtigung sehr hoch, für konstante Parameter aber deutlich niedriger, ist anzunehmen, dass die Leistungsaufnahme von den Parametern beeinflusst wird. Sind hingegen beide Streuungen ähnlich hoch oder niedrig, liegt voraussichtlich keine Abhängigkeit vor.

Dabei ist zu beachten, dass nicht die Streuung innerhalb des Zustands (oder der Transition) betrachtet wird, sondern die Streuung zwischen den Werten verschiedener Messungen. In [Fal14] wird dieses Maß auch als σ_{OUTER} bezeichnet.

Zunächst werden dazu für jede Eigenschaft X eines Zustands oder einer Transition zwei Werte bestimmt. Einerseits die Standardabweichung σ_X ohne Parameterberücksichtigung, und andererseits die mittlere parametrisierte Standardabweichung $\overline{\sigma_X}$.

$$\overline{\sigma_X} = \frac{1}{|Params|} \sum_{\vec{p} \in Params} \sigma_{X,\vec{p}}$$

Diese beschreibt die Streuung der Messdaten für konstante Parameter. Dabei ist *Params* die Menge aller vermessenen Parameterkombinationen und $\sigma_{X,\vec{p}}$ die Standardabweichung aller Messungen von X mit der Parameterkombination $\vec{p}$.

Als nächstes wird das Verhältnis von der Streuung für feste Parameter zur Streuung mit beliebigen Parametern betrachtet. Aus obigen Ausführungen folgt: Sind beide Streuungsmaße nahezu identisch, ist die Metrik X nicht von den Parametern abhängig. Ist $\overline{\sigma_X}$ hingegen deutlich kleiner als σ_X, liegt mit hoher Wahrscheinlichkeit eine Parameter-Abhängigkeit vor. Für diese Heuristik wählen wir $\frac{\overline{\sigma_X}}{\sigma_X} < \frac{1}{2}$ als Schwellwert.

Nun bleibt nur noch die Frage, welche Parameter für diese Abhängigkeit relevant sind. Auch dies lässt sich mit Hilfe der Streuung abschätzen. Ist die Streuung bei Festhalten aller Parameter sehr niedrig, bei Variation des Parameters i aber deutlich höher, so liegt voraussichtlich eine Abhängigkeit von diesem Parameter vor. Hierzu definieren wir $\overline{\sigma_{X,i}}$ als mittlere parametrisierte Standardabweichung bei Variation des Parameters i.

$$\overline{\sigma_{X,i}} = \frac{1}{|Params(i)|} \sum_{\vec{p} \in Params(i)} \sigma_{X,\vec{p}\setminus i}$$

Die Menge $Params(i)$ enthält dabei alle Parameterkombinationen ohne den Parameter i; ebenso ist $\sigma_{X,\vec{p}\setminus i}$ die Standardabweichungen aller Messungen der Eigenschaft X, bei denen die globalen Parameter (ohne Berücksichtigung des Parameters i) $\vec{p}$ gelten. Der Wert $\sigma_{X,\vec{p}\setminus i}$ gibt also die Streuung von Messdaten an, bei denen der Parameter i variabel ist und alle anderen konstant sind. Das Maß $\overline{\sigma_{X,i}}$ ist der Mittelwert von $\sigma_{X,\vec{p}\setminus i}$ über alle Kombinationsmöglichkeiten der verbleibenden Parameter.

Um die Abhängigkeit vom Parameter i festzustellen, wird nun $\overline{\sigma_X}$ mit $\overline{\sigma_{X,i}}$ verglichen. Sind beide nahezu identisch, so hat die Variation dieses Parameters die Streuung nicht nennenswert erhöht. Es ist daher anzunehmen, dass die getestete Eigenschaft nicht von ihm abhängt. Ist $\overline{\sigma_{X,i}}$ hingegen deutlich größer als $\overline{\sigma_X}$, erhöht die Variation des Parameters die Streuung, was für eine Abhängigkeit spricht. Auch hier dient $\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}} < \frac{1}{2}$ als Heuristik.

Somit kann abgeschätzt werden, von welchen Parametern eine bestimmte Eigenschaft eines Zustands oder einer Transition abhängt. Es muss nur noch bestimmt werden, welcher Art diese Abhängigkeit ist, d.h. ob sie beispielsweise linear, logarithmisch oder exponentiell ist. Hierzu geben wir eine Menge von Funktionen vor, die typisch für parameterbasiertes Hardwareverhalten sind, und wählen diejenige, die nach Optimierung per nichtlinearer Regression das beobachtete Verhalten am besten beschreibt. Die folgenden Arten von Funktionen werden dazu verwendet.

- Linear: $f(x) = x$
- Logarithmisch: $f(x) = \ln(x)$
- Logarithmisch mit Verschiebung: $f(x) = \ln(x + 1)$
- Exponentiell: $f(x) = e^x$
- Quadratisch: $f(x) = x^2$
- Bruch: $f(x) = \frac{1}{x}$
- Wurzel: $f(x) = \sqrt{x}$
- Anzahl Eins-Bits: $f(x) = \#_1(x)$

- Anzahl Null-Bits: $f(x) = \#_{0(8)}(x)$ und $f(x) = \#_{0(16)}(x)$

Die Funktionen $\#_1, \#_{0(8)}$ und $\#_{0(16)}$ beschreiben die Anzahl an Eins- und Null-Bits bei Darstellung des Parameters x als binäre Ganzzahl, wobei die Null-Bit-Funktionen zusätzlich zwischen Kodierung als 8- und als 16-Bit-Zahl unterscheiden. Somit können auch einige von Bitmustern abhängige Eigenschaften, wie sie beispielsweise in CPUs auftreten, modelliert werden [Pal+15].

Für jeden Parameter i mit $\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}} < \frac{1}{2}$ wird nun eine Reihe von Regressionsanalysen durchgeführt. Dazu werden die Messwerte anhand von $Params(i)$ so partitioniert, dass innerhalb jeder Partition alle Parameter bis auf i identisch sind.

Für jede Partition und jede oben angegebene Funktion f werden mittels Regressionsanalyse die Regressionsparameter a und b der Funktion $g(p_i) = a + b \cdot f(p_i)$ auf eine minimale Residuenquadratsumme hin optimiert (siehe auch Abschnitt 2.2.3). Dabei ist p_i der Wert des betrachteten Parameters und $g(p_i)$ eine Funktion zur Beschreibung der zu untersuchenden Metrik, also z.B. der Leistung eines Zustands.

Funktionen, bei denen mindestens ein Parameter den Wertebereich verletzt (z.B. $\frac{1}{x}$ für $x = 0$), werden von dieser Analyse ausgenommen und nicht betrachtet. Es wäre zwar möglich, diese um eine geeignete Verschiebung der Parameterwerte zu erweitern, im Interesse möglichst einfacher Modelle wird hiervon aber zunächst abgesehen.

Schließlich werden zu jeder so optimierten Funktion die Residuenquadratsummen der einzelnen Partitionen gemittelt. Die Funktion, welche die geringste mittlere Residuenquadratsumme aufweist, wird als optimale Beschreibung der Abhängigkeit für diesen Parameter betrachtet. Hier wird allerdings nur die Funktion f und nicht die Werte der Regressionsparameter a und b vermerkt, da diese in einem späteren Schritt ohnehin erneut bestimmt werden müssen.

Grund für die Funktionsauswahl per Residuenquadratsumme ist, dass die hier genutzte Regressionsanalyse genau dieses Fehlermaß minimiert, jedoch keinen solchen Bezug zur mittleren Abweichung hat. Es gibt zwar auch Regressionsmethoden, die die mittlere Abweichung minimieren, diese benötigen im Allgemeinen jedoch deutlich mehr Daten, um ein ähnlich gutes Modell wie die hier genutzte Regressionsmethode zu erzeugen [FKL07].

Beispiel 4.5.1. Die Übertragungszeit eines Funkmoduls hängt von den Parametern Datenrate (mit getesteten Werten 100, 150 und 200 kbit/s) und Paketlänge (mit den Werten 8, 32 und 64 Byte) ab. Es soll die Art der Abhängigkeit von der Datenrate bestimmt werden. Hierzu wird jede oben genannte Funktion mit allen Messwerten mit 8 Byte Paketlänge zur Vorhersage der Übertragungszeit trainiert. Dies wird ebenso für alle Messwerte mit 32 und alle mit 64 Byte Paketlänge durchgeführt. Für jede Funktion werden die Residuenquadratsummen der drei Optimierungsläufe gemittelt und die Funktion mit der niedrigsten ausgewählt. In diesem Fall ist dies voraussichtlich $f(x) = \frac{1}{x}$, d.h. ein antiproportionaler Zusammenhang.

Nachdem für jede Modelleigenschaft die Abhängigkeit von jedem einzelnen Parameter bekannt ist, fehlt nun nur noch eine Funktion, die die kombinierte Abhängigkeit von allen relevanten Parametern beschreibt.

4.5.3 Parametrisierte Modellierung

Hier werden zwei Arten von Modellen berücksichtigt. Einerseits kann für einzelne Modelleigenschaften eine manuell vorgegebene Funktion wie z.B. $f(\mathit{bitrate}, \mathit{txbytes}) = a + b \cdot \frac{\mathit{txbytes}}{\mathit{bitrate}}$ bereits im Modell vorhanden sein. Andererseits kann eine per Heuristik festgestellte Parameterabhängigkeit vorliegen.

Falls bereits eine Funktion f vorgegeben wurde, werden ihre Regressionsparameter anhand aller vorhandenen Messwerte per Regressionsanalyse optimiert und die aktualisierten Werte zurück in das Modell geschrieben. Messwerte, in denen ein von der Funktion genutzter Parameter nicht bekannt ist (beispielsweise, weil die Funktion von *txbytes* abhängt, aber noch kein Paket verschickt wurde), werden ausgelassen. Dieser Schritt ist analog zu den parameter- und laufzeitvariablenabhängigen Modellen in verwandten Arbeiten.

Wurde für mindestens einen Parameter i eine Abhängigkeit festgestellt, wird zudem ohne Nutzereingaben eine Funktion erzeugt. Dazu werden die zuvor für jeden relevanten Parameter i bestimmten Beschreibungsfunktionen f_i kombiniert. Da nicht bekannt ist, ob einzelne Parameter individuell oder primär in Kombination mit anderen Parametern das Hardwareverhalten beeinflussen, werden hier alle möglichen Kombinationen aus den für die einzelnen Parameter ermittelten Funktionen gebildet. Es werden also sowohl einzelne Parameter mit Gewichtungsfaktoren nach Art $a_1 f_1 + a_2 f_2 + \cdots$ aufsummiert als auch Kombinationen der Art $a_{12} f_1 f_2 + a_{13} f_1 f_3 + \cdots$. Formal wird dies wie folgt umgesetzt.

Sei eine Modelleigenschaft von den Parametern $p_1, \ldots, p_m$ mit zugehöriger Funktionsmenge $F = (f_1, \ldots, f_m)$ abhängig, wobei jede Funktion f_i den kompletten Parametervektor $\vec{p}$ als Argument erhält, aber nur jeweils den i-ten Parameter zur Berechnung nutzt. Zur Beschreibung einer Modelleigenschaft in Abhängigkeit von allen globalen Parametern definieren wir die Funktion $g(\vec{p})$ wie folgt.

$$g(\vec{p}) = \sum_{F' \in \mathcal{P}(F)} \left(a_{F'} \cdot \prod_{f \in F'} f(\vec{p}) \right)$$

Diese nutzt einen Regressionsparameter $a_{F'}$ für jede Funktionsmenge $F' \subseteq F$; für die leere Menge gilt $\prod_{f \in \emptyset} f = 1$. Zur Veranschaulichung dieses Vorgehens betrachten wir das folgende Beispiel.

Beispiel 4.5.2. Die Übertragungszeit eines Funkmoduls hängt von der Datenrate (*bitrate*) und der Paketlänge (*txbytes*) ab, nicht jedoch von der Sendeleistung (*txpower*). Es wurden die Funktionen $f_{\mathit{bitrate}} : x \mapsto \frac{1}{x}$ und $f_{\mathit{txbytes}} : x \mapsto x$ als optimale Beschreibung identifiziert.

Zur Übertragungszeitberechnung anhand aller Parameter wird nach obigem Schema die folgende Funktion bestimmt.

$$f(bitrate, txbytes, txpower) = a_1 + a_2 \cdot \frac{1}{bitrate} + a_3 \cdot txbytes + a_4 \cdot \frac{txbytes}{bitrate}$$

Anschließend werden die Regressionsparameter der so erzeugten Funktion wie gewohnt per Regression optimiert und die Funktion sowie die optimierten Regressionsparameter in das Modell übernommen.

Zuletzt werden auch hier Fehlermaße zur Beschreibung der Modellgüte berechnet. Im Gegensatz zu Fehlermaßen für statische Modelleigenschaften müssen diese aber mit Vorsicht betrachtet werden, da per Regression optimierte Modelle zur Überanpassung auf die zur Optimierung genutzten Messdaten neigen und die Fehlermaße entsprechend meist zu optimistisch sind. Zur Bestimmung realistischer Fehlermaße kann eine Kreuzvalidierung durchgeführt werden, das Vorgehen dazu wird allerdings erst im Evaluationskapitel in Abschnitt 6.4.2 beschrieben.

Neben MAE und SMAPE wird an dieser Stelle zusätzlich die zur Residuenquadratsumme verwandte Wurzel der mittleren quadratischen Abweichung (kurz *RMSD* für *Root Mean Square Deviation*) verwendet. Zur Berechnung der Fehlermaße werden die Messwerte $X = (X_1, \ldots, X_n)$ mit den Modellwerten $(f(\vec{p}_1), \ldots, f(\vec{p}_n))$ verglichen, welche aus den zugehörigen Parametern $p = (\vec{p}_1, \ldots, \vec{p}_n)$ berechnet werden.

$$RMSD(p, X) = \sqrt{\frac{1}{n} \cdot SSR(p, X)} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(f(\vec{p}_i) - X_i\right)^2}$$

Grund für die Verwendung dieses Maßes ist, dass die Regressionsanalyse auf eine minimale Residuenquadratsumme und somit auch auf eine minimale RMSD hin optimiert, jedoch keinen solchen Bezug zu MAE und SMAPE hat. Der absolute Fehler ist zwar immer noch zur Beurteilung der Modellgüte an sich geeignet, falls verschiedene Funktionen für parametrisierte Modelleigenschaften miteinander verglichen werden, sollte hierzu aber die quadratische Abweichung in Form der RMSD verwendet werden.

An dieser Stelle kann bei einer Auswertung der Modellgüte die Frage auftreten, warum eine manuell vorgegebene oder auch eine automatisch bestimmte Funktion einen verhältnismäßig hohen quadratischen Fehler aufweist. Dieser kann entstehen, weil das Hardwareverhalten auch bei festen Parametern noch hohe Schwankungen aufweist, er kann aber auch bedeuten, dass bisher keine brauchbare Funktion zur Beschreibung des Verhaltens abhängig von den Parametern gefunden wurde.

Um eine Beantwortung dieser Frage zu ermöglichen, definieren wir für jede Modelleigenschaft eine dritte Funktion zur Beschreibung des Hardwareverhaltens. Dies ist die partielle Funktion $l(\vec{p})$, welche nur auf den gemessenen Parameterwerten definiert ist, und für jedes Parametertupel das arithmetische Mittel aller Messwerte mit diesen Parameterwerten zurückgibt. Auch hierfür wird die RMSD bestimmt.

Die Funktion l ist zwar nicht als Modellierungsfunktion geeignet, da sie als partielle Funktion keine Werte für im Testprogramm nicht berücksichtigte Parameterkombinationen vorhersagen kann. Sie kann aber zum Vergleich mit per Regression optimierten Modellierungsfunktionen genutzt werden, da das arithmetische Mittel wie auch die Regressionsanalyse die quadratische Abweichung minimiert. Die RMSD von l ist daher eine untere Schranke für die RMSD aller anderen möglichen Beschreibungsfunktionen.

Falls die RMSD einer vorgegebenen oder geschätzten Funktion also in der selben Größenordnung wie die von l liegt, kann angenommen werden, dass diese Funktion das Hardwareverhalten gut beschreibt – auch dann, wenn beide Fehlermaße eher hoch sind. Falls sie hingegen um mehrere Größenordnung davon abweicht, lohnt sich die Suche nach einer Funktion, die das Verhalten besser beschreibt.

Hierbei muss allerdings beachtet werden, dass die Funktion l ein Musterbeispiel für Überanpassung ist: Sie ist ausschließlich auf den Daten nutzbar, mit denen sie auch optimiert wurde, und sonst unbrauchbar. Da Modellfunktionen auch für andere Daten verwendet werden sollen, müssen diese zwangsläufig generalisieren, was sich meist in einer höheren RMSD niederschlägt. Ein verbleibender kleiner Unterschied zwischen der Güte von l und der Güte einer Modellfunktion hat daher keine Aussagekraft.

4.5.4 Erkennung fehlender Parameter

Zuletzt stellt sich die Frage, ob das Modell das Hardwareverhalten vollständig beschreibt oder ob es noch weitere unbekannte oder übersehene Zustände oder Parameter gibt. Dies kann zwar nur durch detaillierte Auseinandersetzung mit der Dokumentation des Peripheriegeräts abschließend beantwortet werden, die erhobenen Testdaten erlauben aber, erste Vermutungen anzustellen.

Die Idee hierzu ist ähnlich wie schon bei der Erkennung von Parameter-Abhängigkeiten. Jeder Zustand und jede Transition ist in der Regel über verschiedene Wege erreichbar, d.h. es können verschiedene Folgen von Funktionsaufrufen zwischen dem Startzustand UNINITIALIZED und dem Erreichen des Zustands oder der Transition stattfinden. Falls die Streuung einer Eigenschaft ohne Berücksichtigung der zuvor durchgeführten Transitionen hoch, bei Partitionierung nach den Transitionen aber gering ist, ist naheliegend, dass irgendein Funktionsaufruf das Verhalten dieser Eigenschaft verändert. Dies ist ein Indiz für einen im Modell nicht berücksichtigten Parameter.

Wie zuvor wird zunächst überprüft, ob überhaupt eine Transitionsabhängigkeit vorliegt, und falls ja, welche Transitionen für den Unterschied verantwortlich sind. Für ersteres bestimmen wir die Streuung einer Zustands- oder Transitionseigenschaft X unter Berücksichtigung der Parameter $\vec{p}$ und der bis zur Vermessung des Zustands bzw. der Transition abgelaufenen Transitionen, d.h. des bisherigen Wortpräfix w. Für jedes Paar aus festem

Parametertupel $\vec{p}$ und festem Präfix w wird die Streuung bestimmt und anschließend der Mittelwert der Streuungen aller Parameter und Präfixe gebildet.

$$\overline{\sigma_{X,\rightarrow}} = \frac{1}{|\,Traces|} \sum_{(\vec{p},w)\in Traces} \sigma_{X,\vec{p},w}$$

Dabei ist *Traces* die Menge aller Parameterwerte $\vec{p}$ und Präfixe w, mit denen der betrachtete Zustand (oder die betrachtete Transition) vermessen wurde. Dieses Streuungsmaß vergleichen wir mit der parameter-, aber nicht präfixgewahren Standardabweichung $\overline{\sigma_X}$. Gilt $\frac{\overline{\sigma_{X,\rightarrow}}}{\overline{\sigma_X}} < \frac{1}{2}$, so liegt vermutlich eine Präfixabhängigkeit vor und die bisher verwendeten Parameter sind möglicherweise nicht vollständig. Diese Heuristik ist analog zur Überprüfung von $\frac{\overline{\sigma_X}}{\sigma_X}$ zur Erkennung von Parameter-Abhängigkeiten.

Zur Bestimmung der beeinflussenden Transitionen bestimmen wir nun für jede Transition *tr* die Standardabweichung $\overline{\sigma_{X,tr}}$, welche wieder die Parameter $\vec{p}$ festhält, im Präfix w jedoch gewisse Variationen erlaubt: Die Transition *tr* wird bei der Partitionierung nach dem Präfix ignoriert, so dass sowohl Wörter mit als auch Wörter ohne *tr* in die Streuung einfließen.

$$\overline{\sigma_{X,tr}} = \frac{1}{|\,Traces(tr)|} \sum_{(\vec{p},w)\in Traces(tr)} \sigma_{X,\vec{p},w\setminus tr}$$

Die Menge $Traces(tr)$ beschreibt die entsprechenden Paare aus Parameterwerten und Präfixen ohne Berücksichtigung von *tr*, ebenso berücksichtigt das Maß $\sigma_{X,\vec{p},w\setminus tr}$ diese Transition nicht. Diese Maße sind analog zu $\overline{\sigma_{X,i}}$ bzw. $\sigma_{X,\vec{p}\setminus i}$.

Gilt nun $\frac{\overline{\sigma_{X,\rightarrow}}}{\overline{\sigma_{X,tr}}} < \frac{1}{2}$ für eine Transition *tr*, heißt das: Wenn die Transition *tr* bei der Bestimmung der Streuung ignoriert wird, ist diese deutlich höher, als wenn Präfixe mit und ohne *tr* getrennt behandelt werden. Das Vorhandensein bzw. Fehlen von *tr* beeinflusst also das Verhalten der Eigenschaft X. Dies deutet darauf hin, dass *tr* einen im Modell nicht berücksichtigten Parameter verändert.

Auch dieses Konzept lässt sich am besten anhand eines Beispiels veranschaulichen.

Beispiel 4.5.3. Ein Funktreiber verlangt zum Versenden eines Pakets die Einstellung der Sendeparameter mit den Funktionen *setTxPower*, *setDataRate* und *send*. Nach erfolgreicher Paketübertragung wird *txDone* aufgerufen, vor dem Versenden kann zudem die optionale Funktion *enableDynamicPayloads* verwendet werden. Im Testprogramm werden die Funktionsfolgen setTxPower · setDataRate · send · txDone und setTxPower · setDataRate · enableDynamicPayloads · send · txDone mit verschiedenen Parameterkombinationen verwendet.

Während der Auswertung des Timeouts von *txDone* zeigt sich, dass die Streuung unter Berücksichtigung von Parametern und Präfix deutlich niedriger als die Streuung nur unter Berücksichtigung der eingestellten Parameter ist. Bei Aufteilung nach einzelnen Transitionen stellt sich heraus, dass $\overline{\sigma_{X,\text{enableDynamicPayloads}}}$ deutlich höher als $\overline{\sigma_{X,\rightarrow}}$ ist. Werden Transitionsfolgen mit und ohne *enableDynamicPayloads* gemeinsam betrachtet, ist die

Streuung des Timeouts also deutlich höher als bei getrennter Betrachtung von Folgen mit und ohne diesem Funktionsaufruf. Folglich beeinflusst diese Funktion offenbar das Zeitverhalten des TX-Zustands und somit das Timeout von txDone.

An dieser Stelle endet die Auswertung. Falls einzelne Transitionen eine Eigenschaft eines Zustands oder einer Transition beeinflussen, wird dies im Modell vermerkt. Anwender sind dann angehalten, das Datenblatt zu Eigenschaften dieser Transition zu konsultieren und ggf. das Modell durch Hinzufügen eines von der Transition abhängigen Parameters anzupassen. Mit diesem aktualisierten Modell kann eine neue Messung durchgeführt und das detaillierte Verhalten des neuen Parameters automatisch bestimmt werden.

Kapitel 5

Implementierung

Zur Bewertung der Methoden wurde ein Prototyp des Modellverfeinerungskonzepts implementiert. Kern der Implementierung ist das Programm *dfatool*, welches sämtliche im vorherigen Kapitel beschriebenen Aktionen durchführen kann. Zur Modellverfeinerung sind damit nur noch zwei Schritte notwendig: *dfatool loop driver.xml* zur Testprogrammgenerierung und Datenerhebung und *dfatool analyze driver.xml data.tar* zur Auswertung der Daten, Verfeinerung des Modells und Ausgabe der Modellgüte. Beide erfordern keinerlei manuelle Eingriffe.

In diesem Kapitel werden ausgewählte Aspekte dieser Implementierung vorgestellt. Der Fokus liegt auf der Implementierung und Instrumentierung von Gerätetreibern in Kratos, dem XML-Format der Energiemodelle und der Kalibrierung und Verwendung des Messgeräts MIMOSA. Zuletzt folgen Details der zur Evaluierung genutzten Peripheriegeräte sowie der dafür entworfenen Treiber und Hardwaremodelle.

Die interne Struktur des dfatool-Programms und eine Referenz seiner Befehle und Optionen findet sich in Anhang A.2. Ein Praxisbeispiel einer Modellverfeinerung wird zudem in Anhang A.3 vorgestellt.

5.1 Energiegewahre Treiber

Die Implementierung der energiegewahren Treiber orientiert sich an den in Abschnitt 2.5 vorgestellten DFA-Treibern. Da deren voller Funktionsumfang zur Modellverfeinerung nicht notwendig ist, wird hier eine vereinfachte Implementierung verwendet. Das für die Energiemodelle genutzte XML-Format ist allerdings mit den DFA-Treibern kompatibel, so dass im Praxiseinsatz beide Treibervarianten mit den hier erstellten Modellen genutzt werden können.

```
<parameters>
  <param name="symbolrate" functionname="setSymbolRate" functionparam="ksps"/>
  <param name="txbytes" functionname="send" functionparam="len"/>
</parameters>
```

Listing 5.1: XML-Definition der Parameter *symbolrate* (gesetzt vom Argument *ksps* von setSymbolRate(...)) und *txbytes* (gesetzt vom Argument *len* von send(...))

5.1.1 XML-Format

Die Änderungen am XML-Format umfassen primär die Modellierung zusätzlicher Eigenschaften wie Transitionsdauer und relativer Energie, die Unterstützung von Parametern und parametrisierten Eigenschaften und die zur Testprogrammgenerierung benötigten Zusatzinformationen. Diese beinhalten einerseits Funktionsargumente und andererseits Angaben für Peripheriegeräte mit besonderen Anforderungen. Da alle Änderungen auf dem Format der DFA-Treiber aufbauen, sei hier auf Abschnitt 2.5 zur Definition dieses Formats verwiesen. Zwei vollständige Energiemodelle für Peripheriegeräte im erweiterten XML-Format finden sich zudem in Anhang A.1.

Für die statischen Angaben zur relativen Transitionsenergie und Transitionsdauer werden die Attribute *rel_energy* und *duration* zu den bereits vorhandenen Transitionsdefinitionen hinzugefügt. Wie im restlichen Modell wird auch hier Energie in pJ und Zeit in µs angegeben.

Zur Definition der genutzten Hardwareparameter dient das optionale XML-Element *parameters*, welches in der XML-Struktur auf derselben Ebene wie die Zustands- und Transitionsliste liegt. Dieses definiert für jeden Parameter den Namen der für explizite Änderungen genutzten Funktion und des entsprechenden Funktionsarguments. Ein Beispiel hierfür findet sich in Listing 5.1.

Neben expliziten Änderungen können Parameter auch implizit als Nebeneffekt anderer Funktionen verändert werden. Solche Transitionen werden mit einem *affects*-Element gekennzeichnet, welches für jeden von der Funktion auf den Wert x gesetzten Parameter p ein Kind der Form `<param name="`p`" value="`x`">` enthält.

Zur Definition parameterabhängiger Eigenschaften werden eigene XML-Knoten unterhalb der zugehörigen Transitions- bzw. Zustandselemente verwendet. Dies sind beispielsweise *powerfunction* für die Leistung eines Zustands oder *timeoutfunction* für das Timeout einer Transition. Jede solche Eigenschaft erhält wiederum ein *user*-Kind für von Anwendern vorgegebene Funktionsvorschriften und ein *estimate* für die im Rahmen der Auswertung ermittelten.

Funktionsvorschriften erhalten die derzeitigen Werte der Regressionsparameter als Attribute und die Funktion selbst in Textform. Diese kann sich über den Schlüssel *param(i)* auf den i-ten Regressionsparameter und mittels *global(x)* auf den globalen Parameter x be-

```
<powerfunction>
  <user param0="23" param1="5.33"><![CDATA[
    param(0) + param(1) * np.exp(global(txpower))
  ]]></user>
</powerfunction>
```

Listing 5.2: Eine manuell vorgegebene Funktion zur Leistungsbeschreibung mit zwei Regressionsparametern und Abhängigkeit vom globalen Parameter *txpower*.

```
<transition name="send">
  <!-- ... -->
  <param name="str"><value>"Hello World. This is just a test."</value></param>
  <param name="len">
    <value>8</value>
    <value>16</value>
    <value>32</value>
  </param>
</transition>
```

Listing 5.3: Definition von Funktionsargumenten zur Testprogrammgenerierung.

ziehen. Abgesehen davon sind alle mathematischen Operationen erlaubt, die von Python 3.5 und NumPy mit *import numpy as np* unterstützt werden. Listing 5.2 zeigt ein Beispiel für eine solche Funktionsdefinition.

Schließlich müssen noch die zur Testprogrammgenerierung notwendigen Funktionsargumente definiert werden. Hierzu erhält jede Transitionsdefinition ein *param*-Kind je Argument, welches den Namen des Arguments aus der C++-Funktionssignatur und die zum Test genutzten Werte dafür auflistet. Falls es sich um eine Einstellfunktion für einen Modellparameter handelt, ist dies gleichzeitig die Vorgabe der zur Modellverfeinerung genutzten Parameterwerte. Ein Beispiel hierfür findet sich in Listing 5.3.

5.1.2 Einbindung des Modells

Um für bereits vorhandene, nicht-energiegewahre Treiber mit möglichst wenig Aufwand ein Energiemodell zu erstellen, ist die Buchführung über Zustände, Transitionen und Energiedaten fast vollständig vom restlichen Treiberquelltext getrennt. Die zur Erstellung eines Energiemodells notwendigen Änderungen an einem Treiber bestehen lediglich aus

- dem Erben von der Basisklasse *DFADriver* und
- dem Einbinden je eines Header- und eines Implementierungsausschnitts, welche die statische Variable *statepower* und die Enumeration *State* zur Klasse hinzufügen und initialisieren.

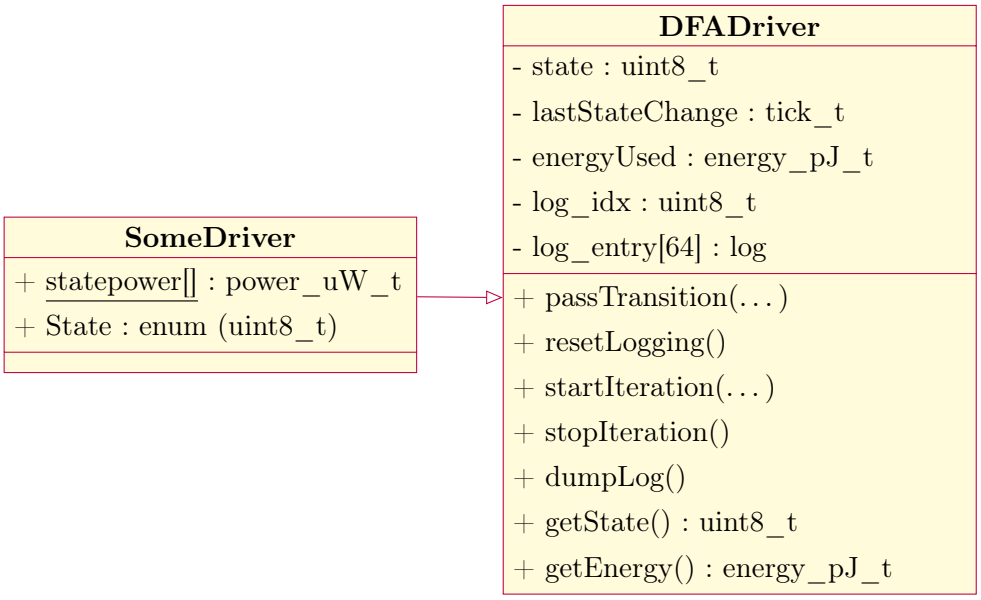

Abbildung 5.1: Die Basisklasse DFADriver und ein sie benutzender Treiber SomeDriver.

Die Generierung dieser Ausschnitte erfolgt automatisch. Auch die Instrumentierung des Treibers zur Buchhaltung über und Signalisierung von Zustands- und Transitionsübergängen findet ohne manuelle Interaktion statt. Hierzu werden aus dem Modell und der Signatur der Treiberfunktionen Aspektheader generiert, die die Treiberfunktionen entsprechend verändern.

Die wichtigsten Elemente der Klasse DFADriver sind in Abbildung 5.1 dargestellt. Sie basiert auf der Klasse DFA_Driver (vgl. Abschnitt 2.5) und enthält gemeinsame Funktionen und Datenstrukturen für energiegewahre Treiber. Dies umfasst Variablen für den aktuellen Zustand, den Zeitpunkt der letzten abgeschlossenen Transition und die bisher insgesamt verbrauchte Energie. Zusätzlich enthält sie eine Logstruktur, die für 64 Paare aus einem Zustand und der darauf folgenden Transition die ID von Zustand und Transition sowie die vom System bestimmte Aufenthaltszeit im Zustands-Transitions-Paar protokolliert. Mit der Funktion *passTransition* werden Energiezähler und Logdaten aktualisiert und mit *resetLogging* wieder zurückgesetzt. Zudem können die Logdaten mittels *dumpLog* in einem festen Format ausgegeben werden. Die Funktionen *startIteration* und *stopIteration* signalisieren Beginn und Ende des Messlaufs.

Für den Treiber generierte Aspekte binden nach jeder Verwendung einer Treiberfunktion einen passTransition-Aufruf ein, der die mittlere Leistung des aktuellen Zustands, Energie und Nummer der durchgeführten Transition und die Nummer des neuen Zustands als Argumente erhält. passTransition bestimmt die aktuelle Systemzeit, berechnet die im vorherigen Zustand verbrauchte Energie und aktualisiert die Daten. Zur Zeitbestimmung wird Quelltext aus der DFA-Treiber-Implementierung verwendet, der auf dem MSP430 Zeitstempel mit einer Auflösung von 2 µs produziert.

```
advice execution("% LM75::getTemp(...)") : after () {
  tjp->target()->passTransition(LM75::statepower[tjp->target()->state], 21713041, 2, 2);
};

advice call ("% RF24::begin(...)") && !within("RF24") : after() {
  tjp->target()->passTransition(RF24::statepower[tjp->target()->state], 1704800, 0, 2);
};
```

Listing 5.4: Advice zur Protokollierung des Energiebedarfs bei jedem Funktionsaufruf (oben) und nur bei externen Funktionsaufrufen (unten).

Im Normalfall nutzen die Aspekte einen *execution*-Pointcut, welcher den passTransition-Aufruf an das Ende jeder Ausführung der ausgewählten Treiberfunktionen anhängt. Bei Treibern, in denen eine modellierte Funktion wiederum eine andere modellierte Funktion aufrufen kann, ist dies allerdings nicht möglich: Dann würde eine Transition während einer anderen Transition stattfinden, was im Modell nicht vorgesehen ist und auch nicht zum Plan des Testprogramms passt.

Hier wird stattdessen ein *call*-Pointcut genutzt, der nur Funktionsaufrufe zulässt, die nicht aus der Treiberklasse stammen. Somit werden nur externe Funktionsaufrufe protokolliert. Da dies durch Einfügen von Quelltext an jedem Funktionsaufruf umgesetzt wird, erhöht es die Programmgröße allerdings signifikant. Listing 5.4 zeigt Beispiele für beide Arten von Aspekten.

Schließlich muss noch der am Buzzer angeschlossene GPIO-Pin mit Synchronisierungssignalen versorgt werden. Auch hierzu wird ein Aspekt verwendet, welcher einen einstellbaren Pin bei Systemstart als Ausgang konfiguriert und anschließend vor jeder Transition auf Eins und nach jeder Transition mit 29 µs Verzögerung auf Null setzt. Diese Wartezeit sorgt dafür, dass auch Transitionen mit einer Dauer unterhalb der Abtastrate von MIMOSA sicher erkannt werden, und wird bei der Auswertung wieder herausgerechnet.

Diesem Aspekt wird zuletzt eine niedrigere Priorität als dem passTransition-Aspekt zugewiesen. Dies sorgt dafür, dass nach jedem Funktionsaufruf zuerst der Buzzer-Pin auf Null gesetzt und dann passTransition(...) aufgerufen wird. Somit beeinflusst die Dauer von passTransition nicht die von MIMOSA gemessenen Zeit- und Energiedaten von Transitionen. Ein Beispiel für einen solchen Aspekt zeigt Listing 5.5.

Zuletzt ist anzumerken, dass die verwendeten Nummern für Zustände und Transitionen der Definitionsreihenfolge in der XML-Datei entsprechen. Sie sind somit wohldefiniert.

5.2 Testprogrammgenerierung

Das Testprogramm wird als Kratos-Anwendung generiert und als einzige Anwendung in das Betriebssystem eingebunden. Somit wird es vom Scheduler nie verdrängt und nur

```
aspect LM75_Trigger {
  pointcut Transition() = "% LM75::shutdown(...)" || "% LM75::start(...)" /* ... */ ;
  advice execution("void  initialize_devices ()") : after() {
    setOutput(4, 6);
  }
  advice execution( Transition()) : before() {
    pinHigh(4, 6);
  }
  advice execution( Transition()) : after() {
    for (unsigned int i = 0; i < 64; i++)
      asm volatile ("nop");
    pinLow(4, 6);
  }
  advice execution( Transition()) : order("LM75_DFA", "LM75_Trigger");
};
```

Listing 5.5: Synchronisierungsaspekt zur Datenerhebung mit MIMOSA.

durch die wegen ihrer Kürze und Seltenheit vernachlässigbaren periodischen Scheduler-Unterbrechungen beeinflusst. Standardmäßig erlaubt es für jedes Wort bis zu zwei Besuche im gleichen Zustand und nutzt 1000 ms Wartezeit je Zustand, diese Werte sind aber konfigurierbar. Auch können hier manuell Zustände ausgeschlossen oder nur Wörter mit einer bestimmten Art von Transitionsfolgen erlaubt werden.

Die Modellierung des Automaten $\mathcal{A}$ und die Generierung der Sprache $L_{k,\star}(\mathcal{A})$ werden vom Perlmodul FLAT::DFA[1] unterstützt. Die letztendlich erzeugte Anwendung enthält eine Endlosschleife, die bei jedem Durchlauf zunächst zwölf Sekunden Wartezeit zur Kalibrierung von MIMOSA vorsieht. Anschließend wird mit *startIteration(n)* der Beginn eines Testlaufs mit n Wörtern signalisiert.

Es folgt für jedes Wort ein Block aus *resetLogging()*, den Funktionsaufrufen und Wartezeiten und *dumpLog()*. Abschließend wird mittels *stopIteration()* das Ende des Testlaufs signalisiert, so dass der die Messung durchführende PC einen ersten Konsistenztest durchführen und einen neuen Testlauf anstoßen kann.

An dieser Stelle werden zusätzlich die Aspekte und C++-Quelltextausschnitte zur Instrumentierung des Treibers und Nutzung der Zustands- und Energiedaten (siehe Abschnitt 5.1.2) generiert. Falls noch keine Leistungs- und Energiedaten vorhanden sind, wird für jeden Zustand 0 µW und für jede Transition 0 pJ eingetragen. Das für die Aspekte notwendige Wissen über die Signatur der Treiberfunktionen wird aus der vom AspectC++-Compiler generierten Datei *repo.acp* entnommen, welche Informationen über die Datei- und Klassenstruktur des gesamten Projekts enthält.

[1] https://metacpan.org/pod/FLAT::DFA

Gleichzeitig wird eine JSON-Datei mit dem Testplan erzeugt. Diese dient als Grundlage für alle späteren Datenerhebungs- und Auswertungsschritte und wird in jedem Schritt erweitert.

Ein Ausschnitt aus einem Testprogramm für ein Funkmodul findet sich in Listing 5.6. Der *buzzer.set*-Aufruf zu Beginn bewirkt dabei, dass alle folgenden sleep-Aufrufe die Wartezeit 200 ms verwenden.

5.3 Messwerterfassung

Zur Erfassung der Messwerte kompiliert dfatool Kratos mit dem zuvor erzeugten Testprogramm und überträgt es auf den MSP430. Anschließend werden so lange Messläufe durchgeführt und protokolliert, bis die Messung manuell beendet wird. Ein einzelner Messlauf besteht dabei aus dem Starten der Messung mit MIMOSA, einer Kalibrierung und der Datenerhebung.

5.3.1 Kalibrierung von MIMOSA

Zur automatischen Kalibrierung von MIMOSA wurde eine Platine entwickelt, welche zwischen MIMOSA und Peripheriegerät angeschlossen wird und MIMOSA wahlweise mit dem Peripheriegerät oder einem von drei Kalibrierwiderständen verbindet. Zur Auswahl des Ziels dienen Relais mit einem Schaltwiderstand von unter 100 mΩ, so dass die Beeinflussung der Messung durch die Relaiskontakte vernachlässigt werden kann. Zudem besteht kein elektrischer Kontakt zwischen dem Messkreis von MIMOSA und der Relais-Ansteuerung, so dass auch hier keine Einflüsse auf die Messung entstehen.

Als Kalibrierungsziele kommen ein 984 Ω-Widerstand, ein 99,013 kΩ-Widerstand und ein offener Kontakt („∞ Ω“) zum Einsatz. Bei der verwendeten Spannung von 3,6 V ergeben sich dadurch Sollströme von 3,66 mA, 36,4 µA und 0 µA.

Zur Ansteuerung der Relais wird ein ATTiny2313A-Mikrocontroller mit USB-Schnittstelle und angepasster PowerSwitch-Firmware[2] verwendet. Mit Hilfe des dazu entwickelten *mimosactl*-Programms kann zwischen Vermessung des Peripheriegeräts und den drei Kalibrierwiderständen gewählt werden. Bilder von USB-Schnittstelle und Relaisplatine finden sich in Abbildung 5.2.

Während der im Testprogramm vorgesehenen Kalibrierungspause werden nacheinander die drei Widerstände (zuerst ∞ Ω, dann 984 Ω und schließlich 99,013 kΩ) eingestellt und für je zwei Sekunden vermessen, anschließend wird MIMOSA wieder mit dem Peripheriegerät verbunden. Die Auswertung dieser Kalibrierungsdaten findet erst bei der Aufbereitung der Messwerte statt.

[2] https://www.obdev.at/products/vusb/powerswitch.html

```
1  void DriverEvalThread::action()
   {
3    Guarded_Buzzer buzzer;

5    while (1) {
       /* wait for MIMOSA calibration */
7      buzzer.sleep(12000);
       buzzer.set(200);
9      radio.startIteration(264);

11     /* ... */

13     /* test run 56 start */
       radio.resetLogging();
15     /* Transition UNINITIALIZED -> IDLE */
       radio.init();
17     buzzer.sleep();
       /* Transition IDLE -> TX */
19     radio.send("Hello World. ... ", 127);
       buzzer.sleep();
21     /* Transition TX -> IDLE */
       /* wait for txDone interrupt */
23     buzzer.sleep();
       /* Transition IDLE -> XOFF */
25     radio.crystal_off();
       buzzer.sleep();
27     radio.dumpLog();

29     /* ... */

31     radio.stopIteration();
     }
33 }
```

Listing 5.6: Ausschnitt aus dem Testprogramm für das CC1200-Funkmodul.

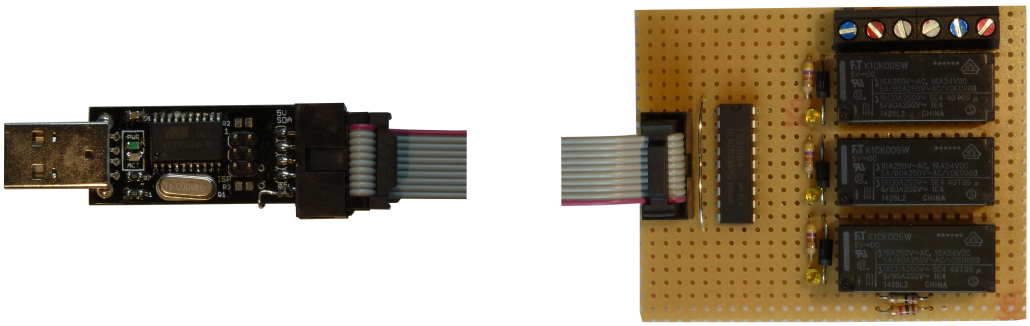

Abbildung 5.2: USB-Steuerung (links) und Relaisplatine (rechts) zur automatischen Kalibrierung von MIMOSA.

5.3.2 Datenerhebung

Nach der Kalibrierung wird mit dem Perlmodul Device::SerialPort[3] eine Verbindung zur seriellen Schnittstelle des MSP430 hergestellt und die Log-Ausgaben des Testprogramms protokolliert. Hierzu wird zunächst auf das von startIteration ausgegebene Startsignal gewartet und anschließend bis zum Stoppsignal von stopIteration aufgezeichnet. Bei ungültigen Ausgaben, die nicht dem in der Klasse DFADriver eingestellten Format entsprechen, wird die Messung sofort abgebrochen.

Sobald stopIteration das Ende des Messlaufs signalisiert, wird die MIMOSA-Messung beendet und mit Hilfe des im JSON-Format gespeicherten Testplans der Konsistenztest durchgeführt. Anschließend wird der um die Log-Ausgaben erweiterte Testplan ebenfalls als JSON-Datei gespeichert und zusammen mit der aktuellen MIMOSA-Logdatei archiviert.

Dieses Vorgehen wird bis zum manuellen Abbruch der Datenerhebung wiederholt, so dass beliebig viele Messläufe möglich sind. Bei einem Abbruch wird der derzeitige Testlauf nicht mehr berücksichtigt, so dass sich nach n vollständigen Testläufen ein Archiv mit n MIMOSA-Logdateien und einem Testplan mit n Datensätzen je Transition und Zustand ergibt.

5.4 Datenaufbereitung

Die bisher genannten Schritte werden von *dfatool loop* durchgeführt und sind vollständig in Perl 5.20 implementiert. Die Aufbereitungs- und Auswertungsschritte ab diesem Punkt finden im Rahmen von *dfatool analyze* statt und nutzen zu großen Teilen Python 3.5 mit den Modulen NumPy und SciPy, da diese einen effizienten und komfortablen Umgang mit großen Datenmengen ermöglichen.

Zunächst werden die von MIMOSA erzeugten Rohdaten anhand der zu Beginn der Messung durchgeführten Kalibrierung in µA-Angaben umgewandelt. Dies findet für jeden

[3] https://metacpan.org/pod/Device::SerialPort

Messlauf separat statt, so dass die einzelnen Messläufe auf Mehrkernsystemen parallel ausgewertet werden können.

Zur Kalibrierung wird in den ersten zehn Sekunden der Messdaten nach dem ersten Punkt gesucht, an dem der ohne Kalibrierung berechnete Strom mindestens 60 % des Sollstroms für den 984 Ω-Widerstand beträgt. Wenn mindestens zwei Sekunden später der Strom auf weniger als 40 % dieses Werts abfällt, wurden Beginn und Ende der 984 Ω-Kalibrierungsphase erkannt. Die ∞ Ω-Kalibrierung liegt dann in den zwei Sekunden davor und die 99,013 kΩ-Kalibrierung in den zwei Sekunden danach.

Diese Methode ist notwendig, da die MIMOSA-Kalibrierung vom PC und nicht vom MSP430 durchgeführt wird. Es steht daher kein Buzzersignal zur Verfügung, mit dem die verschiedenen Kalibrierungsschritte direkt erkannt werden könnten. Konnten Beginn und Ende der 984 Ω-Phase nicht identifiziert werden, wird die Aufbereitung dieser Messung abgebrochen, da vermutlich ein Fehler im Kalibrierungslauf vorliegt.

In jedem Kalibrierungsbereich wird der mittlere Rohdatenwert dem zugehörigen Soll-Stromfluss gegenübergestellt und so die Kalibrierungsfunktion bestimmt und die Rohdaten in Strom-Angaben überführt. Falls die Messung mit dem 680 Ω-Shunt durchgeführt wurde, wird die Kalibrierungsfunktion allerdings nur aus den Daten zu den Sollströmen 0 µA und 36,4 µA erzeugt, da der dritte Wert mit 3,66 mA außerhalb des Messbereichs liegt.

Anschließend wird jeder Bereich zwischen zwei Buzzerflanken als Zustand (Buzzersignal Null) oder Transition (Buzzersignal Eins) deklariert. Dabei wird jede Flanke erst zwei Messpunkte (20 µs) später ausgewertet, da MIMOSA durch die Messen-Integrieren-Zurücksetzen-Methode eine Verzögerung von 20 µs zwischen Stromdaten und Buzzer aufweist. Falls sich hier Inkonsistenzen zeigen, wird die Datenaufbereitung ebenfalls abgebrochen.

Schließlich werden jedem Zustand und jeder Transition aus dem Testplan die aus den MIMOSA-Daten bestimmten Leistungs-, Zeit- und Messbereichsüberschreitungsdaten zugeordnet. Die so aufbereiteten Daten werden in einer JSON-Datei pro Messlauf gespeichert. Für einen späteren Vergleich werden zudem die Kalibrierungsdaten und die Abweichung von den unkalibrierten Daten vermerkt.

Zuletzt werden die JSON-Dateien aller erfolgreichen Datenaufbereitungen mit dem Testplan und den Aufzeichnungen der seriellen Schnittstelle vereinigt, so dass nun jedem Zustand und jeder Transition n Logdatensätze und bis zu n Messdatensätze zugeordnet sind.

5.5 Auswertung

Zur Auswertung werden die Messdaten zunächst nach den drei in Abschnitt 4.5 genannten Schlüsseln aggregiert. Hierzu kommen Python-Dictionaries mit NumPy-Arrays für die ein-

zelnen Messdaten und Tupeln für die Parameterwerte zum Einsatz. Die Parameter werden dabei alphabetisch sortiert, so dass ihre Beschreibung als Tupel immer eindeutig ist.

Mit den aggregierten Daten werden anschließend die verschiedenen Analyseschritte durchgeführt, Fehlermaße bestimmt, die Auswertungsergebnisse grafisch und in Textform dargestellt und das XML-Modell aktualisiert. Neben Modellwerten und Gütemaßen werden hier auch Hinweise ausgegeben, falls beispielsweise eine Modelleigenschaft als statisch deklariert wurde, sich im Rahmen der Auswertung aber eine Parameterabhängigkeit gezeigt hat. Zudem erfolgt eine Fehlermeldung, falls in mindestens einer Messung eines Zustands oder einer Transition mehr als 1 % der Messdaten die Grenze des Messbereichs erreicht haben, da die Messung dann voraussichtlich unzuverlässig ist und mit einem kleineren Messwiderstand wiederholt werden sollte.

Funktionen werden mit der Methode *scipy.optimize.least_squares* optimiert, welche eine nichtlineare Regression mit dem Trust-Region-Verfahren zur iterativen Minimierung der Residuenquadratsumme durchführt. Die in Textform angegebenen Modellfunktionen werden dazu mit Hilfe des *eval*-Schlüsselworts und geringfügiger Vorverarbeitung in Lambdafunktionen überführt, die die von *least_squares* übergebenen Regressions- und Modellparameter nutzen. Auch die im Rahmen der Auswertung erzeugten Funktionen werden zunächst als Text generiert und dann so weiterverarbeitet, um denselben Quelltext für beide Funktionstypen nutzen zu können.

Die grafische Darstellung der Auswertungsergebnisse wird durch das Pythonmodul *matplotlib* unterstützt. Für statische Modellwerte wie Zustandsleistung, Transitionsdauer oder Transitionsenergie werden Boxplots erzeugt, die neben dem Median als neuem Modellwert auch den alten Modellwert, unteres und oberes Quartil der Messwerte und etwaige Ausreißer anzeigen.

Für parameterabhängige Eigenschaften sind Boxplots nicht geeignet, da sie durch die Vielzahl an erzeugten Parameterkombinationen kaum lesbar wären. Doch auch eine Darstellung des Funktionsgraphen in Abhängigkeit von den Parameterwerten ist schwierig, da Funktionen häufig von mehr als einem Parameter abhängen – zur Darstellung einer von n Parametern abhängigen Funktion wird aber ein Graph mit $n + 1$ Dimensionen benötigt.

Aus diesem Grund wird hier stattdessen für jeden Parameter ein eigenes Diagramm erzeugt, welches auf der x-Achse die Werte dieses Parameters und auf der y-Achse die durch die Funktion modellierte Eigenschaft angibt. Dort wird für jede Wertekombination der verbleibenden Parameter ein Funktionsgraph mit zugehörigen Messwerten eingezeichnet. Somit ist sowohl das Verhalten der Hardware als auch die Güte der Modellfunktion für jede Parameterkombination sichtbar.

Beispiele für diese Arten von Graphen und einige mit dieser Implementierung verfeinerte Energiemodelle folgen im nächsten Kapitel.

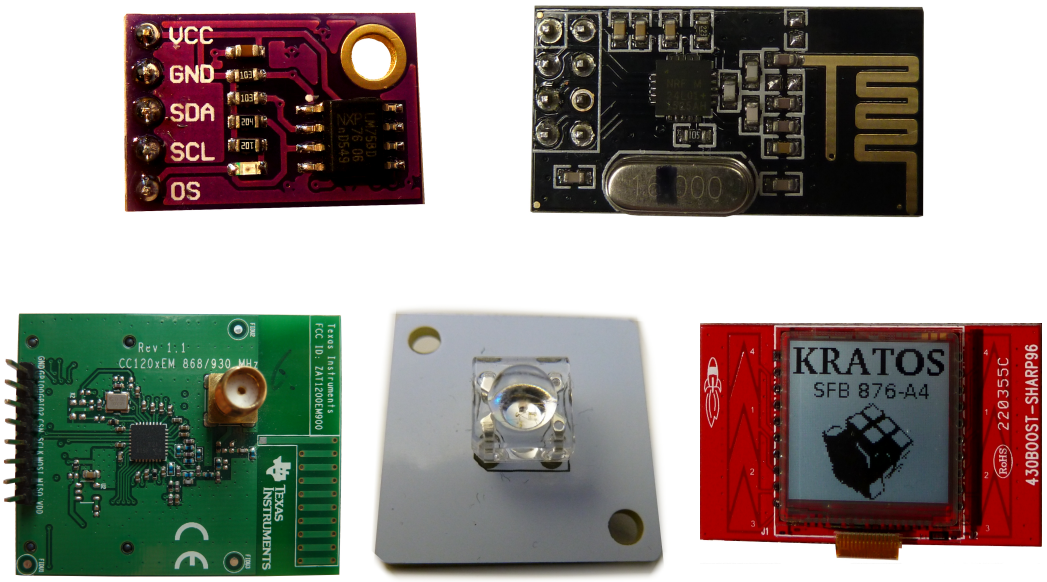

Abbildung 5.3: Zur Evaluation genutzte Peripheriegeräte. Von links nach rechts: Temperatursensor, Funkmodul nRF24L01+, Funkmodul CC1200, MicroMoody und Display.

5.6 Gerätetreiber

Zur Evaluation der Modellverfeinerung dienen fünf Peripheriegeräte mit verschiedenen Eigenschaften, um möglichst viele Aspekte typischer Geräte abzudecken: ein Temperatursensor, ein Display, ein sogenanntes MicroMoody mit seriell ansprechbarer RGB-LED und zwei Funkmodule (siehe Abbildung 5.3). Durch diese Auswahl können sowohl per I^2C als auch per SPI angebundene Komponenten mit verschiedenen Leistungsbereichen und Komplexitätsgraden untersucht werden. Das MicroMoody verfügt zudem über eine veränderbare Firmware, so dass die automatische Erkennung von vorgegebenem Sollverhalten evaluiert werden kann.

In diesem Abschnitt werden die einzelnen Hardwarekomponenten sowie die dafür implementierten Gerätetreiber und Modelle vorgestellt. Die Modelle beschränken sich auf Zustände, Transitionen, Parameter und Funktionsargumente, da die Energiedaten erst im Rahmen der Modellverfeinerung bestimmt werden. Bei den Gerätetreibern werden lediglich die extern nutzbaren Treiberfunktionen erwähnt, da diese den Modelltransitionen entsprechen. Weitere Details werden aus Platzgründen ausgelassen.

5.6.1 Temperatursensor

Als Beispiel für einen Sensor kommt der mit einer I^2C-Schnittstelle ausgestattete LM75B-Temperatursensor von NXP zum Einsatz. Dieser wird hier in Kombination mit einer Evaluationsplatine genutzt, welche neben dem Sensor auch die für den I^2C-Betrieb notwendigen Pull-Up-Widerstände in Höhe von 10 kΩ enthält.

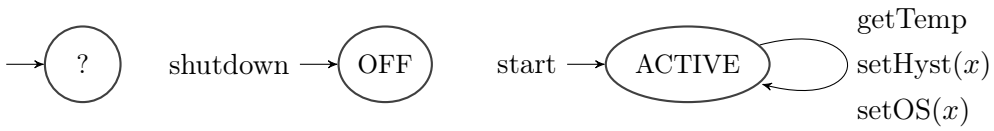

Abbildung 5.4: Treibermodell des LM75B. Die Transitionen start und shutdown können von jedem Zustand aus aufgerufen werden.

Der Sensor misst mit einer Frequenz von 10 Hz und einer Auflösung von 11 Bit bzw. 0,125 °C die Umgebungstemperatur und unterstützt einen Stromsparmodus, in dem keine Messungen stattfinden. Zudem verfügt er über einen Ausgang, welcher für temperaturabhängige Unterbrechungen genutzt werden kann. Der zugehörige Temperaturschwellwert sowie die Hysterese sind konfigurierbar [Sem15].

Das Modell nutzt den Zustand ACTIVE für Normalbetrieb und OFF für den Stromsparmodus. Ein Wechsel zwischen diesen Zuständen ist mit den Transitionen *start* bzw. *shutdown* möglich, die das 8 Bit breite Konfigurationsregister des Sensors entsprechend beschreiben. Beide Transitionen können auch zur Hardware-Initialisierung im Zustand UNINITIALIZED aufgerufen werden.

In ACTIVE kann zudem mittels *getTemp* die aktuelle Temperatur ausgelesen und durch *setOS* und *setHyst* Schwellwert und Hysterese für die temperaturabhängige Unterbrechung eingestellt werden. Diese Funktionen bearbeiten 16 Bit breite Register, welche Festkommazahlen in Zweierkomplementdarstellung beherbergen. Das obere Byte enthält Ganzzahlanteil und Vorzeichenbit, das untere die Nachkommastellen. In der Funktion getTemp wird diese Zahlendarstellung in eine Fließkommazahl umgewandelt, während setOS und setHyst nur Ganzzahlen als Argumente akzeptieren und die Nachkommastellen auf Null setzen.

Unterbrechungen werden hier nicht genutzt, da sie von der Umgebungstemperatur abhängen und deshalb aus Sicht des Energiemodells nichtdeterministisch sind. Eine grafische Darstellung des Modells zeigt Abbildung 5.4.

Als Argumente zur Testprogrammgenerierung kommen 30 und 90 °C für setOS und 29 und 60 °C für setHyst zum Einsatz. Da diese Temperaturen im Laboreinsatz nicht auftreten, werden dadurch keine Unterbrechungen ausgelöst. Globale Parameter oder sonstige Eigenheiten sind nicht vorhanden.

5.6.2 Display

Ein Display des Typs LS013B4DN04 dient als Beispiel für einen Aktuator. Dieses verfügt über 96×96 monochrome Pixel und wird hier im Rahmen einer 430BOOST-SHARP96-Adapterplatine für das MSP430 Launchpad verwendet.

Zur Ansteuerung stellt die Platine die folgenden Methoden zur Verfügung:

- einen *Enable*-Pin zum ein- und ausschalten der Bildanzeige,

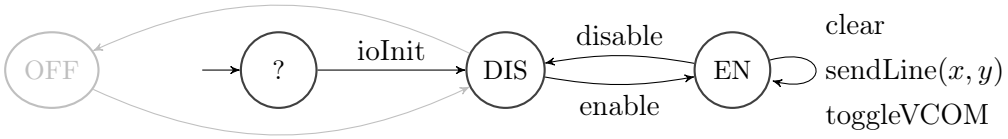

Abbildung 5.5: Treibermodell für das LS013B4DN-Display. Die ausgegrauten Zustände und Transitionen können mit MIMOSA nicht automatisch vermessen werden.

- einen *Power*-Pin für die Stromversorgung und
- eine SPI-Schnittstelle zur Übertragung von Daten und Steuerbefehlen.

Treiber und Modell basieren auf einer vorhergehenden Implementierung in einer anderen Arbeit [Fal14]. Diese unterscheidet zwischen den Zuständen OFF, DISABLED und ENABLED, wovon hier allerdings nur DISABLED und ENABLED berücksichtigt werden können. Der Zustand OFF ist nur durch Deaktivierung der Stromversorgung zum Display erreichbar, wozu MIMOSA nur mit manuellen Eingriffen in der Lage ist.

Zur Ansteuerung des Displays stehen die Funktionen *disable* und *enable* für den Wechsel zwischen den gleichnamigen Zuständen sowie *ioInit* zur Initialisierung zur Verfügung. Im ENABLED-Zustand kann zudem mittels *clear* der Bildschirminhalt gelöscht, per *sendLine(lineno, data)* eine einzelne Zeile übertragen und mit *toggleVCOM* eine Polaritätsumkehr durchgeführt werden. Eine grafische Darstellung des Modells findet sich in Abbildung 5.5.

Bei der Polaritätsumkehr handelt es sich um eine auch als *VCOM Toggle* bezeichnete Maßnahme zur Verlängerung der Lebensdauer von LC-Displays. Bei diesem Displaytyp wird zum Einschalten eines Pixels eine Spannung an einen Flüssigkristall angelegt, welche längerfristig eine Ladung im Kristall aufbaut und diesen dadurch beschädigen kann. Durch regelmäßige Umpolung der Spannung wird dies vermieden, da die Ladung bei jedem Polaritätswechsel wieder abgebaut wird. Im LS013B4DN04-Datenblatt wird dafür eine Frequenz zwischen 1 und 60 Hz empfohlen [Ame11].

Bei der Funktion sendLine werden für das Testprogramm die Zeilennummern 0, 50, 90 und 95 vorgegeben. Als Zeileninhalt kommt eine komplett inaktive Zeile mit den Nutzdaten 0x00...00 und eine komplett transparente mit 0xff...ff zum Einsatz. Globale Parameter gibt es nicht.

Als Besonderheit verlangt das Modell hier, dass im Zustand ENABLED nach jeder Transition ein toggleVCOM-Aufruf folgt, um die regelmäßig notwendige Polaritätsumkehr umzusetzen.

5.6.3 MicroMoody

Das MicroMoody ist ein als Bausatz konzipierter Mikrocontroller mit RGB-LED und I^2C-Schnittstelle zur Übertragung von Lichtbefehlen und Animationssequenzen. Es basiert auf dem von Atmel hergestellten AVR-Mikrocontroller ATTiny85.

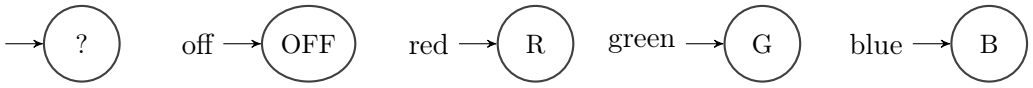

Abbildung 5.6: Nicht-parametrisiertes Treibermodell des MicroMoody. Die Transitionen off, red, green und blue können von jedem Zustand aus aufgerufen werden.

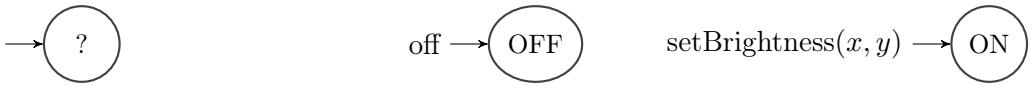

Abbildung 5.7: Parametrisiertes Treibermodell des MicroMoody. Die Transitionen off und setBrightness können von jedem Zustand aus aufgerufen werden.

Während einzelne LEDs in eingebetteten Systemen als Statusindikator verwendet werden können, zählt eine animierbare RGB-LED nicht zu den in diesem Umfeld üblichen Peripheriegeräten. Daher wird in dieser Arbeit eine angepasste Firmware verwendet, welche die Helligkeit der einzelnen Farben auf Basis des zuletzt erhaltenen Befehls einstellt und ihren Zustand nicht selbstständig verändert. So kann mit der RGB-LED als künstlichem Stromverbraucher ein weiter Bereich von Geräteverhalten simuliert werden. Zudem sind auch hier I^2C-Pull-Up-Widerstände in Höhe von 1,5 kΩ auf der Platine enthalten, so dass der energetische Einfluss dieser Widerstände bei Datenübertragungen beobachtet werden kann.

Der Treiber ist auf die Firmware abgestimmt und kann wahlweise die rote, grüne oder blaue LED ein- oder alle LEDs ausschalten. Zusätzlich kann für Rot und Grün eine Helligkeit zwischen 0 (aus) und 255 (an) eingestellt werden, welche mittels Pulsweitenmodulation umgesetzt wird. Zur Farb- und Helligkeitswahl werden drei-Byte-Pakete mit einem Befehls- und zwei Helligkeitsbytes verwendet; die Helligkeitsangaben werden von den statischen Ein- und Ausschaltbefehlen ignoriert. Die Helligkeit der blauen LED ist wegen Einschränkungen des ATTiny85 nicht einstellbar.

Um sowohl statische als auch parametrisierte Modellverfeinerung evaluieren zu können, kommen hier zwei Modelle zum Einsatz. Beide bauen auf dem gleichen Treiber auf, nutzen aber verschiedene Teilmengen der bereitgestellten Funktionen.

Das statische Modell unterstützt lediglich das Ein- und Ausschalten einzelner Farb-LEDs, wobei immer nur eine Farbe gleichzeitig aktiv ist. Es verfügt dazu über die Zustände OFF, RED, GREEN und BLUE mit gleichnamigen Transitionen, die in Abbildung 5.6 dargestellt werden. Parameter, Funktionsargumente oder Besonderheiten gibt es nicht.

Bei der parametrisierten Version werden stattdessen die Zustände ON und OFF verwendet. Nach Einstellen der Helligkeit für die rote und die grüne LED mittels *setBrightness(red, green)* wechselt das MicroMoody in den ON-Zustand, nach Aufrufen von *off* schaltet es die LEDs aus und endet in OFF. Dieses Modell ist in Abbildung 5.7 dargestellt.

Für beide Helligkeitsangaben von setBrightness werden die Werte 0, 1, 2, 16, 32, 64, 128 und 255 zur Testprogrammgenerierung verwendet. Zudem sind die eingestellten Helligkeitswerte *red* und *green* globale Parameter.

5.6.4 Funkmodul CC1200

Ein deutlich komplexeres Peripheriegerät ist der CC1200-Funkchip. Er verfügt über eine Vielzahl von Konfigurationsparametern, die sowohl den Energiebedarf als auch das funktionale Verhalten der verschiedenen Hardwaremodi beeinflussen. Zudem können einzelne Zustandswechsel je nach Zeitverlauf und gewähltem Modus sowohl explizit durch SPI-Befehle als auch implizit durch interne Ereignisse ausgelöst werden [Ins13]. Zur Kommunikation stehen eine SPI-Schnittstelle und ein Ausgang für Unterbrechungen zur Verfügung, welcher u.a. erfolgreich abgeschlossene Datenübertragungen signalisieren kann.

Hier dient wieder ein bereits in Kratos vorhandener Treiber als Grundlage. Nach Erweiterung um Konfigurationsmöglichkeiten für Sendeleistung und Bitrate, Unterbrechungen bei abgeschlossener Übertragung und zusätzliche Transitionen unterstützt er nun die folgenden Hardwarezustände.

- IDLE: Das Modul ist konfiguriert und betriebsbereit.
- SLEEP: Das Modul ist im Schlafmodus, fast alle Komponenten sind deaktiviert.
- SLEEP_EWOR: Das Modul ist im Schlafmodus mit periodischer Empfangsbereitschaft.
- XOFF: Das Modul ist eingeschränkt betriebsbereit, der Taktgeber ist deaktiviert.
- SYNTH_ON: Das Sende- und Empfangsmodul wird vorbereitet.
- RX: Empfangsbereitschaft.
- TX: Daten werden übertragen, anschließend wird eine Unterbrechung ausgelöst.

Dabei ist IDLE der Standardzustand. Hier können per *setTxPower* und *setSymbolRate* Sendeleistung und Datenrate eingestellt werden, zudem ist jeder andere Zustand von hier aus über entsprechende Transitionen direkt erreichbar. Eine grafische Darstellung der verschiedenen Zustände und Transitionen zeigt Abbildung 5.8.

Für *setTxPower* werden die Argumente 10, 20, 30, 40 und 47 vorgegeben, was den Sendeleistungen $-12{,}5$, $-7{,}5$, $-2{,}5$, $+2{,}5$ und $+6$ dBm entspricht. Bei *setSymbolRate* kommen die Datenraten 6, 12, 25, 50, 100, 200 und 250 kbit/s zum Einsatz. Die Funktion *send* wird mit einer konstanten, 127 Byte langen Zeichenkette und den Längenangaben 8, 16, 31, 32, 64 und 127 Byte aufgerufen. Sendeleistung, Datenrate und Paketlänge sind zudem globale Parameter mit den Namen *txpower*, *symbolrate* und *txbytes*.

Die Transition *init* stellt als Standardwerte eine Datenrate von 100 kbit/s bei $+6$ dBm Sendeleistung ein, was im Modell durch entsprechende Nebeneffekte ausgedrückt wird. Zusätzlich konfiguriert sie die Modulationsart der Datenübertragung auf Frequency Shift Keying mit 200 kHz Bandbreite und 50 kHz Frequenzabweichung bei einer Basisfrequenz

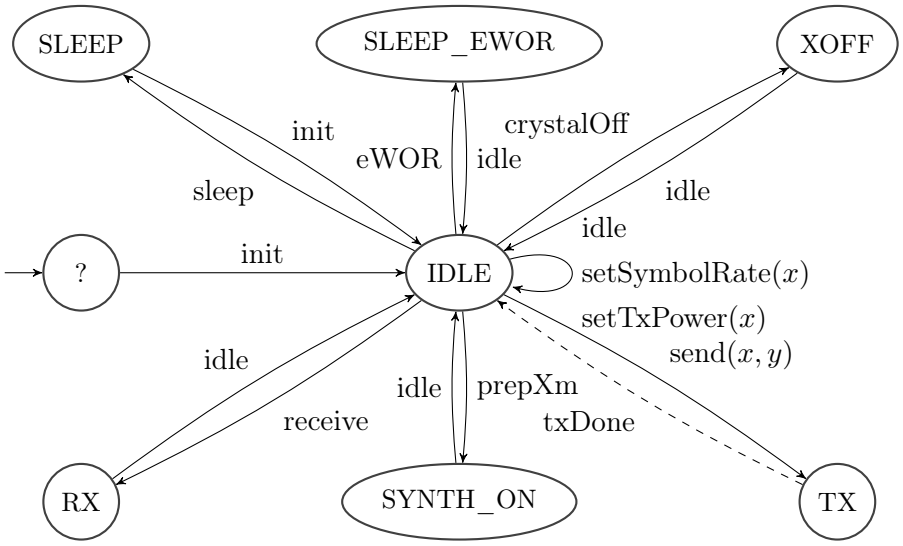

Abbildung 5.8: Treibermodell des CC1200. Die gestrichelte *txDone*-Transition wird per Unterbrechung ausgelöst.

von 868 MHz. Diese Werte werden als statisch betrachtet und weder verändert noch in das Modell aufgenommen.

5.6.5 Funkmodul nRF24L01+

Der nRF24L01+ verfügt ebenfalls über eine Vielzahl an Modi, Konfigurationsmöglichkeiten und wahlweise impliziten oder expliziten Transitionen. Zur Kommunikation stehen ebenso eine SPI-Schnittstelle und ein Ausgang für Unterbrechungen zur Verfügung. Der zugehörige Treiber basiert auf einem bereits in Kratos vorhandenen Treiber, der wiederum auf dem RF24-Projekt[4] aufbaut.

Als Besonderheit nutzt dieses Funkmodul im Normalfall ein Paketformat mit einer festen Paketlänge von 32 Byte. Zudem wird nach einer Datenübertragung automatisch bis zu 1,5 ms auf ein Antwortpaket gewartet und bei ausbleibender Antwort bis zu 15 erneute Sendeversuche unternommen. Es können aber auch dynamische Paketlängen und andere Wartezeiten oder weniger Sendeversuche konfiguriert werden.

Zur Modellierung der Hardware genügen vier Zustände: der zentrale Zustand STANDBY1, der Sendezustand TX, der Empfangszustand RX und der Stromsparzustand POWERDOWN. In STANDBY1 können mittels *setDataRate* und *setPALevel* Bitrate und Sendeleistung konfiguriert und über weitere Transitionen die restlichen Zustände direkt erreicht werden. Eine Übersicht zeigt Abbildung 5.9.

Auch hier wird für die Funktion *send* eine konstante Zeichenkette und eine variable Längenangabe genutzt. In diesem Fall ist die Zeichenkette 32 Byte lang, während als Längenangabe 8, 16 und 32 Byte verwendet werden. Für die Funktionen setPALevel und setDa-

[4] https://github.com/TMRh20/RF24

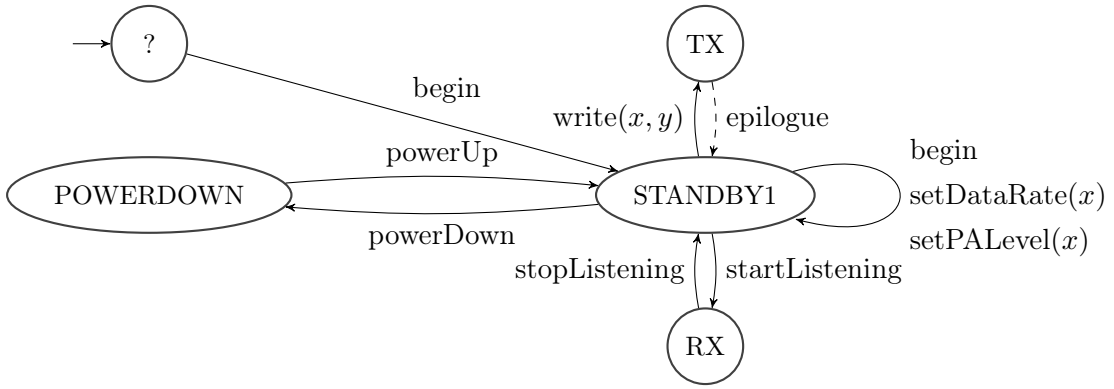

Abbildung 5.9: Treibermodell des nRF24L01+. Die gestrichelte *epilogue*-Transition wird per Unterbrechung ausgelöst.

taRate werden zur Testprogrammgenerierung die Argumente -18, -12, -6 und $+0$ dBm bzw. 250, 1000 und 2000 kbit/s eingetragen. Andere Sendeleistungen und Bitraten werden vom Funkmodul nicht unterstützt.

Die Funktion *begin* stellt Standardwerte von 1000 kbit/s und $+0$ dBm ein, was das Modell entsprechend angibt. Für den Sendezustand aktiviert sie das automatische Warten auf Antwortpakete mit oben genannter Konfiguration. Zusätzlich deaktiviert sie dynamische Paketlängen, so dass unabhängig von der angegebenen Datenlänge in send immer 32 Byte verschickt werden.

Wie beim CC1200 werden nur Sendeleistung, Bitrate und Paketlänge mit den Namen *txpower*, *datarate* und *txbytes* als globale Parameter modelliert. Das Antwort-Warteverhalten des Sendezustands und die Einstellungen zur Paketlänge werden nicht verändert und daher als statische Parameter nicht im Modell berücksichtigt.

Kapitel 6

Evaluation

Zur Evaluierung des Konzepts betrachten wir mit Hilfe der soeben vorgestellten Implementierung erstellte Energiemodelle für die dort genannten Peripheriegeräte. Diese prüfen wir anhand der zugehörigen Datenblätter und manueller Messungen auf Plausibilität, bewerten die Modellgüte und untersuchen die Funktionalität der verschiedenen automatischen Auswertungsaspekte. Als Ausgangspunkt der automatisierten Verfeinerung dienen die Automatenmodelle aus Abschnitt 5.6.

Dieses Kapitel beginnt mit einer Untersuchung der Genauigkeit von MIMOSA und des Effekts der hier entworfenen automatischen Kalibrierung. Es folgt die Analyse der erstellten statischen und parametrisierten Energiemodelle sowie speziell auf die Erkennung von Parameter-Abhängigkeiten ausgerichteter Tests. Zum Abschluss wird der zur Modellverfeinerung notwendige Zeitaufwand betrachtet.

6.1 Genauigkeit und Kalibrierung von MIMOSA

Zur Bewertung der Genauigkeit von MIMOSA und der Auswirkung der automatischen Kalibrierung wurden Kontrollmessungen mit einer programmierbaren Stromsenke mit eingebautem Messgerät durchgeführt, welche anstelle eines Peripheriegeräts an MIMOSA angeschlossen wurde. Durch Einstellen verschiedener Sollströme im gesamten Messbereich und Vergleich der MIMOSA-Daten mit denen des zweiten Messgeräts kann der Messfehler sowohl der kalibrierten als auch der unkalibrierten MIMOSA-Daten bestimmt werden.

Hierzu wurde eine Keysight N6785A Source/Measure Unit (kurz *SMU*) zusammen mit einem Keysight N6705B DC Power Analyzer verwendet. Diese unterstützt unter anderem den Betrieb als variable Stromsenke mit einstellbarer Sollfunktion und eingebautem Amperemeter, so dass automatisch ein Bereich an Sollströmen durchlaufen und gleichzeitig der tatsächlich fließende Strom aufgezeichnet werden kann.

Als Sollstrombereich kam die kleinste unterstützte Einstellung von 0 bis 4 A zum Einsatz. Hier können Ströme mit einer Genauigkeit von 0,04 % + 1,4 mA in 75 µA-Schritten

Shunt	Soll	Ist		Abweichung	
330 Ω	28 µA	188 µA	188 µA	571 %	571 %
	102 µA	229 µA	231 µA	125 %	126 %
	179 µA	276 µA	281 µA	54 %	57 %
	257 µA	328 µA	336 µA	28 %	31 %
	338 µA	386 µA	397 µA	14 %	17 %
33 Ω	98 µA	360 µA	299 µA	267 %	205 %
	178 µA	415 µA	356 µA	133 %	100 %
	253 µA	465 µA	408 µA	84 %	61 %
	406 µA	571 µA	518 µA	41 %	28 %
	481 µA	626 µA	576 µA	30 %	20 %

Tabelle 6.1: Abweichungen im unteren Messbereich. Links sind unkalibrierte, rechts kalibrierte Ist-Werte eingetragen.

eingestellt werden. Der Messbereich wurde auf 100 mA gesetzt, was eine Messgenauigkeit von 0,025 % + 10 µA ergibt.

Mit diesen Einstellungen wurde die Genauigkeit von MIMOSA für die Messwiderstände 330 Ω, 82 Ω und 33 Ω untersucht. In allen drei Fällen wurde der komplette Messbereich (bis 5 mA, bis 20 mA und bis 50 mA) zweimal linear durchlaufen und neben den von der SMU gemessenen Sollströmen auch die unkalibrierten und kalibrierten Ist-Ströme von MIMOSA aufgezeichnet. Die dazu verwendeten Kalibrierungsdaten wurden aus zuvor durchgeführten automatischen Kalibrierungsläufen entnommen.

Die Messergebnisse für 330 und 33 Ω sind in Abbildung 6.1 und Tabelle 6.1 aufgezeichnet. Die Graphen zeigen die prozentuale Abweichung der Ist-Werte von MIMOSA von den Soll-Werten der SMU, während die Tabelle sowohl absolute als auch relative Daten für den in den Graphen nicht sichtbaren unteren Messbereich enthält.

Es zeigt sich, dass die Abweichungen von MIMOSA deutlich weniger linear sind als beim Entwurf der Kalibrierungsroutine angenommen. Vielmehr ist eine Kombination aus zwei linearen Messfehlern erkennbar – einer mit hoher Steigung in den unteren 20 % des Messbereichs und ein näherungsweise konstanter im restlichen Bereich. Insbesondere gibt MIMOSA für sehr niedrige Ströme deutlich zu hohe Werte an, während im restlichen Bereich eine Abweichung nach unten vorliegt.

Die Kalibrierung ist eingeschränkt in der Lage, diese Abweichungen auszugleichen. Beim 330 Ω-Messwiderstand liegt der Messfehler für kalibrierte Werte oberhalb von 1 mA bei unter 0,25 %; bei Verwendung des 33 Ω-Shunts ergibt sich bei Strömen ab 2 mA ein Fehler von weniger als 1 % – die Abweichung ist also um den Faktor fünf bis zehn geringer als bei unkalibrierten Messwerten. In diesem Wertebereich kann dank der Kalibrierung zuverlässig und hinreichend akkurat gemessen werden.

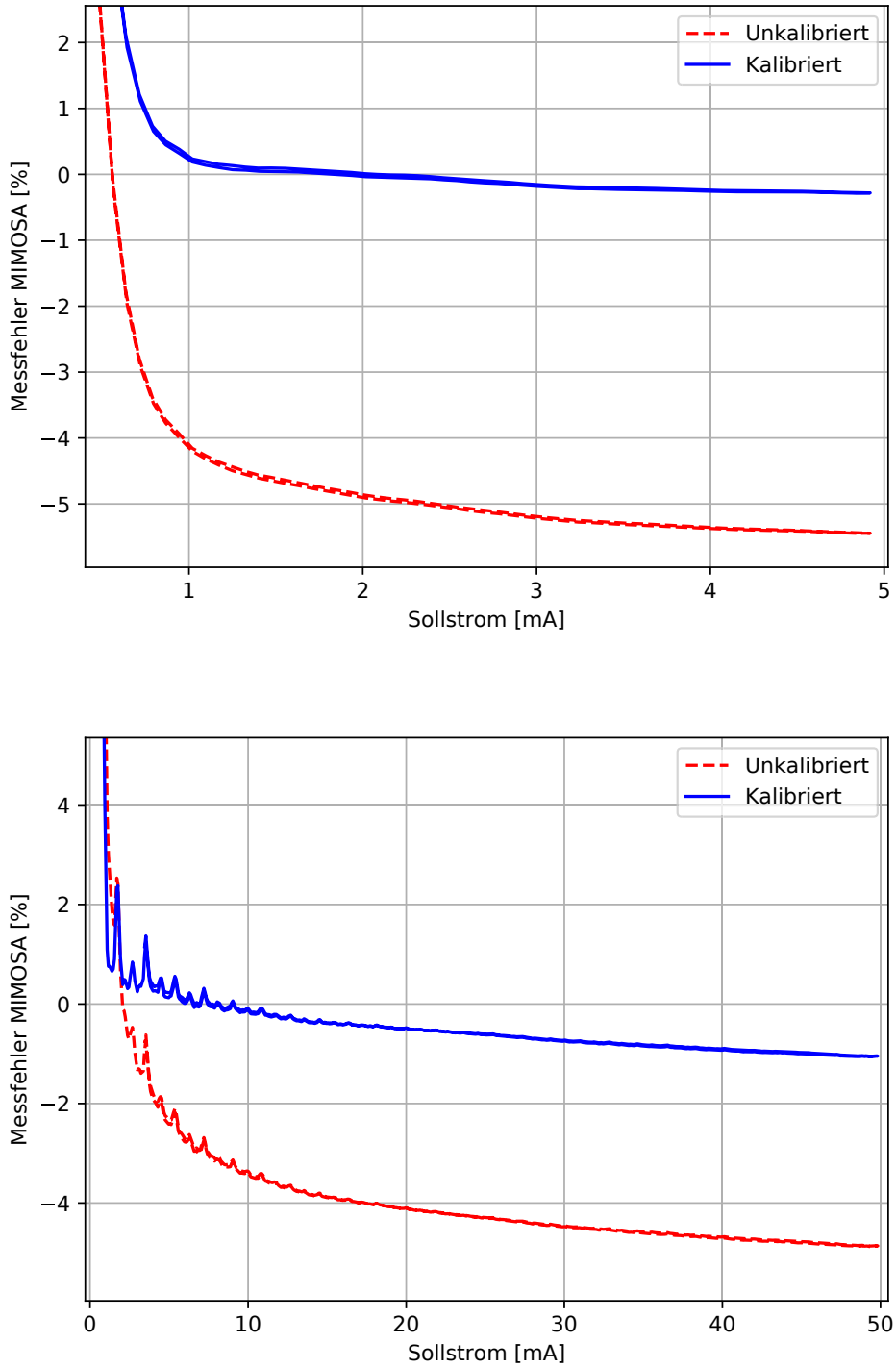

Abbildung 6.1: Messfehler der kalibrierten und unkalibrierten MIMOSA-Daten für die Messwiderstände 330 Ω (oben) und 33 Ω (unten).

Unterhalb der oben genannten Grenze ist die Abweichung deutlich höher. Für den 330 Ω-Widerstand liegt der Fehler der kalibrierten Daten sogar geringfügig über dem der unkalibrierten, bei 33 Ω ist er lediglich um ein Viertel niedriger und damit immer noch sehr hoch. Die kalibrierten Daten weisen hier eine Abweichung von bis zu 250 µA auf. Modellwerte, die Leistungen von weniger als 3,6 bzw. 7,2 mW angeben, müssen also mit Vorsicht betrachtet und um bis zu 900 µW nach unten korrigiert werden.

Der Grund für diesen hohen Fehler im unteren Messbereich konnte im Rahmen dieser Arbeit nicht eindeutig identifiziert werden. Die derzeit naheliegendste Vermutung ist, dass es sich um ein Softwareproblem in libmimosa oder ein Firmwareproblem in MIMOSA selbst handelt, welches erst nach der Veröffentlichung von [BGS13] bei der Portierung auf neue Hardware auftrat. Diese Annahme folgt aus einer untypischen Verteilung der von MIMOSA bzw. libmimosa ausgegebenen Rohdaten.

In der Praxis liefern auch hochpräzise Messgeräte bei Vermessung eines konstanten Stroms nicht permanent denselben Messwert, sondern eine Folge verschiedener Messwerte, deren Mittelwert den tatsächlichen Strom ergibt. Der Grund dafür ist, dass die von den Komponenten des Messgeräts aufbereiteten Messdaten durch ein Grundrauschen überlagert werden. Dieses folgt üblicherweise einer Normalverteilung mit Streuung $\sigma \neq 0$, was sich in den Messwerten widerspiegelt.

Eine Betrachtung der Messdaten als Histogramm (siehe Abbildung 6.2) zeigt, dass die von der SMU aufgezeichneten Daten wie erwartet normalverteilt sind. Die Daten von MIMOSA verhalten sich jedoch anders: Hier treten nur sehr wenige Messwerte mit großen Lücken dazwischen auf, die keiner statistischen Verteilung zu folgen scheinen.

Eine Betrachtung der zugehörigen Rohdaten offenbart eine weitere Auffälligkeit. Insbesondere im unteren Messbereich weisen diese eine Häufung von Zahlen auf, deren Binärdarstellung auf eine Folge von sechs bis acht 1-Bits endet. Eine Messung ohne Stromfluss ergibt sogar ausschließlich die Rohdaten 63 (00111111), 127 (01111111), 191 (10111111) und 255 (11111111), welche den vier Säulen des unteren Histogramms in Abbildung 6.2 entsprechen. Ein derartig systematischer Fehler ist ein deutliches Indiz für Soft- oder Firmwareprobleme.

Die Auswirkung dieser Eigenart auf die Evaluierung der Modellverfeinerung ist erfreulicherweise gering. Zwar sind Modellwerte im unteren Messbereich wegen der hohen Abweichung nur bedingt zur Konstruktion praktisch nutzbarer Energiemodelle geeignet. Da sich der Mittelwert der Messwerte aber, abgesehen von diesen Abweichungen, wie erwartet verhält und insbesondere bei zu verschiedenen Zeitpunkten durchgeführten Messungen mit den gleichen Strömen nicht schwankt, sind die Messergebnisse von MIMOSA dennoch reproduzierbar. Ein Vergleich mit manuell durchgeführten Messungen ist daher problemlos möglich, bei Vergleichen mit Datenblättern und Berechnungen muss lediglich im unteren Messbereich die Abweichung beachtet werden. Die Funktion der verschiedenen Auswertungsmethoden wird hiervon nicht beeinträchtigt.

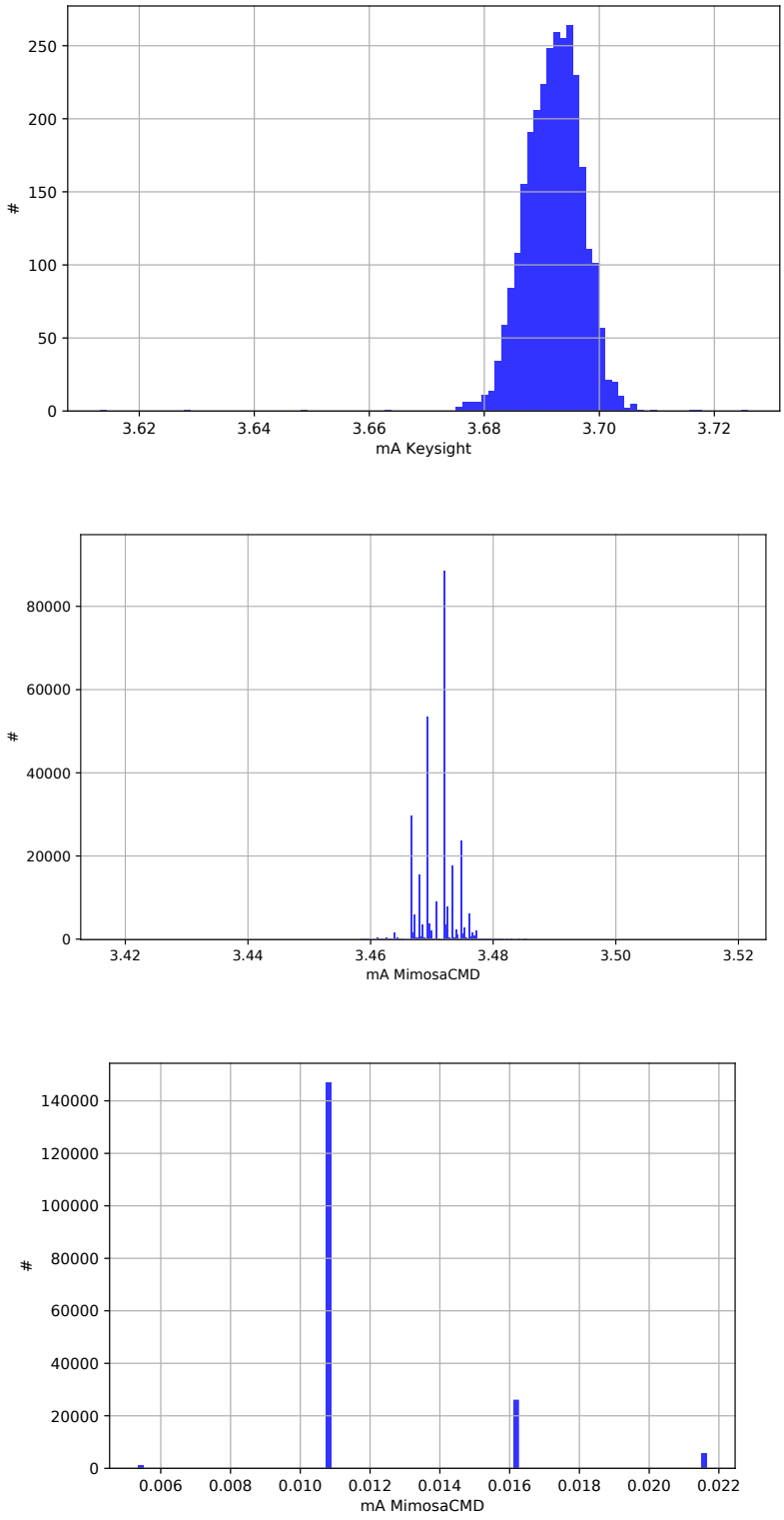

Abbildung 6.2: Messwerthistogramme für die Keysight N6785A SMU (oben) und MIMOSA (mittig und unten) bei verschiedenen, konstanten Sollströmen. Alle MIMOSA-Daten wurden mit dem 330 Ω-Messwiderstand erhoben.

6.2 Statische Modelle

Als nächstes betrachten wir die automatisch erstellten statischen Modelle der verschiedenen Peripheriegeräte. Deren Angaben zu Zustandsleistung, Transitionsenergie und Transitionsdauer werden anhand von Datenblättern, manuell durchgeführten Messungen und den Gütemaßen MAE und SMAPE auf Plausibilität geprüft. Ein Vergleich zwischen relativer und absoluter Transitionsenergie erfolgt erst in Abschnitt 6.3.

6.2.1 Temperatursensor

Da der LM75B-Temperatursensor nur über wenige Transitionen und Zustände verfügt, wurden bei der Testprogrammgenerierung für jedes Wort bis zu drei Besuche im gleichen Zustand und eine Aufenthaltsdauer von 250 ms eingestellt. Zur Messung wurde der 680 Ω-Shunt und somit der kleinste mögliche Messbereich verwendet. Von insgesamt acht Messläufen konnten sechs ausgewertet werden; die anderen beiden enthielten Synchronisierungsfehler, welche vermutlich durch einen schlechten Kontakt am Buzzer-Pin entstanden.

Tabelle 6.2 zeigt die daraus bestimmten Modelldaten. Für jeden Zustand ist die modellierte Leistung $\tilde{P}$ mit absoluter Abweichung (MAE und SMAPE) dieses Modellwerts und Anzahl n der ausgewerteten Messungen angegeben. Jede Transition enthält Angaben über absolute und relative Energie $\tilde{E}$ bzw. $\widetilde{\Delta E}$, Dauer $\tilde{t}$, die zugehörigen Abweichungen und die Anzahl ausgewerteter Messungen.

Bei beiden Zuständen liegt die mittlere absolute Abweichung der modellierten Leistung unter 3 µW, bei POWEROFF sogar unter 1 µW. Dies spricht dafür, dass die Zustände korrekt vermessen und ausgewertet wurden, und zeigt durch den niedrigen Modellfehler zudem, dass der Sensor mit den getesteten Transitionen tatsächlich nur diese beiden Zustände erreicht. Die geringfügig erhöhte Abweichung in ACTIVE wird von den periodischen Temperaturmessungen des Sensors verursacht – da diese alle 100 ms stattfinden, enthalten manche Daten des ACTIVE-Zustands zwei und manche drei solcher Messungen. Eine Aufnahme der Temperaturmessung in das Modell ist aber nicht möglich, da ihre Durchführung abgesehen vom Stromverbrauch nicht extern beobachtet werden kann.

Auch das Datenblatt bestätigt die Korrektheit der Modelldaten. Hier werden typischerweise 100 bis maximal 200 µA Stromverbrauch für ACTIVE und 0,2 bis 1 µA für POWEROFF angegeben [Sem15]. Unter Berücksichtigung der zuvor festgestellten Abweichungen im unteren Messbereich von MIMOSA entsprechen die ermittelten Modellwerte von 92,5 und 1,9 µA diesen Angaben. Dies wird auch durch eine manuell durchgeführte Messung und Auswertung mit den gleichen Ergebnissen unterstützt.

Für die Transitionen liegen keine Herstellerangaben vor. Die Werte sind aber auch hier plausibel, wie die folgenden Berechnungen zeigen.

Die Hauptaufgabe der Treiberfunktionen ist die Kommunikation mit dem Temperatursensor über die I^2C-Schnittstelle. Diese wird vom DCO des MSP430 mit einem Takt

Zustand	$\tilde{P}$	MAE	SMAPE	n
ACTIVE	332 µW	2,53 µW	0,76 %	3510
POWEROFF	7,1 µW	0,69 µW	8,9 %	3090

Transition	$\tilde{E}$		$\widetilde{\Delta E}$		$\tilde{t}$		n
getTemp		26,0 µJ		21,7 µJ		12,7 ms	360
	1,97 µJ	8,2 %	1,95 µJ	9,9 %	2,39 µs	0,02 %	
setHyst		22,1 µJ		19,0 µJ		9,1 ms	720
	1,95 µJ	9,8 %	1,93 µJ	11 %	1,49 µs	0,02 %	
setOS		21,8 µJ		18,7 µJ		9,1 ms	720
	1,95 µJ	9,9 %	1,94 µJ	12 %	1,56 µs	0,02 %	
shutdown		11,8 µJ		11,7 µJ		7,0 ms	3960
	1,12 µJ	8,3 %	0,45 µJ	3,6 %	2,58 µs	0,04 %	
start		12,4 µJ		12,4 µJ		7,0 ms	2160
	0,74 µJ	5 %	0,41 µJ	3 %	2,49 µs	0,04 %	

Tabelle 6.2: Für den LM75B erstellte statische Modellwerte und zugehörige Güteangaben. Die Felder der Transitionstabelle geben in der ersten Zeile den Modellwert und in der zweiten Zeile MAE und SMAPE an.

von 4 kHz versorgt, so dass das Übertragen eines einzelnen Bits 250 µs ± 3,5 % benötigt. Pro Adress- oder Datenbyte ergibt dies mit dem zugehörigen Ack-Bit eine Dauer von etwa 2,25 ms.

Die Transitionen *start* und *shutdown* übertragen inklusive der Adresse drei Bytes, *setOS* und *setHyst* vier und *getTemp* fünf. Die reine Übertragungsdauer beträgt also 6,75 , 9 und 11,25 ms und weicht bis auf getTemp um weniger als 3,6 % von den Modellwerten ab; diese Abweichung kann durch die Ungenauigkeit des DCO-Takts und den in der Berechnung nicht berücksichtigten zusätzlichen Zeitaufwand für die Start- und Stopp-Signale der I^2C-Kommunikation erklärt werden. Die höhere Dauer in getTemp folgt hingegen aus der Umrechnung der ausgelesenen Temperatur in eine Fließkommazahl. Die Zeitangaben sind somit allesamt plausibel, wofür auch ihre niedrige SMAPE von unter 0,05 % spricht.

Bei der Transitionsenergie fällt auf, dass sie einerseits sehr hoch ist, und andererseits die Messdaten im Mittel um etwa 10 % vom daraus erstellten Modellwert abweichen – allerdings nur bei Transitionen, die mit Temperaturdaten arbeiten. Hier schwankt der Energiebedarf der einzelnen Transitionen also erheblich.

Grund dafür sind die Pull-Up-Widerstände an den I^2C-Datenleitungen, die hier ebenfalls von MIMOSA versorgt werden. Beim Übertragen einer Null verursachen diese einen Stromfluss von 360 µA je Widerstand, so dass im Mittel 648 µW auf das Taktsignal (welches während der Hälfte der Übertragungszeit Null ist) und weitere 0 bis 1,3 mW auf die Daten-

leitung entfallen. Für die Transitionen start und shutdown ergibt dies einen Energiebedarf von 4,5 bis 13,6 µJ, was mit den Modellwerten übereinstimmt. Für setOS und setHyst ergeben sich unter Berücksichtigung der Leistung des ACTIVE-Zustands zwischen 8,9 und 20,7 µJ, für getTemp 12,4 bis 29,0 µJ. Auch dies stimmt im Rahmen der Messgenauigkeit mit den Modelldaten überein.

Die verschiedenen mittleren Abweichungen der einzelnen Transitionen werden ebenfalls durch die Pull-Up-Widerstände verursacht. Die Funktionen start und shutdown übertragen immer dieselben Daten, so dass auch der Einfluss der Pull-Ups kaum variiert und die mittlere Energie-Abweichung niedrig ist. setOS und setHyst hingegen übertragen verschiedene Schwellwerttemperaturen, deren Binärkodierung verschieden viele Null-Bits enthält – der Energiebedarf hängt also von den Funktionsargumenten ab und ist entsprechend schlecht statisch vorhersagbar. Selbiges gilt auch für getTemp, nur dass die Variationen hier durch Schwankungen in der Raumtemperatur verursacht werden.

An dieser Stelle zeigt sich, dass Modelleigenschaften neben globalen Parametern auch von den gerade verwendeten Funktionsargumenten abhängen können. Eine dahingehende Erweiterung des Modells ist daher ein lohnendes Ziel für eine weitere, auf diesem Konzept aufbauende Arbeit.

Der Transitionsenergiebedarf des LM75B selbst ist verglichen mit den Effekten der Pull-Up-Widerstände gering und daher in den Modellwerten nicht erkennbar. Dies konnte durch eine manuelle Analyse einzelner Transitionen bestätigt werden: Zu Zeitpunkten, an denen auf SDA und SCL eine Eins übertragen wird (d.h. die Pull-Ups nicht aktiv sind), weicht die Leistungsaufnahme um weniger als 20 % von der des vorherigen Zustands ab.

Insgesamt können wir festhalten, dass die automatisierte Modellverfeinerung zu einer einwandfreien Modellierung des LM75B in der Lage ist. Lediglich die modellierte Transitionsenergie ist merkbar ungenau, was auf einer Einschränkung des verwendeten Modellkonzepts beruht.

6.2.2 Display

Auch das Display wurde mit 680 Ω-Shunt, bis zu drei Zustandsbesuchen und 200 ms Aufenthaltszeit vermessen. Jeder der acht durchgeführten Messläufe konnte ausgewertet werden.

Die erzeugten Modelldaten zeigt Tabelle 6.3. Für die Transitionen sind hier zusätzlich die korrespondierenden Transitionsnamen aus [Fal14] angegeben, in der ebenfalls ein Energiemodell für dieses Display erstellt wurde.

Im Datenblatt für das Display finden sich nur Werte für den ENABLED-Zustand mit periodischen Zusatztransitionen. Dort werden im Mittel 6 µW für einen toggleVCOM-Aufruf je Sekunde und 12 µW für zusätzlich eine vollständige Bildaktualisierung je Sekunde angegeben [Ame11]. Abhängig davon, wie weit die Leistung des reinen ENABLED-Zustands ohne periodische Transitionen unterhalb von 6 µW liegt, ergeben sich Abweichun-

Zustand	$\tilde{P}$			n
DISABLED	22,1 µW	0,669 µW	2,9 %	1144
ENABLED	24,2 µW	0,733 µW	2,9 %	5800

Transition	$\tilde{E}$		$\widetilde{\Delta E}$		$\tilde{t}$		n
clear		14,1 nJ		13,3 nJ		30 µs	176
(ENA2ENA_CLEAR)	1,25 nJ	8,3 %	1,22 nJ	8,6 %	0,9 µs	2,4 %	
disable		0 nJ		0 nJ		0 µs	1136
(ENA2DISA)	1,17 nJ		1,13 nJ		1,4 µs		
enable		0 nJ		0 nJ		0 µs	1144
(DISA2ENA)	1,23 nJ		1,2 nJ		1,5 µs		
ioInit		0 nJ		0 nJ		0 µs	968
(ENTRY)	0,16 nJ		0,13 nJ		2,4 µs		
sendLine		37,9 nJ		33,4 nJ		180 µs	1408
(ENA2ENA_LINE)	1,95 nJ	5,1 %	1,95 nJ	5,8 %	0,5 µs	0,3 %	
toggleVCOM		31 nJ		30,2 nJ		30 µs	3080
(ENA2ENA_CLEAR)	2,18 nJ	9,2 %	2,16 nJ	9,5 %	1,0 µs	2,9 %	

Tabelle 6.3: Energiemodell des LS013B4DN04 LC-Displays und zugehörige Güteangaben.

gen von mindestens 400 % zwischen Modell- und Datenblattwert, was in diesem Messbereich allerdings im Rahmen der erwarteten Messfehler liegt. Hierfür sprechen auch die mit einem MIMOSA-Prototypen ermittelten Ergebnisse aus [Fal14], welche im Mittel 18,7 µW für DISABLED und 19,3 µW für ENABLED angeben – also sehr ähnliche Werte zu den hier automatisch bestimmten.

Bei der Transitionsenergie zeigt sich ein gegenteiliges Bild. Aus dem Datenblatt lässt sich ein Energiebetrag von $6\,\mu W \cdot 1\,s \cdot \frac{1}{96} = 62{,}5\,nJ$ je aktualisierter Zeile berechnen. Die hier gemessene Transitionsenergie von 37,9 bzw. 33,4 nJ ist zwar nur um den Faktor zwei niedriger als dieser Wert, sollte wegen der Ungenauigkeiten von MIMOSA aber höher sein. Die korrespondierende Angabe in [Fal14] beträgt dagegen nur 3,8 nJ.

Eine manuelle Analyse der Messwerte zeigt, dass die für sendLine modellierte Energie die von MIMOSA gemessenen Daten korrekt widerspiegelt und somit kein Fehler in der automatischen Auswertung ist. Abgesehen von möglicherweise fehlerhaften Datenblattangaben ist eine naheliegende Begründung dafür, dass das Display einen Teil seines Energiebedarfs durch die Signalleitungen der SPI-Schnittstelle deckt – dieser kann von MIMOSA nicht erfasst werden.

Die Chip Select-Leitung ist hier als *Active High* spezifiziert, d.h. wann immer das Display angesprochen werden soll, liegt dort eine Spannung von etwa 3,6 V an. Es ist möglich, dass das Display dadurch nicht nur Daten entgegennimmt, sondern auch einen Teil

seiner Energie über diesen Eingang bezieht – dieses Verhalten wird als *Querversorgung* bezeichnet. Diese Vermutung wurde durch eine Untersuchung des Displays ohne Stromversorgung bestätigt: Trotz fehlender Versorgungsspannung war es zuverlässig in der Lage, Daten entgegenzunehmen und anzuzeigen. Es ist daher naheliegend, dass die gemessenen Energiewerte der Transitionen wegen Querversorgung durch die Datenleitungen zu niedrig sind.

Aus diesem Grund wird hier auf eine weitere Analyse der Transitionsenergiewerte verzichtet. Dies gilt auch für die mit 0 nJ angegebenen Transitionen *enable*, *disable* und *ioInit* – diese bestehen ausschließlich aus dem Umschalten von GPIO-Pins und sind daher so kurz, dass die Buzzer-Auflösung von 10 µs nicht ausreicht, um ihre tatsächliche Dauer und damit auch die Energie zu bestimmen. Wegen dieser Einschränkung fehlt auch die prozentuale Abweichung der Transitionseigenschaften, da sie für Modell- und Messwerte von Null nicht definiert ist.

Die Zeitangaben der Transitionen sind wiederum im Rahmen der Erwartungen. *clear* und *toggleVCOM* übertragen drei Byte bei einer Frequenz von 1 MHz, was einer Übertragungsdauer von mindestens 24 µs entspricht, während die von *sendLine* übertragenen 15 Byte mindestens 120 µs zur Übertragung und einige Zusatzzeit zur Formatierung der Zeilennummer benötigen. Dies ist mit den Modellwerten von 30 und 180 µs konsistent. Auch hier spricht der niedrige Modellfehler für die Korrektheit der ermittelten Werte. Die Abweichung von 3 % bei clear und toggleVCOM folgt in erster Linie aus der verhältnismäßig niedrigen zeitlichen Auflösung von 10 µs.

Im Vergleich mit [Fal14] ist die Transitionsdauer hier konstant um etwa 120 µs niedriger. Dies liegt voraussichtlich daran, dass hier die Zeiten zur Aktualisierung des Onlinemodells nicht in die Transitionsdauer einfließen.

6.2.3 MicroMoody

Das MicroMoody mit nicht-parametrisierter Firmware wurde in 14 Messläufen mit 33 Ω-Shunt vermessen, von denen alle Läufe auswertbar waren. Wegen der starken Vernetzung des DFAs (von jedem Zustand aus ist jeder andere erreichbar) wurde pro Wort nur ein einziger Besuch jedes Zustands erlaubt. Die Verweilzeit in jedem Zustand betrug 500 ms.

Tabelle 6.4 zeigt die Ergebnisse der Modellverfeinerung. Diese werden lediglich mit dem in der Firmware vorgegebenen Verhalten des MicroMoody verglichen, da für die LED und den Bausatz als Ganzes keine Datenblätter vorliegen.

Der sehr niedrige Fehler von Zustandsleistung und Transitionsdauer zeigt, dass das in der Firmware und dem Automatenmodell abgebildete Verhalten korrekt vermessen wurde. Auch die identischen Leistungswerte der Zustände B und G sind korrekt, da blaue und grüne LEDs häufig ein ähnliches Energieverhalten haben. Rote LEDs setzen bei glei-

Zustand	$\tilde{P}$			n
B	29,4 mW	12,0 µW	0,04 %	238
G	29,4 mW	11,4 µW	0,04 %	238
OFF	7,06 mW	0,19 µW	<0,01 %	238
R	49,1 mW	72,2 µW	0,15 %	252

Transition	$\tilde{E}$		$\widetilde{\Delta E}$		$\tilde{t}$		n
blue		374 µJ		106 µJ		9,14 ms	322
	129 µJ	38 %	4,52 µJ	4,2 %	3,79 µs	0,04 %	
green		372 µJ		103 µJ		9,14 ms	322
	129 µJ	38 %	4,48 µJ	4,3 %	3,91 µs	0,04 %	
off		373 µJ		104 µJ		9,14 ms	322
	61,1 µJ	13 %	1,7 µJ	1,7 %	3,76 µs	0,04 %	
red		379 µJ		110 µJ		9,14 ms	322
	59,9 µJ	21 %	2,68 µJ	2,4 %	3,54 µs	0,04 %	

Tabelle 6.4: Modellwerte der unparametrisierten MicroMoody-Variante.

cher Spannung hingegen mehr Strom um, was die deutlich höhere Leistungsaufnahme im Zustand R erklärt.

Wie schon beim LM75B stimmen die Transitionszeiten mit den theoretischen Werten überein. Jeder MicroMoody-Befehl besteht aus einem Adress- und drei Datenbytes, was zusammen mit den Ack-Bits 36 Bit bzw. 9 ms je Transition entspricht. Die Transitionsenergie ist auch hier wegen der von MIMOSA versorgten Pull-Up-Widerstände sehr hoch, die Abweichung der relativen Energie mit unter 5 % aber akzeptabel. Erläuterungen zur bemerkenswert hohen Abweichung der absoluten Energie folgen erst in Abschnitt 6.3, es kann aber bereits vorweggenommen werden, dass es sich dabei nicht um einen Modell-, Konzept- oder Implementierungsfehler handelt.

Schließlich zeigt das MicroMoody auch den Nutzen der deterministischen Testprogrammgenerierung. Bei ersten Auswertungen wies das Energiemodell einen Fehler von 4,5 % in den Zuständen B und G auf, während sich OFF und R wie in Tabelle 6.4 verhielten. Eine Betrachtung der nach einzelnen Wörtern (d.h. einzelnen Folgen von Funktionsaufrufen) aufgeschlüsselten Messwerte zeigte, dass in etwa der Hälfte der Testläufe die erste Transitionsfolge (? $\xrightarrow{\text{blue}}$ B $\xrightarrow{\text{green}}$ G $\xrightarrow{\text{off}}$ OFF) unübliches Verhalten aufwies: Die Leistungsaufnahme aller Zustände war mit OFF identisch. Erst ab dem zweiten Wort war das Verhalten mit den restlichen Transitionsfolgen konsistent.

Die Firmware enthält also einen Fehler, wegen dessen nach der Kalibrierung von MIMOSA und dem damit einhergehenden kurzzeitigen Trennen des MicroMoody von der Spannungsversorgung in manchen Fällen zunächst keine I^2C-Befehle ausgewertet werden.

Durch Auslassen dieser Transitionsfolge bei der Modellverfeinerung konnte der Fehler umgangen und ein korrektes Energiemodell erstellt werden. Bei nichtdeterministischer Testprogrammgenerierung wäre das Fehlerbild im Modell weniger eindeutig und damit auch schwieriger nachverfolgbar gewesen.

6.2.4 Funkmodul CC1200

Für den CC1200 wurden wegen der komplexen Hardware drei verschiedene Arten von Messungen durchgeführt. Neben einer Messung ohne Einschränkungen mit maximal zwei Besuchen je Zustand waren dies zwei Messungen mit maximal vier wiederholten Zustandsbesuchen und Einschränkung auf Transitionsfolgen der Art init · setSymbolRate · setTxPower · send · txDone · sleep bzw. init · setSymbolRate · setTxPower · receive · idle · sleep.

Die Aufteilung in verschiedene Messungen ist notwendig, da mit nur zwei oder drei maximalen Zustandsbesuchen die Zustände TX und RX nicht mit allen Parameterkombinationen vermessen und deren Parameterabhängigkeit somit nicht beurteilt werden könnte. Eine uneingeschränkte Messung mit maximal vierfachem Aufenthalt im gleichen Zustand hätte aber knapp drei Tage je Testlauf benötigt, da $L_4(\mathcal{A})$ hier 152064 verschiedene Wörter enthält und somit 152064 Transitionsfolgen im Testprogramm abgelaufen werden müssten.

Insgesamt wurden 47 Messläufe für die drei verschiedenen Messungen durchgeführt, von denen 36 auswertbar waren. Dabei wurden der 33 Ω-Messwiderstand und eine Zustands-Aufenthaltszeit von 200 ms verwendet. Die Ergebnisse zeigt Tabelle 6.5.

Die Leistungsdaten der Zustände entsprechen bis auf SLEEP und SLEEP_EWOR den Datenblattangaben [Ins13]. Die Werte dieser Zustände sind höher als erwartet, was der oberflächlichen Implementierung im Treiber geschuldet ist: Die zugehörigen Funktionen versetzen das Funkmodul zwar in den jeweiligen Schlafzustand, stellen vorher aber nicht die notwendigen Konfigurationsparameter ein, so dass die im Datenblatt angegebene niedrige Leistungsaufnahme nicht erreicht wird.

Für die Zustände RX und TX wird mit $\frac{\overline{\sigma_P}}{\sigma_P} = 0{,}034$ bzw. 0,013 (vgl. Abschnitt 4.5.2) eine Parameter-Abhängigkeit der Leistung festgestellt, was den hohen Fehler ihrer statischen Leistungsangaben erklärt. Dies wird auch im Boxplot der Leistungsdaten in Abbildung 6.3 sichtbar. Wegen der vielen Messwerte mit verschiedenen Parametern und daraus folgenden Leistungswerten hat der TX-Zustand einen hohen Interquartilsabstand, es treten jedoch keinerlei Ausreißer auf, die sonst zur Erklärung der Abweichung im statischen Modell herhalten könnten.

Im Zustand SLEEP liegt gemäß Heuristik keine solche Abhängigkeit vor ($\frac{\overline{\sigma_P}}{\sigma_P} = 1$), auch bei manueller Analyse der Messdaten findet sich kein Zusammenhang zu Parametern oder vorhergehenden Transitionsfolgen. Die Abweichung hier kann folglich nur durch nicht modellierbare Schwankungen im Hardwareverhalten erklärt werden.

Zustand	$\tilde{P}$			n
IDLE	9,5 mW	15,6 µW	0,16 %	17480
RX*	85,2 mW	245 µW	0,29 %	615
SLEEP	143 µW	10,6 µW	7,1 %	55
SLEEP_EWOR	81,8 mW	7,46 µW	0,01 %	55
SYNTH_ON	60 mW	12,9 µW	0,02 %	55
TX*	92,5 mW	11366 µW	12 %	4200
XOFF	780 µW	1,32 µW	0,17 %	55

Transition	$\tilde{E}$		$\widetilde{\Delta E}$		$\tilde{t}$		n
crystalOff		115 nJ		-62,1 nJ		20 µs	175
	22,6 nJ	24 %	15,8 nJ	30 %	3,31 µs	22 %	
eWOR		318 nJ		-80,9 nJ		40 µs	175
	24,6 nJ	7,4 %	17 nJ	21 %	3,26 µs	7,2 %	
idle		718 nJ		-547 nJ		20 µs	835
	284 nJ	44 %	234 nJ	54 %	3,87 µs	26 %	
init		23 µJ		22,3 µJ		4,98 ms	5085
	34,9 µJ	13 %	54,9 µJ	45 %	4,92 ms	0,29 %	
prepXm		379 nJ		189 nJ		20 µs	175
	95,1 nJ	33 %	62,6 nJ	45 %	3,43 µs	23 %	
receive		380 nJ		190 nJ		20 µs	735
	95,5 nJ	33 %	60,6 nJ	41 %	3,67 µs	24 %	
send		4,3 µJ		454 nJ		400 µs	4200
	17,5 µJ	51 %	403 nJ	31 %	1,8 ms	53 %	
setSymbolRate		1,0 µJ		15,4 nJ		100 µs	4095
	20,4 µJ	3,8 %	379 nJ	34 %	2,11 ms	3,5 %	
setTxPower		0,29 µJ		3,73 nJ		30 µs	3985
	2,55 µJ	5,2 %	50,5 nJ	75 %	262 µs	4,2 %	
sleep		0,10 µJ		-68,6 nJ		20 µs	3885
	13,3 µJ	27 %	266 nJ	31 %	1,37 ms	25 %	
txDone		0 nJ		0 nJ		0 µs	4200
	10,7 nJ		95,6 nJ		1,11 µs		

Tabelle 6.5: Statische Modellwerte des CC1200. Die Leistung der mit $\star$ markierten Zustände hängt von globalen Parametern ab und ist daher nur schlecht statisch modellierbar.

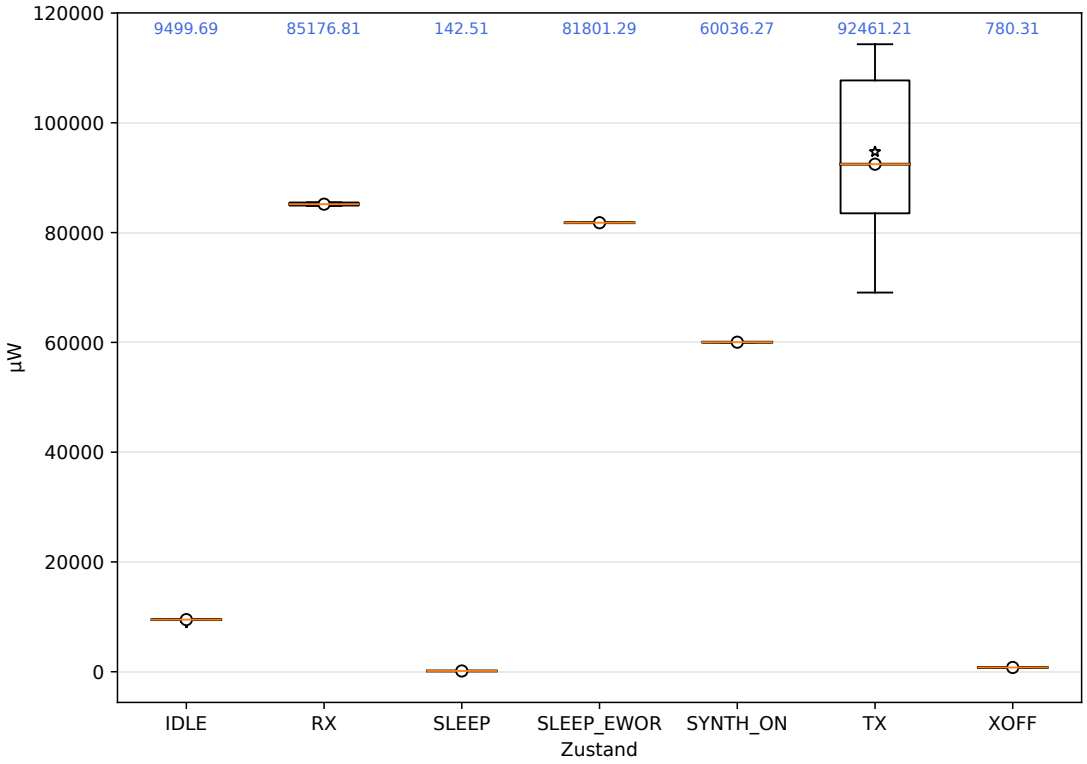

Abbildung 6.3: Leistungsangaben der CC1200-Zustände als Boxplot. Der Median der Daten ist als Kreis eingezeichnet, der Mittelwert als Stern.

Bei den Transitionen zeigen sich einige Auffälligkeiten. Die hohen Fehler bei Zeitangaben für Transitionen mit einer Dauer im unteren µs-Bereich folgen aus der geringen Zeitauflösung von MIMOSA, während die Dauer von *send* von der Variation der Paketlänge und der damit einhergehenden unterschiedlichen Übertragungsdauer auf der SPI-Schnittstelle beeinflusst wird. Bei *init*, *sleep*, *setTxPower* und *setSymbolRate* liegen Ausreißer durch fehlerhaftes Verhalten des Funkmoduls vor.

Die mittleren Fehler der Transitionsenergie begründen sich analog, da die während einer Transition umgesetzte Energie maßgeblich von ihrer Dauer abhängt. Die Abweichungen von init folgen auch aus dem dabei durchgeführten Zurücksetzen des Funkmoduls, was zwischenzeitlich undefiniertes Verhalten und daher viele Ausreißer zur Folge hat.

Eine Parameterabhängigkeit kann als Grund für die Abweichungen der Energieangaben ausgeschlossen werden. Die von der parametergewahren Vorhersagefunktion $l(\vec{p})$ modellierten Energiewerte weisen ebenfalls hohe mittlere Abweichungen auf, die sich nur um wenige Prozent von denen des statischen Modells unterscheiden. Eine Modellierung in Abhängigkeit der aktuellen Parameter bringt daher keine signifikanten Vorteile. Dies wird auch durch die Heuristik der automatischen Auswertung unterstützt, welche lediglich bei sleep, setTxPower und send mit $\frac{\overline{\sigma_X}}{\sigma_X} \approx \frac{1}{4}$ eine wenig eindeutige (und wie oben angemerkt nicht vorhandene) Abhängigkeit vermutet und in den restlichen Fällen eindeutig keine feststellt.

6.2.5 Funkmodul nRF24L01+

Auch die Messungen für den nRF24L01+ mussten wegen der Hardwarekomplexität eingeschränkt werden. Hierzu wurde die maximale Anzahl an Zustandsbesuchen auf vier gesetzt und nur die Transitionsfolgen

- begin · setDataRate · setPALevel · write · epilogue · powerDown,
- begin · setDataRate · setPALevel · startListening · stopListening · powerDown und
- begin · powerDown · powerUp · powerDown · powerUp · powerDown · powerUp · powerDown

erlaubt. Es wurden 16 Messungen mit einem Shunt von 68 Ω und 400 ms Wartezeit in den Zuständen durchgeführt, die alle auswertbar waren. Tabelle 6.6 zeigt die resultierenden Modellwerte.

Wie schon beim CC1200 wird die Leistungsaufnahme der Zustände TX und RX von dynamischen Konfigurationsparametern des Funkmoduls beeinflusst und ist daher nicht statisch modellierbar. Die automatische Auswertung stellt diese Parameterabhängigkeit mit $\frac{\overline{\sigma_P}}{\sigma_P} = 0{,}006$ für RX und $0{,}0003$ für TX eindeutig fest.

Die Zustände POWERDOWN und STANDBY1 sind von den Parametern unabhängig, was auch so erkannt wird. Im Vergleich mit den Datenblattangaben von 3,24 µW für POWERDOWN und 93 µW für STANDBY1 fällt auf, dass die hier gemessenen Leistungswerte deutlich niedriger sind [Sem08]. Eine probeweise durchgeführte Messung an einem zweiten

Zustand	$\tilde{P}$			n
POWERDOWN	0,04 µW	0,01 µW	30 %	48
RX*	52,2 mW	1,13 mW	2,2 %	192
STANDBY1	7,24 µW	1,16 µW	14 %	3136
TX*	18,4 mW	8,11 mW	34 %	576

Transition		$\tilde{E}$		$\widetilde{\Delta E}$		$\tilde{t}$	n
begin		1,65 µJ		1,65 µJ		19,8 ms	784
	57,3 nJ	3,4 %	1,6 µJ	7,5 %	1,63 ms	10 %	
epilogue		15,4 nJ		-744 nJ		40 µs	576
	2,58 nJ	15 %	367 nJ	36 %	3,87 µs	8,6 %	
powerDown		4,55 nJ		3,85 nJ		90 µs	832
	1,42 nJ	35 %	1,42 nJ	41 %	1,66 µs	2 %	
powerUp		1,64 µJ		1,64 µJ		10 ms	48
	65,6 nJ	3,9 %	65,6 nJ	3,9 %	2,5 µs	0,03 %	
setDataRate		7,75 nJ		6,78 nJ		140 µs	768
	1,97 nJ	26 %	1,96 nJ	30 %	3,4 µs	2,5 %	
setPALevel		4,7 nJ		3,73 nJ		90 µs	768
	1,57 nJ	35 %	1,57 nJ	45 %	1,42 µs	1,5 %	
startListening		4,31 µJ		4,31 µJ		260 µs	192
	131 nJ	3 %	131 nJ	3 %	4,48 µs	1,7 %	
stopListening		194 nJ		-13,5 µJ		260 µs	192
	109 nJ	62 %	516 nJ	3,7 %	15,3 µs	5,6 %	
write		218 nJ		215 nJ		510 µs	576
	56,3 nJ	27 %	56,3 nJ	27 %	3,78 µs	0,73 %	

Tabelle 6.6: Statische Modellwerte des nrf24L01+. Die Leistung der mit $\star$ markierten Zustände hängt von Parametern ab.

nRF24L01+-Modul ergab allerdings höhere Modellwerte von 0,08 µW für POWERDOWN und 34 µW für STANDBY1, so dass hier konservative Herstellerangaben im Datenblatt eine naheliegendere Erklärung als eine Querversorgung über die Datenleitungen sind.

Bei den Transitionen ist die in *begin* und *powerUp* enthaltene Wartezeit zur Initialisierung des Funkmoduls erkennbar. Zudem fällt auf, dass die Dauer von *write* im Gegensatz zur Sendetransition des CC1200 kaum schwankt, da die Funktion im nRF24L01+-Treiber wegen der standardmäßig konstanten Paketlänge unabhängig von der Länge der angegebenen Daten immer 32 Byte an das Funkmodul schickt. Die Dauer von *stopListening* wird mit $\frac{\overline{\sigma_t}}{\sigma_t} = 0{,}216$ als möglicherweise parameterabhängig erkannt, was den dort auftretenden Modellfehler von 5,6 % erklärt.

Für die hohen Abweichungen der Energiedaten sind wiederum Rauschen in den Messdaten und zufällige Schwankungen im Hardwareverhalten verantwortlich. Eine Parameterabhängigkeit wird von der Heuristik in jedem Fall ausgeschlossen und durch einen Vergleich mit $l(\vec{p})$ bestätigt: Für jede absolute und relative Energieangabe weicht der mittlere Fehler von $l(\vec{p})$ nur um wenige Prozentpunkte vom statischen Modellwert ab, meist sogar um weniger als einen. Es ist daher anzunehmen, dass die Transitionsenergie auch für dieses Funkmodul nur mit bis zu 35 % Abweichung modellierbar ist.

6.3 Transitionskosten

Bisher wurde zwar die Plausibilität einzelner Energiewerte geprüft, aber kein Vergleich zwischen relativen und absoluten Angaben vorgenommen. Dies wird in diesem Abschnitt nachgeholt. Dabei werden nur die absoluten Abweichungen vom Modellwert berücksichtigt, da ein Vergleich der prozentualen Daten wegen unterschiedlicher Bezugswerte keine Aussagekraft hat.

Bei Betrachtung der Tabellen 6.2 bis 6.6 zeigt sich, dass die relative Transitionsenergie fast immer einen niedrigeren Fehler als die absolute aufweist. Die einzigen Ausnahmen sind die init- und txDone-Transitionen des CC1200 sowie begin, epilogue und stopListening des nRF24L01+. Bei Temperatursensor und Display sind die Unterschiede gering, während der Fehler der relativen Transitionsenergie beim MicroMoody und den restlichen CC1200-Transitionen um den Faktor fünf bis 20 niedriger als der der absoluten ist. Der nRF24L01+ weist hingegen bei den restlichen Transitionen keinen signifikanten Unterschied zwischen relativen und absoluten Energiedaten auf.

Daraus folgt, dass abhängig von der Peripherie und dem Transitionstyp teils absolute und teils relative Transitionsenergie die bessere Wahl ist. Tatsächlich ist dieses Verhalten nicht zufällig, sondern hängt von den Eigenschaften der Transitionen und der beteiligten Zustände ab.

Kein signifikanter Unterschied zwischen relativer und absoluter Transitionsenergie besteht dann, wenn die möglichen Startzustände einer Transition eine ähnliche mittlere Leis-

tungsaufnahme aufweisen. In diesem Fall ist es unerheblich, ob sie von der Transitionsenergie abgezogen wird oder nicht, da sie ohnehin quasi konstant ist. Dieser Effekt wird beim LM75B und dem Display deutlich, in dem jeder Funktionsaufruf nur einen einzigen möglichen Ausgangszustand hat, dessen Leistungsaufnahme zudem nicht von irgendwelchen Parametern abhängt. Auf die oben nicht namentlich erwähnten Transitionen des nRF24L01+ trifft dies ebenfalls zu.

Wenn die verschiedenen Ausgangszustände – oder der eindeutige, aber parameterabhängige Ausgangszustand – einer Funktion hingegen starke Leistungsschwankungen aufweisen, sind relative Energieangaben klar überlegen. Dies zeigt sich insbesondere beim MicroMoody, wo die Transitionen zum Ändern der Farbe sowohl vom Zustand OFF (mit sehr niedriger Leistung) als auch von B und G (mittlere Leistung) und R (hohe Leistung) ausgehen können. Da der Effekt der Transition erst nach deren Ende eintritt, ist die Transitionsenergie für ihre gesamte Dauer von der Leistung des vorher aktiven Zustands abhängig. Gerade diese Abhängigkeit wird von relativen Energie-Angaben herausgefiltert, was den dort deutlich niedrigeren Fehler erklärt.

Gegenüber absoluten Energiewerten unterlegen sind relative Angaben schließlich in Transitionen, die entweder erst nach einer impliziten Zustandsänderung durch eine Unterbrechung aufgerufen werden, oder deren Effekte bereits zu ihrer Laufzeit aktiv werden. Ersteres trifft auf txDone (CC1200) und epilogue (nRF24L01+) zu: Diese Transitionen werden erst ausgeführt, nachdem der Sendezustand verlassen wurde, weshalb ihr Energiebedarf in keinem Bezug zur Leistung des Sendezustands steht und daher nur schlecht relativ zu dieser modelliert werden kann. Zweiteres betrifft begin (CC1200) sowie init und stopListening (nRF24L01+). Die Funktionen begin und init führen langwierige Initialisierungsroutinen mit vielen Teilschritten aus, deren Effekte schon während der Ausführung der Funktion eintreten. stopListening hingegen beendet zuerst den Empfangsmodus des Funkmoduls, um anschließend den Status auszulesen – hier wird die Änderung sogar direkt zu Beginn der Transition aktiv.

Insgesamt zeigen diese Daten, dass die Nutzung relativer Energiewerte für Transitionen fast immer sinnvoll ist. Einerseits verringern sie den Energieberechnungsaufwand in Online-Modellen, da pro Transition nur ein- statt zweimal die Systemzeit ausgelesen werden muss, und andererseits verringern sie den Modellfehler um bis zu Faktor 20. Sie führen also in vielen Fällen zu einer deutlich höheren Modellqualität.

Von den hier betrachteten Transitionen haben lediglich fünf von 35 unter der Nutzung relativer Energiewerte gelitten, was auf bereits vor oder während der Transition eintretende Transitionseffekte zurückzuführen ist. Es ist anzunehmen, dass die Modellgüte in diesen Fällen durch die Verwendung von zum Folgezustand relativen Energiewerten verbessert werden kann. In jedem Fall sollten in der Praxis genutzte Modelle mehrere Arten von Transitionsenergie-Angaben unterstützen, um für jeden Transitionstyp die optimale verwenden zu können.

6.4 Parametrisierte Modellwerte

Nachdem Abschnitt 6.2 gezeigt hat, dass die automatisierte Testprogrammgenerierung und Messdatenauswertung plausible statische Modelle generiert und somit als verlässlich betrachtet werden kann, wird nun die automatische Erkennung und Modellierung von parametrisierten Modellwerten ausgewertet.

Hierzu betrachten wir zunächst die Erkennung des Vorhandenseins einer Abhängigkeit und des Abhängigkeitstyps. Nach einem Exkurs zur Modellbewertung mittels Kreuzvalidierung folgt eine Gütebewertung der so erstellten parametrisierten Modelle und ein Vergleich mit von Hand vorgegebenen Modellen.

Da Temperatursensor, Display und statische MicroMoody-Firmware über keinerlei Parameter verfügen, werden hier nur noch die parametrisierte MicroMoody-Firmware sowie die Funkmodule CC1200 und nRF24L01+ betrachtet.

6.4.1 Abhängigkeitserkennung

Die Evaluation der Abhängigkeitserkennung besteht aus zwei Schritten. Sie beginnt mit einem Test, ob verschiedene Arten von vorgegebenen Abhängigkeiten zuverlässig erkannt werden. Hierzu wurde die MicroMoody-Firmware mit verschiedenen Sollfunktionen zur Umsetzung angegebener Helligkeitswerte in tatsächliche Leistungsaufnahme angepasst und ihr Verhalten jeweils automatisch vermessen und ausgewertet. Darauf folgt eine Betrachtung der beiden Funkmodule als Praxisbeispiel für Peripheriegeräte mit parameterabhängigen Energiemodellen. Hier werden die Auswertungsergebnisse auf Plausibilität geprüft und mit manuell vorgegebenen Abhängigkeitstypen verglichen.

MicroMoody

Die parametrisierte MicroMoody-Firmware (vgl. Abschnitt 5.6.3) verfügt über die Zustände ON und OFF und die globalen Parameter *red* und *green* mit Wertebereich 0 bis 255. Im OFF-Zustand ist die Leistungsaufnahme konstant, während sie in ON von den per Parameter eingestellten Helligkeitswerten abhängt – standardmäßig ist dieser Zusammenhang wegen der verwendeten Pulsweitenmodulation linear.

Die automatisierte Auswertung sollte daher feststellen, dass die Leistung des Zustands ON sowie die Energie der dort beginnenden Transition *off* linear von red und green abhängt. Im Zustand OFF und bei der Transition *setBrightness* sollten keinerlei Abhängigkeiten vorliegen, ebenso bei Dauer und relativer Energie der Transition off.

Mit Ausnahme der relativen Energie von off werden diese Annahmen von den in Tabelle 6.7 gezeigten Auswertungsdaten bestätigt. Für die Leistung von ON zeigt sich wie erwartet eine eindeutige Abhängigkeit von den Parametern red und green, genau so wie auch bei der Energie von off. Diese wird korrekt als linear identifiziert und die Modellab-

Eigenschaft	$\frac{\overline{\sigma_X}}{\sigma_X}$	$\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}}$ (green)	$\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}}$ (red)	**SMAPE**	
ON ($\tilde{P}$)	0,001	0,002	0,001	53 %	0,6 %
OFF ($\tilde{P}$)	1,000	–	–	0,1 %	
setBrightness ($\tilde{E}$)	1,000	–	–	2,4 %	
setBrightness ($\widetilde{\Delta E}$)	1,000	–	–	3,1 %	
setBrightness ($\tilde{t}$)	1,000	–	–	0,3 %	
off ($\tilde{E}$)	0,017	0,036	0,019	33 %	0,6 %
off ($\widetilde{\Delta E}$)	0,022	0,044	0,036	9,3 %	1,7 %
off ($\tilde{t}$)	0,025	0,911	0,903	1,6 %	

Tabelle 6.7: Erkennung von Parameter-Abhängigkeiten bei Vermessung der parametrisierten MicroMoody-Firmware. Die SMAPE-Spalte zeigt links die Abweichung des statischen und rechts die des parametrisierten Modells.

weichung bei Berücksichtigung der Parameter ist entsprechend niedrig. Ebenso zeigt sich eindeutig keine Abhängigkeit der Leistung von OFF sowie irgendeiner Eigenschaft von setBrightness.

Relative Energie und Dauer der Transition off weichen allerdings von den Annahmen ab. Bei der Dauer vermutet die Heuristik eine Parameter-Abhängigkeit, stellt dann aber bei Betrachtung der einzelnen Parameter korrekt fest, dass doch keine vorliegt. Grund hierfür ist, dass für dieses Modell nur zwei vollständige Messläufe durchgeführt wurden, so dass jede Parameterkombination lediglich zweimal vermessen wurde. Die Streuung von Werten mit identischen Parametern ist alleine deshalb schon geringer als die Streuung von Werten ohne Parameterberücksichtigung, weshalb auch die Fehleinschätzung der Heuristik nicht überraschend ist. Im Praxiseinsatz kann dies durch die Erhebung weiterer Messdaten leicht umgangen werden.

Bei der relativen Transitionsenergie stellt die Heuristik hingegen auch für die Parameter green und red eine eindeutige Abhängigkeit fest, deren Berücksichtigung zudem den mittleren Modellfehler reduziert. Der Unterschied ist allerdings deutlich geringer als bei der absoluten Transitionsenergie. Grund hierfür ist nicht ein Fehler in der Auswertung, sondern eine Eigenheit des parametrisierten MicroMoody-Treibers. Dieser überträgt in jeder Transition drei Bytes an das MicroMoody – entweder ein Byte zum Einschalten der LED gefolgt von den Helligkeitswerten von grün und rot oder ein Byte zum Ausschalten der LED gefolgt von zwei beliebigen Bytes. Da für jede Transition der gleiche Sendepuffer verwendet wird, überträgt off nach dem Ausschaltbyte immer die zuvor von setBrightness verwendeten Helligkeitswerte. Diese werden vom MicroMoody zwar nicht ausgewertet, verursachen wegen der Pull-Up-Widerstände an der I^2C-Schnittstelle aber merkbare Unterschiede in der relativen Transitionsenergie.

Bei den im Testprogramm verwendeten Helligkeitswerten (0, 1, 2, 16, 32, 64, 128, 255) weist 0 eine besonders hohe relative Energie auf, da acht Null-Bits übertragen werden, während 255 keine Null-Bits und somit eine sehr geringe relative Energie hat. Die Werte dazwischen haben allesamt genau sieben Null-Bits und liegen somit auch energetisch zwischen 0 und 255. Da die LEDs deutlich mehr Leistung als die Pull-Up-Widerstände umsetzen, fällt dies erst bei Betrachtung der relativen und nicht schon bei der absoluten Energie auf.

Die erkannte Abhängigkeit ist also korrekt und auch die automatisch bestimmten Modellfunktionen $\#_{0(8)}(red)$ und $\#_{0(8)}(green)$ (d.h. Anzahl der Null-Bits bei Binärkodierung als Acht-Bit-Zahl) unterstützen diese Erklärung. Der parametrisierte Modellfehler ist mit 1,7 % entsprechend niedrig.

Um auch die Erkennung anderer Abhängigkeiten zu überprüfen, wurden in einem letzten Analyseschritt weitere Sollfunktionen getestet. Hierzu wurde die MicroMoody-Firmware angepasst, so dass die per I^2C angegebenen Werte für red und green nicht mehr direkt, sondern abhängig von einer vorgegebenen Funktion in Helligkeit umgesetzt werden. Bei Tests mit verschiedenen Kombinationen von linearen, logarithmischen, quadratischen und Wurzelfunktionen wurde in jedem Fall sowohl die Abhängigkeit selbst als auch die Art der Funktion für die Leistung von ON und die absolute Energie von off zuverlässig erkannt.

Die korrekte Funktion der automatischen Erkennung und Modellierung von parameterabhängigem Modellverhalten ist damit empirisch bestätigt, so dass als nächstes die für die Funkmodule erstellten Modelle untersucht werden können.

Funkmodule

Beide Funkmodule verfügen über drei Parameter: Sendeleistung (*txpower*), Bitrate (*symbolrate* bzw. *datarate*) und Paketlänge (*txbytes*). Sendeleistung und Bitrate werden beim Initialisieren der Module auf Standardwerte gesetzt und sind damit immer bekannt, während die Paketlänge nur definiert ist, nachdem die Sendefunktion aufgerufen wurde.

Zu erwarten ist, dass die Leistungsaufnahme der Sendezustände von der eingestellten Sendeleistung und möglicherweise Bitrate abhängt. Das Timeout der darauf folgenden Unterbrechung (*txDone* bzw. *epilogue*) sollte von Paketlänge und Bitrate bestimmt werden und von der Sendeleistung unabhängig sein, während die Leistung der Empfangszustände und die Dauer von *stopListening* ausschließlich von der Bitrate abhängt.

Auch hier bestätigen die Auswertungsergebnisse in Tabelle 6.8 die Annahmen. Lediglich die Abhängigkeit des CC1200-Sendezustands von der Paketlänge und der fehlende Zusammenhang zwischen epilogue-Timeout und Paketlänge beim nRF24L01+ entsprechen nicht den Erwartungen, begründen sich aber in Eigenheiten der Funkmodule.

Auf Basis der ermittelten Abhängigkeiten und Abhängigkeitsarten bestimmt die automatisierte Modellverfeinerung für den CC1200 die folgenden Funktionen zur Berechnung

Eigenschaft	$\frac{\overline{\sigma_X}}{\sigma_X}$	$\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}}$ (datarate)	$\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}}$ (txbytes)	$\frac{\overline{\sigma_X}}{\overline{\sigma_{X,i}}}$ (txpower)
CC1200: TX ($\tilde{P}$)	0,013	0,055	0,086	0,015
CC1200: RX ($\tilde{P}$)	0,034	0,035	–	0,959
CC1200: txDone ($\tilde{T}$)	0,001	0,001	0,001	0,771
nRF24: TX ($\tilde{P}$)	0,001	0,001	0,909	0,001
nRF24: RX ($\tilde{P}$)	0,006	0,006	–	0,994
nRF24: stopListening ($\tilde{t}$)	0,216	0,216	–	0,970
nRF24: epilogue ($\tilde{T}$)	0,001	0,001	0,873	0,849

Tabelle 6.8: Erkennung von Parameter-Abhängigkeiten für die Funkmodule CC1200 und nRF24L01+.

von TX- und RX-Leistung (in µW) sowie txDone-Timeout (in µs). Im Sinne einer kompakten Darstellung werden die Regressionsparameter dabei gerundet ohne Nachkommastellen angegeben.

$$
\begin{aligned}
P_{TX}(sr, txb, txp) &= 81805 - 574 \cdot \ln(txb) - 1242 \cdot \sqrt{sr} + 18 \cdot txp^2 \\
&\quad + 233 \cdot \ln(txb) \cdot \sqrt{sr} - 0 \cdot \ln(txb) \cdot txp^2 - 1 \cdot \sqrt{sr} \cdot txp^2 \\
&\quad + 0 \cdot \ln(txb) \cdot \sqrt{sr} \cdot txp^2 \\
P_{RX}(symbolrate) &= 84415 + 206 \cdot \ln(symbolrate + 1) \\
T_{txDone}(sr,\ txb) &= 366 - 0 \cdot txb + \frac{80102}{sr} + 8000 \cdot \frac{txb}{sr}
\end{aligned}
$$

Die Regressionsparameter von P_{TX} zeigen, dass die Paketlänge tatsächlich einen klar modellierbaren Einfluss auf diesen Zustand nimmt. Dieser begründet sich im Funkprotokoll des CC1200: Vor jeder Datenübertragung wird zunächst eine konstante Präambel gesendet, bei deren Übertragung der CC1200 geringfügig weniger Leistung benötigt als beim Senden der darauf folgenden Nutzdaten. Je kürzer das gesendete Paket ist, desto höher ist der zeitliche Anteil der Präambel am gesamten TX-Zustand, was die mittlere Leistungsaufnahme entsprechend verringert.

Dieser Zusammenhang ist auch in der Übertragungsdauer T_{txDone} sichtbar. Wie erwartet wird der Funktionswert primär durch das Verhältnis aus Paketlänge zu Bitrate bestimmt. Ein alleiniger Einfluss durch die Paketlänge ist nicht vorhanden, wohl aber ein durch die Präambel bedingter alleiniger Einfluss der Bitrate – je höher die Bitrate, desto weniger Zeit wird zum Senden der Präambel aufgewendet. Abbildung 6.4 zeigt die Funktionsgraphen von T_{txDone} abhängig von Bitrate und Paketlänge. Die einzelnen Graphen im Bitratendiagramm entsprechen verschiedenen Paketlängen und Sendeleistungen, die einzelnen Graphen im Paketlängendiagramm verschiedenen Bitraten und Sendeleistungen. Da die Sendeleistung keinen messbaren Einfluss auf die Übertragungsdauer hat, überde-

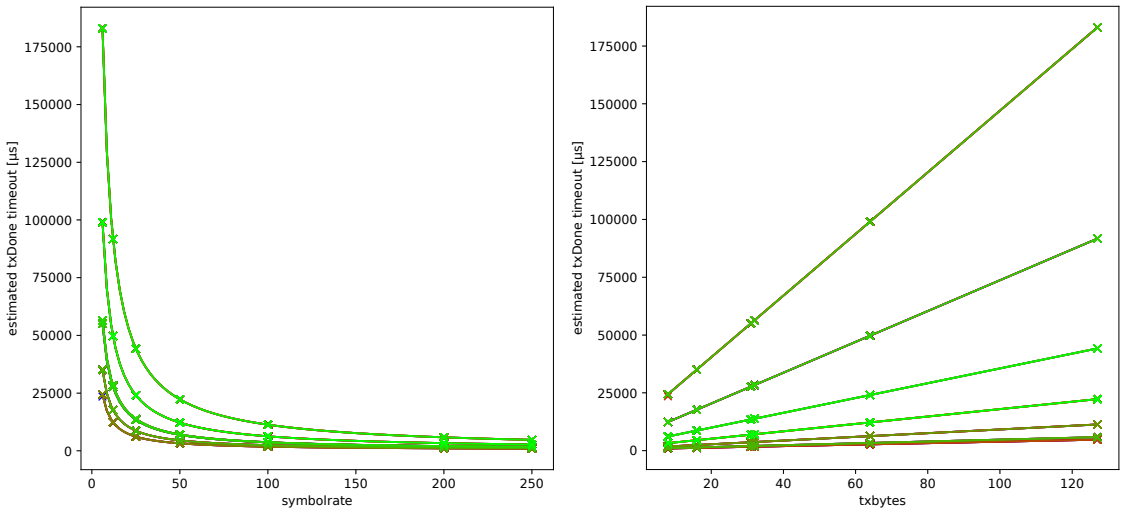

Abbildung 6.4: Messwerte und Funktionsgraphen für die Übertragungsdauer (T_{txDone}) des CC1200 in Abhängigkeit von der eingestellten Bitrate (links) und Paketlänge (rechts).

cken sich die Funktionsgraphen der einzelnen Sendeleistungseinstellungen, weshalb sie im Diagramm nicht unterscheidbar sind.

Schließlich zeigen die Funktionen, dass für Modelle mit niedrigem Fehler sowohl alleinstehende Einflüsse einzelner Parameter, als auch kombinierte Einflüsse mehrerer Parameter beachtet werden müssen. So kann T_{txDone} weder durch die Funktionsvorschrift $a_1 + a_2 \cdot txb + \frac{a_3}{sr}$, noch durch $a_1 + a_2 \cdot \frac{txb}{sr}$ akkurat beschrieben werden – erst die Kombination aus dem alleinigen Einfluss von *txbytes* und dessen Verhältnis zu *symbolrate* erfasst das Verhalten der Hardware vollständig. Die in Abschnitt 4.5.3 konzipierte Funktionsgenerierung per Potenzmenge sieht solche kombinierten Einflüsse vor und ist daher auch für Modelleigenschaften, die von mehr als nur einem Parameter abhängen, geeignet.

Für den nRF24L01+ bestimmt die Modellverfeinerung die folgenden Funktionen, die hier ebenfalls mit gerundeten Regressionsparametern angegeben werden.

$$
\begin{aligned}
P_{TX}(sr,\ txp) &= 13918 + 245 \cdot txp + \frac{8368807}{sr} + 271527 \cdot \frac{txp}{sr} \\
P_{RX}(symbolrate) &= 48531 + 117 \cdot \sqrt{symbolrate} \\
t_{stopListeting}(datarate) &= 248 + \frac{12231}{datarate} \\
T_{epilogue}(symbolrate) &= 26062 + \frac{5314558}{symbolrate}
\end{aligned}
$$

Eine grafische Darstellung von P_{RX} und $T_{epilogue}$ findet sich zudem in Abbildung 6.5. Im Vergleich zum CC1200 weist das Verhalten dieses Moduls zwei signifikante Unterschiede auf: Alle Pakete haben unabhängig von den zu übertragenden Nutzdaten eine feste Länge von 32 Byte und bei Funkübertragungen wird automatisch auf eine Empfangsbestätigung

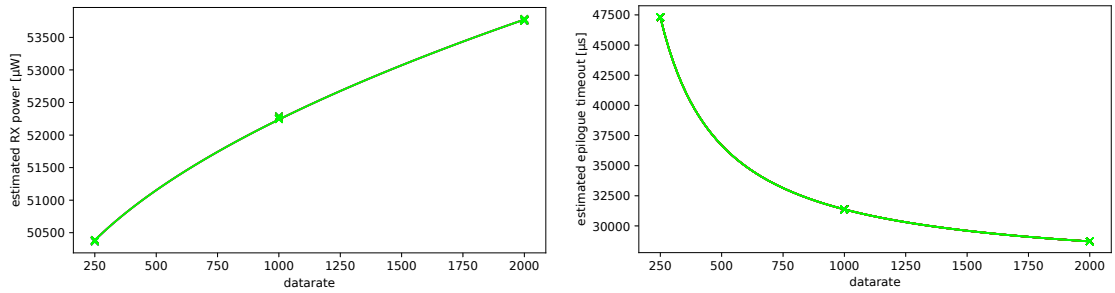

Abbildung 6.5: Automatisch bestimmte Funktionen für P_{RX} (links) und $T_{epilogue}$ (rechts) des nRF24L01+.

gewartet sowie der Übertragungsversuch bis zu 15 mal wiederholt. Bei zweiterem werden die 15 Sendeversuche hier immer ausgereizt, da während der Messungen kein zweites Funkmodul als Empfänger aktiv ist.

Diese Unterschiede zeigen sich in den Modellfunktionen durch Unabhängigkeit vom Parameter *txbytes* und dem mit 26 ms sehr hohen statischen Anteil von $T_{epilogue}$. Nicht dadurch erklärbar ist allerdings, warum für P_{TX} hier ein linearer Zusammenhang bestimmt wird, während dieser beim CC1200 als quadratisch modelliert wurde und auch bei Funkmodulen allgemein häufig quadratisch oder exponentiell ist.

Ein Vergleich mit einer manuell vorgegebenen quadratischen Berechnungsfunktion (siehe Abbildung 6.6) zeigt, dass auch beim nRF24L01+ tatsächlich eine quadratische Abhängigkeit vorliegt, wegen der eigenwilligen Parameterkodierung des zugehörigen Treibers aber nicht erkannt wird. Dieser nutzt nicht wie der CC1200 positive Zahlen, um die Differenz zur minimal möglichen Sendeleistung auszudrücken, sondern erwartet die absolute Sendeleistung als Argument für die Funktion *setPALevel*. Die zur Modellerstellung genutzten Parameterwerte sind daher -18, -12, -6 und $0\,\mathrm{dBm}$.

Da eine quadratische Funktion bei negativen Zahlen mit steigenden Werten nicht wächst, sondern abfällt, wird die quadratische Abhängigkeit von der automatisierten Modellverfeinerung nicht erkannt. Erst eine Verschiebung in den positiven Bereich durch Modellierung der Sendeleistung als „$x - 18\,\mathrm{dBm}$“ bringt Abhilfe. Eine Unterstützung solcher Verschiebungen ist daher eine sinnvolle Erweiterungsmöglichkeit des hier entworfenen Modellverfeinerungskonzepts.

Schließlich ist anzumerken, dass bei der Auswertung der Messdaten für den CC1200 die Parameterabhängigkeits-Heuristik auch bei der Energie einiger Transitionen eine Abhängigkeit vermutet, obwohl die Modellgüte von $l(\vec{p})$ eindeutig zeigt, dass dort kein Zusammenhang zwischen Messwerten und Parametern besteht. Grund hierfür ist eine Beeinflussung der Heuristik durch einzelne Ausreißer in den Messwerten. Eine ebenfalls sinnvolle Konzepterweiterung ist daher, die Heuristik zur Parameterabhängigkeitserkennung beispielsweise durch zusätzliche Berücksichtigung von $l(\vec{p})$ robuster gegenüber Ausreißern zu machen.

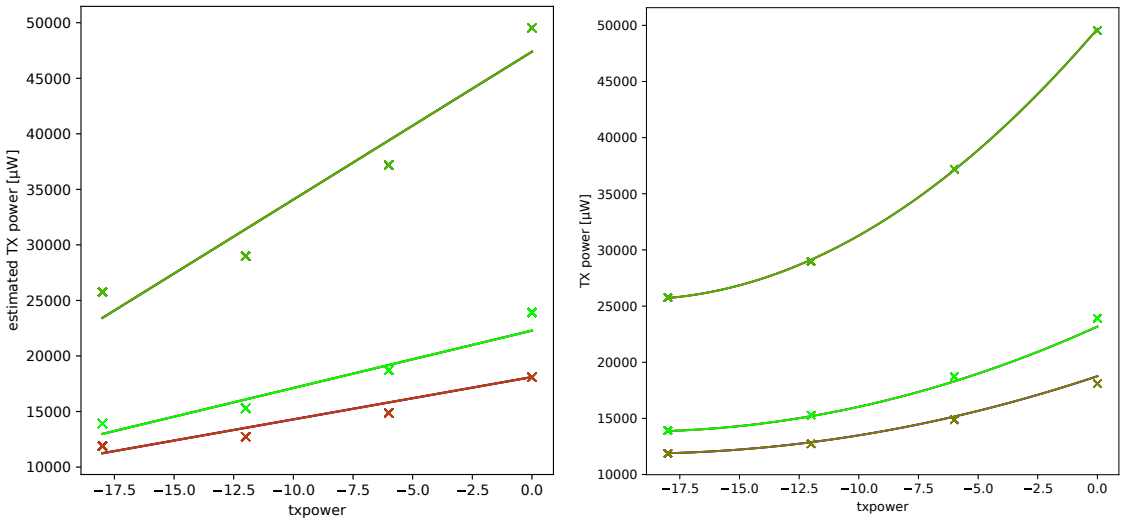

Abbildung 6.6: Automatisch bestimmte (links) und manuell vorgegebene (rechts) Modellfunktion für die Sendeleistung (P_{TX}) des nRF24L01+.

Zuletzt stellt sich nur noch die Frage nach der Güte der hier vorgestellten Modelle. Im Gegensatz zu den statischen Angaben kann diese durch Berechnung von RMSD, MAE und SMAPE der jeweiligen Funktionen auf den vorliegenden Messdaten nicht verlässlich abgeschätzt werden. Eine derartige Güteberechnung ist nämlich anfällig für Überanpassung: Da die Funktionen auf denselben Daten bewertet werden, mit denen sie auch trainiert wurden, zeigt ein niedriger Modellfehler lediglich, dass diese spezifischen Daten gut abgebildet werden. Ob tatsächlich der funktionale Zusammenhang zwischen den einzelnen Datenpunkten erfasst wurde, lässt sich daraus nicht schließen.

Die partielle Funktion $l(\vec{p})$ veranschaulicht dieses Problem besonders gut. Da sie jedem Parametertupel das arithmetische Mittel der Messwerte zuordnet, ist sie per Definition das Modell mit der geringsten quadratischen Abweichung. Gleichzeitig ist l als partielle Funktion nicht in der Lage, Leistungs-, Energie- oder Zeitwerte für im Training nicht enthaltene Parametertupel zu bestimmen. Sie ist zwar auf den vorliegenden Messdaten optimal, in der Praxis aber nutzlos.

Eine verlässliche Bewertung der Modellgüte ist daher nur möglich, wenn das Modell zuerst auf einem Datensatz trainiert wird und anschließend auf einem zweiten, unabhängigen Datensatz die Gütemaße bestimmt werden [FKL07, S. 159ff]. Ergibt auch dieser Validierungsdatensatz niedrige Abweichungen, ist das Modell i.A. praktisch nutzbar. Abhängig davon, ob die Modellgüte für bereits bekannte Parameterwerte oder auch für unbekannte Kombinationen bestimmt werden soll, ist dazu eine Vielzahl von Trainings- und Validierungsmessungen notwendig.

Da dies in der Praxis sehr aufwändig ist, wird stattdessen oft eine *Kreuzvalidierung* durchgeführt. Die bereits vorhandenen Messdaten werden dazu in eine Trainings- und eine

Validierungsmenge aufgeteilt, so dass die Trainingsmenge die Resultate der Trainings- und die Validierungsmenge die Resultate der Validierungsmessung imitiert. Dieses Verfahren wird auch in dieser Arbeit genutzt und im folgenden Abschnitt vorgestellt.

6.4.2 Exkurs: Kreuzvalidierung

Das grundlegende Prinzip der Kreuzvalidierung ist einfach: Unter Voraussetzung eines hinreichend großen Datensatzes wird dieser in eine Trainings- und eine Validierungsmenge partitioniert, die gegebene Modellfunktion mittels Regressionsanalyse auf die Trainingsdaten hin optimiert und schließlich anhand der Validierungsdaten ihre Gütemaße bestimmt.

Es stellt sich lediglich die Frage, wie genau die Daten aufgeteilt werden sollen. Hierzu gibt es einige verbreitete Methoden, die sich durch ihren *Bias* (d.h. der Optimismus oder Pessimismus der Gütemaße im Vergleich mit der „richtigen“ Güte) und ihre *Varianz* (d.h. die Streuung der Gütemaße verschiedener Kreuzvalidierungsläufe auf den gleichen Daten) unterschieden.

- k-fache Kreuzvalidierung partitioniert die Daten in k möglichst gleich große Mengen und führt k Trainings- und Validierungsläufe durch. Im ersten Lauf wird das Modell anhand der ersten Menge evaluiert, nachdem es zuvor auf den Mengen zwei bis k trainiert wurde. Im zweiten Lauf dient die zweite Menge zur Evaluierung und die restlichen zum Training und so weiter. Die Gütemaße der Kreuzvalidierung sind die Mittelwerte der Gütemaße der einzelnen Läufe [Rei10].
- Leave-One-Out-Kreuzvalidierung (kurz LOO) verhält sich auf n Datenpunkten wie k-fache Kreuzvalidierung mit $k = n$. Sie führt n Läufe durch, in denen jeweils ein Datenpunkt zur Validierung und alle anderen zum Training genutzt werden [XL01].
- Monte-Carlo-Kreuzvalidierung wählt eine zufällige Teilmenge der Daten als Trainings- und den Rest als Validierungsmenge. Üblicherweise enthält die Trainingsmenge die Hälfte bis zwei Drittel aller Daten. Das Verfahren wird mehrfach (bis zu einige hundert Mal) durchgeführt, um sicherzustellen, dass möglichst viele Datenpunkte mindestens einmal zur Validierung dienen [XL01; Had+13].

Bei k-facher Kreuzvalidierung wird zudem zwischen zufälliger, systematischer und stratifizierter Partitionierung unterschieden. Zufällige Partitionierung verhält sich ähnlich wie eine Monte-Carlo-Kreuzvalidierung mit großer Trainingsmenge, stellt aber im Gegensatz zu dieser sicher, dass jeder Datenpunkt genau einmal zur Validierung verwendet wird. Systematische Partitionierung sortiert die Daten und nutzt im ersten Lauf die Elemente mit Index 1, $k+1$, $2k+1$, $\ldots$ zur Validierung, im zweiten solche mit Index 2, $k+2$, $2k+2$, $\ldots$ und so weiter. Dieses Verfahren ist nur nutzbar, wenn die Daten z.B. anhand eines Zeitstempels sortierbar sind. Im Gegensatz zur Monte-Carlo-Kreuzvalidierung liefert es deterministische Gütewerte, ist aber anfällig gegenüber periodischen Daten [Rei10].

Stratifizierte k-fache Kreuzvalidierung bildet die Partitionen so, dass die Verteilung der Daten in jeder Partition ungefähr gleich ist. Falls die Daten z.B. mehrere Häufungspunkte aufweisen, kann dies durch gleichmäßige Aufteilung der Daten in den einzelnen Häufungspunkten auf die verschiedenen Partitionen umgesetzt werden [Koh+95; Rei10].

In der Praxis wird die Leave-One-Out-Methode kaum verwendet, da sie einerseits mit einem hohen Berechnungsaufwand verbunden ist, und andererseits wie die Modellbewertung ohne Kreuzvalidierung zu optimistische Gütemaße liefert und daher einen hohen Bias hat [XL01]. Stattdessen wird k-fache Kreuzvalidierung mit $k = 10$ bevorzugt, da diese einen guten Kompromiss aus meist niedrigem Bias und niedriger Varianz bietet. Auch Monte-Carlo-Kreuzvalidierung ist wegen ihres niedrigen Bias verbreitet, hier muss aber die durch die Zufallskomponente erhöhte Varianz beachtet werden [Had+13].

Zur Anwendbarkeit der stratifizierten Kreuzvalidierung für die hier erhobenen Daten muss berücksichtigt werden, dass diese Methode für Messdaten mit unbekannten Häufungspunkten und vielen verschiedenen Parameterwerten vorgesehen ist. In dieser Arbeit werden aber nur die im Modell vorgegebenen Parameterwerte verwendet, so dass zwar viele Messdaten aber nur wenige verschiedene Parameterkombinationen vorhanden sind. Insbesondere kann typischerweise jeder Parameterkombination ein Häufungspunkt der modellierten Messdaten zugeordnet werden, was deutlich in den Abbildungen 6.4 und 6.6 im vorherigen Abschnitt sichtbar ist.

Eine stratifizierte Partitionierung würde daher im Extremfall Messwerte für jede Parameterkombination sowohl im Training als auch in der Validierung verwenden. Da die Modellgüte für unbekannte Parameter so nie überprüft wird, sind die bestimmten Gütemaße zu optimistisch. Aus diesem Grund kommt diese Methode hier nicht zum Einsatz.

Stattdessen verwenden wir zwei verschiedene Kreuzvalidierungsmethoden, um sowohl die Modellgüte allgemein als auch die Vorhersagefähigkeit für unbekannte Parameter im Besonderen zu bestimmen. Hierzu kommt einerseits eine 200-fache Monte-Carlo-Kreuzvalidierung mit Aufteilung der Messdaten in zwei Drittel Trainings- und ein Drittel Validierungsmenge zum Einsatz und andererseits eine zehnfache systematische Kreuzvalidierung, bei der die Daten anhand ihrer Parameterwerte partitioniert werden.

Die Monte-Carlo-Kreuzvalidierung dient der allgemeinen Bewertung der Modellgüte. Durch die vielen Wiederholungen wird die Varianz gering gehalten und gleichzeitig eine Vielzahl von Fällen überprüft. Durch die zufällige Aufteilung werden sowohl Fälle getestet, in denen eine Parameterkombination aus der Validierungsmenge bereits mit anderen Messwerten im Training vorkam, als auch solche, in denen das Modell darauf nicht trainiert wurde. Diese Methode ist daher eine geeignete Abschätzung für die Modellgüte in der Praxis, da auch dort das Modell nach Möglichkeit mit den Parametern trainiert wird, mit denen es später am häufigsten genutzt wird, aber auch neue Parameterwerte oder -kombinationen auftreten können.

Methode	P_{TX}		P'_{TX}		P_{RX}		T_{txDone}	
MC	814	0,71 %	2083	1,71 %	68	0,07 %	11	0,10 %
$k = 10$	963	0,93 %	2251	2,03 %	70	0,07 %	8	0,10 %

Tabelle 6.9: RMSD und SMAPE des parametrisierten CC1200-Modells für Monte-Carlo-Kreuzvalidierung (oben) und parametergewahre zehnfache Kreuzvalidierung (unten).

Die systematische Kreuzvalidierung dient hingegen der Bestimmung der Modellgüte für nicht im Training betrachtete Parameter. Hierzu werden die Messdaten nicht anhand ihres Zeitstempels, sondern anhand der Parameter sortiert und partitioniert, so dass jede Kombination von Parameterwerten nur in genau einer Partition vorkommt. Das Modell wird also ausschließlich auf Daten validiert, deren Parameterwerte im Training nicht vorkamen. Dieses Vorgehen wird im Folgenden auch als parametergewahre Kreuzvalidierung bezeichnet.

6.4.3 Modellgüte

Wie schon die Evaluation der Abhängigkeitserkennung besteht dieser Abschnitt aus zwei Teilen. Er beginnt mit einer Gütebewertung der in Abschnitt 6.4.1 vorgestellten Modellfunktionen für die Funkmodule CC1200 und nRF24L01+. Anschließend folgt eine Untersuchung der automatischen Funktionsbestimmung bei verschiedenen Sollfunktionen und eingeschränktem Parameterraum. Diese dient zu Gütebewertung des *Meta-Modells*, nämlich der automatischen Erkennung von Abhängigkeiten und Modellfunktionen.

Funkmodule

Tabelle 6.9 zeigt die Kreuzvalidierungsergebnisse für den CC1200. Neben den automatisch bestimmten Funktionen P_{TX}, P_{RX} und T_{txDone} enthält diese zusätzlich die manuell vorgegebene Funktion P'_{TX}. Diese dient dem beispielhaften Test, ob die in P_{TX} modellierten gemeinsamen Einflüsse einzelner Parameter tatsächlich nützlich sind oder hier eine getrennte Betrachtung der Parameter genügt.

$$P'_{TX}(sr, txb, txp) = a_1 + a_2 \cdot \ln(txb) + a_3 \cdot \sqrt{sr} + a_4 \cdot txp^2$$

Aus den Ergebnissen folgt, dass die automatisch bestimmten Funktionen das Hardwareverhalten gut abbilden. Empfangsleistung und Übertragungsdauer werden auch für im Training nicht betrachtete Parameterwerte mit einer Abweichung von weniger als 0,1 % berechnet, während die Berechnung der Sendeleistung auf unter 1 % genau ist. Im Vergleich mit Tabelle 6.5 reduziert die Berücksichtigung von Parametern den Modellfehler auf weniger als ein Zehntel der Abweichung des statischen Modells.

Methode	P_{TX}		P'_{TX}		P_{RX}		$t_{stopListening}$		$T_{epilogue}$	
MC	1526	5,66 %	337	1,20 %	15	0,02 %	4	1,36 %	3	0,01 %
$k = 10$	1775	6,53 %	349	1,30 %	16	0,03 %	4	1,46 %	3	0,01 %

Tabelle 6.10: RMSD und SMAPE des parametrisierten nrf24L01+-Modells für Monte-Carlo-Kreuzvalidierung (oben) und parametergewahre zehnfache Kreuzvalidierung (unten)

Auch im Vergleich mit anderen Modellen sind diese Werte gut. Das für Funkmodule beliebte Three States Model with State Transitions erreicht beispielsweise mit 1,13 % einen ähnlichen mittleren Fehler und unterstützt keinerlei Parameter, während das auf Smartphones und Laptops genutzte Sesame-Framework mit Parameterunterstützung bei Modellierung des Gesamtsystems sogar eine mittlere Abweichung von 5 % aufweist [Hur+11; DZ11].

Die im Vergleich zu P_{TX} doppelt so hohe mittlere Abweichung von P'_{TX} bedeutet schließlich, dass die zusätzlichen Regressionsparameter und gemeinsamen Parametereinflüsse in P_{TX} für das Modell nützlich sind und nicht zu einer Überanpassung führen.

Die Daten für das Funkmodul nRF24L01+ in Tabelle 6.10 zeigen für Empfangsleistung und Übertragungsdauer ein ähnliches Bild: Der Modellfehler liegt jeweils unter 0,1 %. Die Dauer von *stopListening* ist mit 1,5 % Abweichung im Rahmen der Messgenauigkeit ebenfalls akzeptabel. Die Leistungsaufnahme im Sendezustand wird mit einer Abweichung von 6 % nur mäßig gut beschrieben, was aus der bereits bemerkten Fehlerkennung des quadratischen Sendeleistungseinflusses als lineare Funktion folgt. Die Tabelle enthält daher zusätzlich die manuell erstellte Funktion P'_{TX}, die diesen Fehler durch Verschiebung des *txpower*-Parameters korrigiert und wie folgt definiert ist.

$$P'_{TX}(sr,\ txp) = a_1 + a_2 \cdot (a_5 + txp)^2 + \frac{a_3}{sr} + a_4 \cdot \frac{(a_5 + txp)^2}{sr}$$

Diese Funktion beschreibt das Hardwareverhalten mit einem mittleren Fehler von 1,3 % hinreichend gut.

Insgesamt zeigt sich, dass die automatische Funktionsgenerierung in der Praxis überwiegend einwandfrei funktioniert. Alle parameterabhängigen Eigenschaften des CC1200 werden korrekt erkannt, beim nRF24L01+ ist nur die Art der Abhängigkeit zwischen TX-Leistung und *txpower* fehlerhaft. Dieser Fehler folgt aus den zur ersten Konzeptevaluation bewusst einfach gehaltenen Funktionsvorschriften zur automatischen Modellierung und kann voraussichtlich durch die Aufnahme komplexerer Funktionen behoben werden.

Sollzusammenhang	x	$\ln(x+1)$	$\sqrt{x}$	x^2
SMAPE ($k = 10$)	0,69 %	0,64 %	0,52 %	0,40 %
Erkennungsrate	100 %	100 %	100 %	100 %
SMAPE (MC)	1,05 %	1,27 %	0,74 %	1,40 %
Erkennungsrate	98 %	96 %	90 %	98 %

Tabelle 6.11: Güte des Meta-Modells und Erkennungsrate der Sollfunktion bei zehnfacher (oben) und Monte-Carlo-Kreuzvalidierung (unten) der MicroMoody-Leistung auf Parameterbasis.

Meta-Modell

Zur weiteren Analyse der automatischen Funktionsbestimmung kommt eine abgewandelte Kreuzvalidierungsmethode zum Einsatz. Hierzu wird für jede Trainingsmenge eine vollständige automatische Auswertung mit Analyse der Parameterabhängigkeit und der Art des Zusammenhangs durchgeführt, welche wiederum für jede Trainingsmenge eine eigene Funktion mit zugehörigen Regressionsparametern generiert, deren Güte beim darauf folgenden Validierungsschritt bestimmt wird. Falls die Modellverfeinerung in vielen Fällen unpassende Funktionen liefert, also z.B. einen quadratischen anstatt eines linearen Zusammenhangs modelliert, schlägt sich dies in entsprechend hohen mittleren Fehlern wieder.

Für diese Robustheitsbewertung wurde die MicroMoody-Firmware mit verschiedenen Funktionen zur Berechnung der LED-Helligkeit auf Basis der per I^2C übertragenen Helligkeitswerte ausgestattet und ihr Energieverhalten vermessen. Die Gütebestimmung beschränkt sich auf die modellierte Leistung des ON-Zustands, da diese ausschließlich von den Helligkeitsparametern abhängt. Neben der prozentualen Abweichung der modellierten Leistung ist hier zusätzlich interessant, wie häufig die in der Firmware vorgegebene Funktion von der automatischen Auswertung erkannt wird.

Hierzu wurden eine lineare, einer logarithmische, eine quadratische und eine Wurzelfunktion in der Firmware vorgegeben und die damit erhobenen Messwerte mittels Kreuzvalidierung ausgewertet. Als Kreuzvalidierungsverfahren kamen eine parametergewahre zehnfache systematische und eine parametergewahre Monte-Carlo-Kreuzvalidierung zum Einsatz – die Validierungsmenge enthält also nur Parameterkombinationen, auf denen das (Meta-)Modell nicht trainiert wurde. Die Ergebnisse dieses Experiments zeigt Tabelle 6.11.

Bei der zehnfachen Kreuzvalidierung wird in jedem Fall die korrekte Funktion erkannt, zudem beträgt der mittlere Modellfehler weniger als 0,7 %. Auch die parameterbasierte Monte Carlo-Kreuzvalidierung zeigt einen hohen Anteil korrekt erkannter Funktionen und einen entsprechend niedrigen Fehler bis maximal 1,4 %. Da die zehnfache Kreuzvalidierung neun Zehntel der Parameterkombinationen zum Training nutzt, die Monte-Carlo-Version hingegen lediglich zwei Drittel, ist auch der Unterschied zwischen den Verfahren nicht überraschend.

Auch die wenigen fehlerhaft erkannten Funktionen ähneln immer noch der Sollfunktion. So wird anstelle von $\sqrt{x}$ am häufigsten eine logarithmische Funktion bestimmt, welche im Wertebereich 0 bis 255 mit entsprechender Skalierung deutlich näher an $\sqrt{x}$ verläuft als eine lineare oder gar quadratische. Zusammen mit den niedrigen Abweichungen und hohen Erkennungsraten der korrekten Funktion zeigt dies, dass die automatische Abhängigkeitserkennung auch unter widrigen Bedingungen robust ist.

6.5 Erkennung fehlender Parameter

Die automatisierte Abhängigkeitserkennung und -modellierung kann nur gute Modelle erzeugen, wenn alle relevanten Parameter bekannt und im Modell angegeben sind. Fehlen einzelne Parameter, die das Hardwareverhalten beeinflussen, schlägt sich dies in höheren Abweichungen der erzeugten Modellfunktionen nieder. Um dies zu vermeiden, sieht das Modellverfeinerungskonzept eine Erkennung von Treiberfunktionen vor, die solche fehlenden Parameter verändern.

Zur Evaluation dieses Aspekts wurden Treiber und Modell des nRF24L01+ um die Funktion *enableDynamicPayloads* erweitert, welche im IDLE-Zustand aufgerufen werden kann und das Funkmodul von Paketen mit konstanter Länge auf eine dynamische Paketgröße umschaltet. Die modellierte Zustandsmenge und die globalen Parameter wurden allerdings nicht verändert, so dass die Funktion aus Sicht des Modells keinen Einfluss auf das Hardwareverhalten hat.

In Läufen mit dieser Funktion sollten Übertragungsdauer und -leistung zusätzlich von der Paketlänge abhängen. Zweiteres begründet sich darin, dass die Wartezeit für erneute Sendeversuche konstant ist, die Sendedauer für dynamische Paketlängen aber variabel – bei kurzen Paketen wird daher im Mittel weniger Zeit mit der eigentlichen Übertragung verbracht und daher eine geringere mittlere Leistung benötigt.

Eine mit diesem Model durchgeführte Modellverfeinerung stellt mit $\frac{\overline{\sigma_{X,\rightarrow}}}{\overline{\sigma_X}} = 0{,}009$ für die Leistung von TX und $\frac{\overline{\sigma_{X,\rightarrow}}}{\overline{\sigma_X}} = 0{,}007$ für das Timeout der *epilogue*-Funktion fest, dass diese vom Präfix des Worts (d.h. den zuvor abgelaufenen Funktionsaufrufen) abhängen. Der Einfluss von enableDynamicPayloads wird mit $\frac{\overline{\sigma_{X,\rightarrow}}}{\overline{\sigma_{X,tr}}} = 0{,}017$ für TX und 0,013 für epilogue ebenfalls eindeutig festgestellt, ein Einfluss anderer Funktionen wird nicht vermerkt. Auch bei den restlichen Modelleigenschaften wird wie erwartet keine Abhängigkeit vom Präfix gefunden. Dies gilt ebenso für das Modell des CC1200, was sich mit den getroffenen Annahmen über die Hardware deckt.

Somit ist auch diese Heuristik in der Lage, das Vorhandensein fehlender Parameter und die zugehörige Funktion zu erkennen. Mit dem daraus gewonnenen Wissen können Anwender das Modell um von diesen Funktionen abhängige Parameter erweitern und durch eine erneute Modellverfeinerung bestimmen, wie genau sie das Hardwareverhalten beeinflussen.

6.6 Zeitaufwand

Die vorhergehenden Abschnitte haben gezeigt, dass die verschiedenen automatisierten Modellverfeinerungsmethoden nur in Ausnahmefällen manuelle Eingriffe erfordern – in dieser Evaluation lediglich bei der Beschreibung der Sendeleistung des nRF24L01+ sowie nach der Erkennung der Auswirkungen der Funktion enableDynamicPayloads. Manche dieser Eingriffe erfordern lediglich eine neue Auswertung bestehender Messdaten, in anderen Fällen muss die gesamte Modellverfeinerung wiederholt werden. Da zu hohe Wartezeiten nach solchen Änderungen die effektive Nutzung des Konzepts einschränken, ist auch die Dauer von Datenerhebung und Auswertung relevant.

Erstere hängt von der Komplexität des modellierten Peripheriegeräts bzw. des zugehörigen Automaten $\mathcal{A}$ ab. Ein komplexer Automat mit vielen Parametern hat eine große Sprache $L_{k,\star}(\mathcal{A})$, so dass die Datenerhebung entsprechend lange dauert. Bei den hier verwendeten Modellen reicht die Dauer von 20 bis 30 Minuten für den LM75B über eine Stunde für den nRF24L01+ bis hin zu drei Stunden für den CC1200. Diese Wartezeit ist nicht vernachlässigbar, im Sinne zuverlässiger Messwerte aber auch nicht vermeidbar.

Die Dauer des Auswertungsschritts ist weitgehend vernachlässigbar. Die Aufbereitung der Messdaten benötigt auf einer Intel Core i5-2320-CPU aus dem Jahr 2011 etwa eine Minute Echtzeit je 20 Minuten Messzeit, die Auswertung selbst bewegt sich zwischen wenigen Sekunden für Modelle ohne Parameter und knapp unter einer Minute für den CC1200. Da die Aufbereitungsergebnisse zwischengespeichert und von nachträglichen Modelländerungen nicht beeinflusst werden, ist somit keine nennenswerte Wartezeit zwischen zwei Auswertungen auf denselben Messdaten vorhanden.

Insgesamt ist eine effektive Nutzung der Modellverfeinerung möglich. Zwar benötigt die Datenerhebung selbst aus Praxisgründen einen nicht unerheblichen Zeitraum; da sie keinerlei manuelle Intervention benötigt, ist diese Wartezeit aber vertretbar.

Kapitel 7

Fazit

In dieser Arbeit wurden Methoden zur automatisierten Verfeinerung von Energiemodellen konzipiert und anhand einer prototypischen Implementierung mit verschiedenen Peripheriegeräten evaluiert. Neben der Automatisierung bekannter Arbeitsschritte wie Testprogrammerstellung und Messdatenauswertung wurden dabei zwei Methoden zur Modellierung der Transitionsenergie verglichen und eine eigens entwickelte Heuristik zur Erkennung und Modellierung von Parameter-Abhängigkeiten in Zustands- und Transitionseigenschaften untersucht.

Sie zeigt, dass ein automatisiertes Mess- und Auswertungsverfahren trotz der Unterschiede zwischen verschiedenen Peripheriegeräten mit einem generischen Ansatz umsetzbar ist. Sowohl bei der Messdatenerhebung als auch bei der Auswertung sind kaum manuelle Eingriffe notwendig. Das als Grundlage genutzte Automatenmodell mit Zeit-, Energie- und Leistungsannotationen kann sowohl statisches als auch parameterabhängiges Verhalten beschreiben und erlaubt die Generierung von Testprogrammen mit minimaler manueller Intervention.

Zur Modellierung der Transitionsenergie wurden sowohl absolute als auch zum vorherigen Zustand relative Energieangaben untersucht. Wir konnten feststellen, dass relative Angaben für Transitionen geeignet sind, deren Wirkung erst nach dem Transitionsende eintritt, während absolute Angaben für bereits zu Beginn oder während der Transition eintretende Effekte einen niedrigeren Modellfehler aufweisen.

Die Erkennung und Modellierung von parametrisierten Modelleigenschaften beruht auf einer Heuristik, welche die Messdaten anhand verschiedener Kriterien partitioniert und die Streuung von Partitionen mit festen und Partitionen mit variablen Parametern vergleicht. Zur Beschreibung des Abhängigkeitstyps kommt eine Reihe von Funktionskandidaten zum Einsatz, aus der für jeden Parameter und jede davon abhängige Modelleigenschaft mittels nichtlinearer Regression und Vergleich der quadratischen Modellfehler eine Funktion ausgewählt wird. Aus diesen Teilergebnissen wird eine Funktion zur Beschreibung der Eigenschaft durch alle Parameter erzeugt.

Hier zeigt die Evaluation, dass sowohl die Heuristik zur Abhängigkeitserkennung als auch die Erstellung der Gesamtfunktion bei künstlich modellierter Hardware robust arbeiten und auch bei echten Peripheriegeräten plausible Modelle generieren. Bis auf eine Ausnahme weisen die so erstellten Modellfunktionen einen mittlere Fehler von unter 1,5 % auf, lediglich die Sendeleistung des nRF24L01+ wird mit einer Abweichung von 6 % beschrieben. Dieser Modellfehler folgt aus den bewusst einfach gehaltenen Funktionskandidaten und konnte durch eine manuelle Anpassung der Funktion behoben werden.

Alle bekannten oder vorgegebenen Parameter-Abhängigkeiten wurden von der Heuristik zuverlässig erkannt. In einzelnen Fällen wurden nicht parametrisierte Modelleigenschaften allerdings fälschlich als parameterabhängig klassifiziert, obwohl ein parametrisiertes Modell keinen nennenswerten Gütegewinn brachte.

Schließlich wurde auch eine automatische Erkennung von im Modell nicht berücksichtigten Parametern konzipiert und ihre Funktion anhand eines exemplarischen Tests mit dem nRF24L01+ erfolgreich verifiziert.

Für weitere Arbeiten an diesem Konzept bietet sich eine Verbesserung der Heuristik an, um weniger parameter-unabhängige Eigenschaften als parameterabhängig zu klassifizieren. Dies gilt ebenso für die Erweiterung der genutzten Modelle, um Transitionen zur Verwendung absoluter oder relativer Energie-Angaben zu klassifizieren.

Zudem hat die Auswertung gezeigt, dass Energie und Dauer von Transitionen neben globalen Parametern auch von lokalen Funktionsargumenten abhängen können. Eine dahingehende Erweiterung von Modell und Auswertung erscheint daher sinnvoll. Hier kann auch untersucht werden, ob die manuelle Vorgabe globaler Parameter durch eine Analyse von Funktionsargumenten und ihren Auswirkungen ersetzt werden kann.

Schließlich wurde das Thema Modellverfeinerung in dieser Arbeit primär in Bezug auf die automatische Erhebung statischer Modelldaten sowie die Erkennung und Modellierung von Parametern und Parameter-Abhängigkeiten umgesetzt. In Hinblick auf Online- und Offline-Modelle bleibt die Frage, wie sinnvoll die Unterstützung und gleichzeitige Verwaltung verschiedener Modelle mit unterschiedlichem Detailgrad ist. So könnte sowohl ein grobes und ggf. rein statisches Online-Modell gepflegt werden als auch ein detailliertes Offline-Modell, welches als Verfeinerung mit dem Online-Modell verknüpft ist.

In diesem Punkt ist auch die automatische Erkennung und Modellierung von Teilzuständen wie der im CC1200-Sendezustand enthaltenen Präambel interessant. Hierdurch kann detaillierte Kenntnis über die innere Funktion von Peripheriegeräten gewonnen werden, so dass unintuitives Verhalten möglicherweise auch ohne Blick in die rohen Messdaten erklärt werden kann. Zudem ist der dadurch gewonnene Detailgrad für Batteriemodelle interessant, da Batterieverhalten auch durch kurze Leistungsspitzen beeinflusst wird.

Anhang A

Implementierungsdetails

Dieser Anhang enthält zwei XML-Modelle als Beispiele für das Format der entworfenen Energiemodelle sowie einige für die Evaluation des Konzepts nicht benötigte, aber zur Anwendung der implementierten Software relevante Details. Hierzu zählen eine Befehlsreferenz des in dieser Arbeit entwickelten *dfatool*-Programms und ein Anwendungsbeispiel zur Vermessung des nRF24L01+.

A.1 Energiemodelle

Hier werden zwei Arten von Energiemodellen vorgestellt, die an verschiedenen Punkten im Modellverfeinerungszyklus verwendet werden.

A.1.1 Display

Als erstes wird das Hardwaremodell des Displays mit Stand vor der ersten Modellverfeinerung wiedergegeben. Dieses enthält noch keine Energie-Angaben, sondern gibt lediglich die modellierten Zustände und Transitionen vor. Zudem enthält es die zur Testprogrammgenerierung genutzten Funktionsargumente und gibt an, dass im ENABLED-Zustand regelmäßige toggleVCOM-Aufrufe zur Polaritätsumkehr notwendig sind.

```
<?xml version="1.0"?>
<data>
  <driver name="sharpLS013B4DN">
    <states>
      <state name="UNINITIALIZED"/>
      <state name="DISABLED"/>
      <state name="ENABLED"/>
    </states>
    <transitions>
      <transition name="ioInit">
        <src>UNINITIALIZED</src>
```

```
      <dst>DISABLED</dst>
      <level>user</level>
    </transition>
    <transition name="enable">
      <src>DISABLED</src>
      <dst>ENABLED</dst>
      <level>user</level>
    </transition>
    <transition name="disable">
      <src>ENABLED</src>
      <dst>DISABLED</dst>
      <level>user</level>
    </transition>
    <transition name="sendLine">
      <param name="lineno">
        <value>0</value>
        <value>50</value>
        <value>90</value>
        <value>95</value>
      </param>
      <param name="line">
        <value>(unsigned char *)"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
          </value>
        <value>(unsigned char *)"\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff"</value>
      </param>
      <src>ENABLED</src>
      <dst>ENABLED</dst>
      <level>user</level>
    </transition>
    <transition name="toggleVCOM">
      <src>ENABLED</src>
      <dst>ENABLED</dst>
      <level>user</level>
    </transition>
    <transition name="clear">
      <src>ENABLED</src>
      <dst>ENABLED</dst>
      <level>user</level>
    </transition>
  </transitions>
</driver>
<after-transition>
  <if state="ENABLED">
    <transition name="toggleVCOM"/>
```

```
    </if>
  </after-transition>
</data>
```

A.1.2 Funkmodul nRF24L01+

Das hier abgedruckte Energiemodell des nRF24L01+-Funkmoduls entspricht dem Stand nach dem zweiten Verfeinerungszyklus. Im ersten Durchlauf wurden die Energiewerte und *estimate*-Funktionen bestimmt. Anschließend wurde nach Betrachtung des Funktionsgraphen für die TX-Leistung manuell eine *user*-Funktion hinzugefügt und diese im zweiten Verfeinerungslauf mit Regressionsparametern ausgestattet.

```
<?xml version="1.0"?>
<data>
  <driver name="RF24">
    <parameters>
      <param name="datarate" functionname="setDataRate_num" functionparam="speed"/>
      <param name="txpower" functionname="setPALevel_num" functionparam="level"/>
      <param name="txbytes" functionname="write_nb" functionparam="len"/>
    </parameters>
    <states>
      <state name="UNINITIALIZED"/>
      <state name="POWERDOWN" power="0"/>
      <state name="STANDBY1" power="7"/>
      <state name="RX" power="52254">
        <powerfunction>
          <estimate param0="48530.7323548634" param1="117.252744034887"><![CDATA[
            0 + param(0) + param(1) * np.sqrt(global(datarate))
          ]]></estimate>
        </powerfunction>
      </state>
      <state name="TX" power="18414">
        <powerfunction>
          <user param0="9901.63558040847" param1="11.7063412513263"
            param2="3919966.42471389" param3="12969.5986611517"
            param4="19.4689730995954"><![CDATA[
              param(0) + param(1) * (param(4)+global(txpower))**2 + param(2)/global(datarate)
              + param(3) * (param(4)+global(txpower))**2 / global(datarate)
          ]]></user>
          <estimate param0="13918.0987720778" param1="245.185542707497"
            param2="8368807.37972909" param3="271527.645821987"><![CDATA[
              0 + param(0) + param(1) * global(txpower) + param(2) * 1 / global(datarate)
              + param(3) * global(txpower) * 1 / global(datarate)
          ]]></estimate>
```

```
        </powerfunction>
      </state>
    </states>
    <transitions>
      <transition name="begin" energy="1652249" rel_energy="1649571" duration="19830">
        <src>UNINITIALIZED</src><src>STANDBY1</src><src>RX</src>
        <dst>STANDBY1</dst>
        <level>user</level>
        <affects>
          <param name="datarate" value="1000"/>
          <param name="txpower" value="0"/>
        </affects>
      </transition>
      <transition name="powerDown" energy="4547" rel_energy="3854" duration="90">
        <src>STANDBY1</src>
        <dst>POWERDOWN</dst>
        <level>user</level>
      </transition>
      <transition name="powerUp" energy="1641765" rel_energy="1641381" duration="10030">
        <src>POWERDOWN</src>
        <dst>STANDBY1</dst>
        <level>user</level>
      </transition>
      <transition name="startListening" energy="4309602" rel_energy="4307769"
          duration="260">
        <src>STANDBY1</src>
        <dst>RX</dst>
        <level>user</level>
      </transition>
      <transition name="stopListening" energy="193775" rel_energy="-13533693"
          duration="260">
        <src>RX</src>
        <dst>STANDBY1</dst>
        <level>user</level>
        <durationfunction>
          <estimate param0="247.888808139524" param1="12231.1046534396">
            <![CDATA[0 + param(0) + param(1) * 1 / global(datarate)]]>
          </estimate>
        </durationfunction>
      </transition>
      <transition name="setPALevel_num" energy="4700" rel_energy="3728" duration="90">
        <src>STANDBY1</src>
        <dst>STANDBY1</dst>
        <level>user</level>
```

```
        <param name="level">
78        <value>-18</value>
          <value>-12</value>
80        <value>-6</value>
          <value>0</value>
82      </param>
      </transition>
84    <transition name="setDataRate_num" energy="7749" rel_energy="6777" duration="140">
        <src>STANDBY1</src>
86      <dst>STANDBY1</dst>
        <level>user</level>
88      <param name="speed">
          <value>1000</value>
90        <value>2000</value>
          <value>250</value>
92      </param>
      </transition>
94    <transition name="write_nb" energy="218339" rel_energy="214618" duration="510">
        <src>STANDBY1</src>
96      <dst>TX</dst>
        <level>user</level>
98      <param name="buf">
          <value>"Hello␣World.␣LTIsWzNB8SlNlCGUIza"</value>
100     </param>
        <param name="len">
102       <value>8</value>
          <value>16</value>
104       <value>32</value>
        </param>
106   </transition>
      <transition name="epilogue" energy="15449" rel_energy="-744114" duration="40">
108     <src>TX</src>
        <dst>STANDBY1</dst>
110     <level>epilogue</level>
        <timeoutfunction>
112       <estimate param0="26062.1299660899" param1="5314557.8972845">
            <![CDATA[0 + param(0) + param(1) * 1 / global(datarate)]]>
114       </estimate>
        </timeoutfunction>
116   </transition>
    </transitions>
118 </driver>
</data>
```

Verzeichnis	Inhalt
`data`	Erhobene Messdaten und aufbereitete Datensätze
`dfatool/bin`	Hauptprogramm dfatool und Teilprogramme
`dfatool/lib`	Bibliotheken für dfatool, analyze und merge
`kratos`	Quelltext und Konfiguration von Kratos
`kratos/dfa-driver`	Energiemodelle im XML-Format
`mimosa-autocal`	Quelltext und Firmware für mimosactl.

Tabelle A.1: Verzeichnisstruktur von dfatool und Hilfsprogrammen.

An dieser Stelle sei angemerkt, dass die Treiberfunktionen `setPALevel`, `setDataRate` und `write` in der hier erstellten Implementierung `setPALevel_num`, `setDataRate_num` und `write_nb` heißen. In den bisherigen Abschnitten wurden sie im Interesse einer kompakten Darstellung verkürzt notiert, das soeben wiedergegebene Modell nutzt aber die vollständigen Namen.

A.2 dfatool-Referenz

Die Implementierung der Modellverfeinerung besteht aus Anpassungen in Kratos, dem Programm *dfatool* mit zugehörigen Bibliotheken und Teilprogrammen und dem zur Kalibrierung genutzten Programm *mimosactl*. Als Teilprogramme dienen die Pythonskripte *analyze.py* und *merge.py*, welche von dfatool aufgerufen werden und Datenaufbereitung bzw. Auswertung und Modellverfeinerung durchführen. Tabelle A.1 zeigt die Verzeichnisstruktur dieser Komponenten.

Die Bibliotheken für dfatool selbst bestehen aus

- `AspectC::Repo` zum Auslesen der Treibersignaturen aus der Datei repo.acp,
- `Kratos::DFADriver` zur Generierung von Testprogrammen, Instrumentierung von Treibern und Durchführung von Messungen,
- `Kratos::DFADriver::DFA` für die Verwaltung des DFAs und seiner Sprache,
- `Kratos::DFADriver::Model` als Schnittstelle zum XML-Modell,
- `MIMOSA` zur Kontrolle von mimosactl und MimosaCMD und
- `MIMOSA::Log` zum Laden und Auswerten von Messdaten mit Hilfe der Pythonskripte analyze und merge.

Zusätzlich wird das externe Modul `FLAT` mitgeliefert, da es im Gegensatz zu den restlichen Abhängigkeiten nicht leicht von Hand installiert werden kann.

Die Pythonskripte werden durch die Module `dfatool` und `plotter` unterstützt. Ersteres enthält Hilfsfunktionen zur Auswertung und die Klasse MIMOSA zum Laden und Vorverarbeiten von MIMOSA-Logdaten. Letzteres kapselt die Funktionen zur Darstellung von Boxplots und Funktionsgraphen.

Aus Anwendersicht finden alle Interaktionen mit dem Skript `dfatool` im `kratos`-Verzeichnis statt. Dieses setzt Umgebungsvariablen für die Perl- und Pythonmodule und gibt anschließend alle Argumente an `../dfatool/bin/dfatool` weiter, um die eigentliche Arbeit zu erledigen. Es geht davon aus, dass die von Kratos verwendeten Teile der MSP430-Entwicklungsumgebung und kconfig sowie die Programme mimosactl und MimosaCMD bereits im Ausführungspfad vorhanden sind.

Zur Verwaltung von energiegewahren Gerätetreibern und Testprogrammen stellt dfatool die folgenden Funktionen zur Verfügung, die nach dem Schema `dfatool` [*Optionen*] *Aktion treiber.xml* aufgerufen werden.

- `dfatool enable` instrumentiert den im Energiemodell vermerkten Gerätetreiber, um die durch Zustände und Transitionen verbrauchte Energie vorzuhalten. Dabei werden nur die statischen Modellangaben genutzt. Optional kann ein GPIO-Pin mit `--trigger-port` und `--trigger-pin` als Trigger für MIMOSA konfiguriert werden. Die Option `--ignore-nested-calls` schränkt die Energiebuchführung ein, so dass innerhalb der Treiberklasse aufgerufene Transitionsfunktionen ignoriert werden. Falls noch keine Energiewerte bekannt sind, werden alle Angaben auf Null gesetzt.
- `dfatool disable` hebt diese Instrumentierung auf. Der Gerätetreiber wird also von einem energiegewahren zurück in einen herkömmlichen Treiber überführt.
- `dfatool flash` kompiliert Kratos und überträgt das Betriebssystem auf den MSP430.
- `dfatool maketest` erzeugt das Testprogramm als Kratos-Anwendung. Mit den Optionen `--exclude-states` und `--trace-filter` können die verwendeten Wörter eingeschränkt werden, zudem kann mittels `--state-duration` und `--trace-revisit` die Aufenthaltszeit in Zuständen sowie die maximale Zahl an Besuchen des gleichen Zustands konfiguriert werden.
- `dfatool rmtest` entfernt das Testprogramm.
- `dfatool reset` setzt den MSP430 zurück.
- `dfatool log` führt einen vollständigen Messlauf durch. Es setzt den MSP430 zurück, verbindet sich mit dessen serieller Schnittstelle, startet eine Messung mit MIMOSA, erzeugt mittels mimosactl die Kalibrierungsdaten und zeichnet schließlich die Ausgaben auf. Sobald der MSP430 das Ende des Messlaufs signalisiert, wird die MIMOSA-Messung beendet. Die Konfiguration von MIMOSA wird über die Optionen `--shunt` und `--voltage` angegeben und mit den erhobenen Messdaten archiviert.
- `dfatool loop` kapselt alle zur Testprogrammerstellung und Datenerhebung notwendigen Schritte. Es führt nacheinander die Aktionen enable, maketest und flash aus. Anschließend wird die log-Aktion bis zur manuellen Beendigung wiederholt. Sämtliche Argumente dieser Aktionen werden auch hier akzeptiert.

Zur Auswertung und sonstigen Handhabung von erhobenen Testdaten dient das Schema `dfatool` [*Optionen*] *Aktion treiber.xml daten.tar* mit den folgenden Funktionen.

- `dfatool analyze` wertet die angegebenen Messdaten aus und verfeinert das XML-Modell. Zudem werden die Modelldaten, Gütemaße und weitere Hinweise in Textform ausgegeben. Mit der Option `--no-update` kann eine Aktualisierung des Modells verhindert und mit `--plot=...` eine grafische Darstellung der Messdaten angefordert werden. Hier sind unter anderem die Argumente `states`, `transitions` und `timing` möglich, um Leistungs-, Energie- und Zeitdaten nach Zustand bzw. Transition aufgeschlüsselt als Boxplots darzustellen. Das Argument `fit` zeigt die geschätzten und manuell vorgegebenen Funktionen sowie die zugehörigen Messwerte als Graphen an.
- `dfatool validate` bestimmt die Güte des gegebenen Modells auf den gegebenen Messdaten. Hiermit kann eine Validierung durchgeführt werden, indem ein Modell erst auf einem Satz von Messdaten verfeinert und dann auf einem anderen Datensatz bewertet wird. Zudem werden hier die während der Ausführung vom Online-Modell bestimmten Daten ausgewertet.
- `dfatool crossvalidate` dient zur Kreuzvalidierung. Für jede Modelleigenschaft werden systematische zehnfache und Monte-Carlo-Kreuzvalidierung sowohl mit als auch ohne Parameterberücksichtigung bei der Partitionierung durchgeführt und die bestimmten Gütemaße ausgegeben. Dieser Vorgang ist noch nicht optimiert und kann einige Stunden in Anspruch nehmen.
- `dfatool ls` und `dfatool list` geben eine kurze Übersicht über die angegebenen Daten, den genutzten Messwiderstand und die Kalibrierungsergebnisse.
- `dfatool show` zeigt die Messdaten aufgeschlüsselt nach den einzelnen Wörtern des Testplans. Somit können Unstimmigkeiten in den Testläufen untersucht werden, wie es beispielsweise beim MicroMoody der Fall war.

A.3 Anwendungsbeispiel

Zuletzt zeigen wir anhand der Verfeinerung des nRF24L01+-Energiemodells die Anwendung von dfatool. Als Ausgangspunkt dienen der in Abschnitt 5.6.5 vorgestellte Treiber und das XML-Modell aus Abschnitt A.1.2. Letzteres verfügt zu Beginn der Verfeinerung noch über keine Leistungs-, Energie- oder Zeitattribute. Auch entsprechende Funktionen sind zu diesem Zeitpunkt noch nicht vorhanden, lediglich die globalen Parameter und Funktionsargumente sind bereits definiert.

Zur Datenerhebung wird zunächst mittels kconfig der nRF24L01+-Treiber in Kratos aktiviert und so konfiguriert, dass die später von dfatool erstellten Implementierungs- und Header-Ausschnitte eingebunden werden. Anschließend werden MIMOSA, Kalibrierplatine und MSP430 Launchpad mit dem PC verbunden und das Funkmodul an das MSP430 Launchpad angeschlossen. Dabei wird der Spannungsversorgungspin des nRF24L01+ ausgelassen. Die Aufteilung der restlichen Signale auf die MSP430-GPIO-Pins hängt von der Konfiguration des Treibers ab.

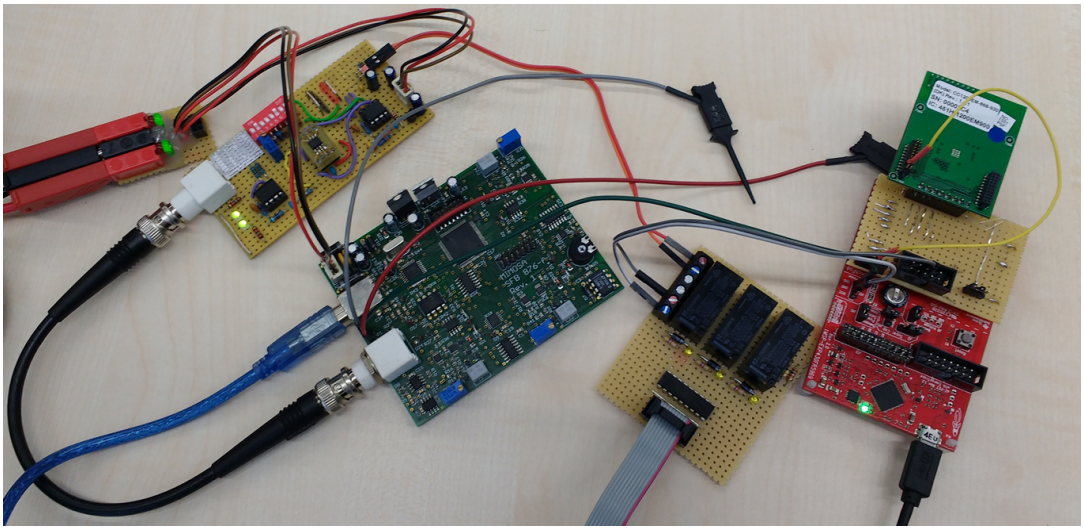

Abbildung A.1: Aufbau zur Modellverfeinerung des CC1200-Funkmoduls. Von links nach rechts: Versorgungsspannung, Shunt- und Integratorplatine von MIMOSA, Relaisplatine zur Kalibrierung und MSP430 Launchpad mit CC1200. Die aus dem Bild führenden Kabel enden in einem Laornetzteil, einem USB-Port, dem USB-Adapter der Kalibrierplatine und einem weiteren USB-Port.

Als nächstes wird der Eingang der Kalibrierplatine mit dem DUT-Ausgang von MIMOSA verbunden. Der VCC-Ausgang der Platine wird als Versorgungsspannung an den nRF24L01+ angeschlossen, der GND-Ausgang hingegen mit einem GND-Pin des Launchpads verbunden. Somit haben beide Geräte ein gemeinsames Massepotential.

Zuletzt muss noch das Buzzer-Signal von MIMOSA beschaltet werden. Hierzu bietet sich der Anschluss `P4.6` des MSP430 an, da dieser auf dem Launchpad leicht zugänglich und zudem bereits mit einer roten LED verbunden ist. Somit kann während einer Messung auch visuell inspiziert werden, ob die Transitionen bzw. Triggersignale wie erwartet ablaufen. Der Buzzer-GND-Pin von MIMOSA kann ebenfalls mit einem GND-Pin des Launchpads verbunden werden, dies ist aber optional und hat keinen merkbaren Einfluss auf die Messung.

Abbildung A.1 zeigt den gesamten Aufbau für das CC1200-Modul – die Beschaltung hier ist identisch.

Nun kann ein Testprogramm generiert und damit eine Messung durchgeführt werden. Für den nRF24L01+ kommt dazu die folgende Kommandozeile zum Einsatz.

```
./dfatool --ignore-nested-calls --trace-revisit 4 --shunt 68 --voltage 3.6 \
--trigger-port 4 --trigger-pin 6 --state-duration 400 \
--trace-filter begin,setDataRate_num,setPALevel_num,write_nb,epilogue,powerDown \
--trace-filter begin,setDataRate_num,setPALevel_num,startListening,stopListening,powerDown \
--trace-filter begin,powerDown,powerUp,powerDown,powerUp,powerDown,powerUp,powerDown \
loop dfa-driver/nrf24l01p_int.xml
```

Während der Messung wird die Nummer des aktuellen Messlaufs sowie der Fortschritt innerhalb dieses Laufs angezeigt. Sobald genug Daten erhoben wurden, kann die Messung durch Drücken von Strg+C beendet werden. Die Messdaten werden dabei nach dem Schema *YYYYmmdd_HHMMSS*_RF24.tar im Datenverzeichnis abgespeichert, also z.B. als `data/20170220_164723_RF24.tar`.

Mit diesen Daten wird die Modellverfeinerung durchgeführt. Für obiges Beispiel genügt dazu die folgende Kommandozeile.

```
./dfatool analyze dfa-driver/nrf24l01p_int.xml ../data/20170220_164723_RF24.tar
```

Optional kann z.B. mit `--plot=fit,states` eine grafische Übersicht über Zustände und parameterabhängige Funktionen angezeigt werden. Im Falle dieses Funkmoduls zeigte diese, dass die Leistung des TX-Zustands nicht korrekt modelliert wurde. Nach einer manuellen Anpassung der Funktionsvorschrift kann die neue Funktion durch Wiederholung des `dfatool analyze`-Befehls mit aus den Daten bestimmten Regressionsparametern ausgestattet und ihr Modellverhalten grafisch überprüft werden.

Sobald alle Modelleigenschaften zufriedenstellend modelliert werden, ist die Modellverfeinerung abgeschlossen. Das Ergebnis für dieses Beispiel findet sich in Abschnitt A.1.2.

Abkürzungsverzeichnis

ADC

Analog-Digital-Wandler (*Analog Digital Converter*). 8

DCO

Digital angesteuerter Oszillator (*Digitally Controlled Oscillator*). 15

DFA

Deterministischer endlicher Automat (*Deterministic Finite Automaton*). 44

DUT

Getestetes Gerät (*Device Under Test*). 8

FRAM

Nichtflüchtiger Arbeitsspeicher (*Ferroelectric Random Access Memory*). 15

GPIO

Generische Ein-/Ausgabeschnittstelle (*General Purpose Input/Output*). 16

I²C

Serielle Peripherieschnittstelle (*Inter-Integrated Circuit*). 17

Kratos

Maßschneiderbares ressourcengewahres Betriebssystem (*Kratos is a Resource-Aware Tailored Operating System*). 18

MAE

Mittlere absolute Abweichung (*Mean Absolute Error*), auch als MAD für *Mean Absolute Deviation* bekannt. 14

MIMOSA

Messgerät zur **i**ntegrativen **M**essung **o**hne **S**pannungs**a**bfall. 27

PTA

Endlicher Automat mit Kosten und Zeitangaben (*Priced Timed Automaton*). 10

RMSD

Wurzel der mittleren quadratischen Abweichung (*Root Mean Square Deviation*). 57

SCL

Taktleitung der I^2C-Schnittstelle (*Serial Clock*). 17

SDA

Datenleitung der I^2C-Schnittstelle (*Serial Data*). 17

SMAPE

Symmetrische prozentuale Abweichung (*Symmetric Mean Absolute Percentage Error*). 52

SMCLK

Taktsignal für MSP430-Subsysteme (*Sub-system Master Clock*). 16

SMU

Kombiniertes Versorgungs- und Messgerät (*Source/Measure Unit*). 79

SPI

Serielle Peripherieschnittstelle (*Serial Peripheral Interface*). 17

SRAM

Statischer Arbeitsspeicher (*Static Random Access Memory*). 15

SSR

Residuenquadratsumme (*Sum of Squared Residuals*). 13

UART

Serielle Schnittstelle (*Universal Asynchronous Receiver Transmitter*). 16

VCOM

Umpolbare Spannung (*Complementary Voltage*) bei LC-Displays. 74

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[AH09] Jacob Andersen und Morten Tranberg Hansen. “Energy bucket: A tool for power profiling and debugging of sensor nodes”. In: *Third International Conference on Sensor Technologies and Applications (SENSORCOMM'09)*. IEEE. 2009, S. 132–138.

[Ame11] Sharp Microelectronics of the Americas. *LCD Module Application Note*. LS013B4DN04. 2011.

[BGS12] Markus Buschhoff, Christian Günter und Olaf Spinczyk. “A unified approach for online and offline estimation of sensor platform energy consumption”. In: *8th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE. 2012, S. 1154–1158.

[BGS13] Markus Buschhoff, Christian Günter und Olaf Spinczyk. “MIMOSA, a Highly Sensitive and Accurate Power Measurement Technique for Low-Power Systems”. In: *Real-World Wireless Sensor Networks*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[DZ11] Mian Dong und Lin Zhong. “Self-constructive high-rate system energy modeling for battery-powered mobile systems”. In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011, S. 335–348.

[Fal14] Robert Falkenberg. “Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme”. Masterarbeit. TU Dortmund, 2014.

[FKL07] Ludwig Fahrmeir, Thomas Kneib und Stefan Lang. *Regression: Modelle, Methoden und Anwendungen*. Statistik und ihre Anwendungen. Springer Berlin Heidelberg, 2007.

[GP11] Gabriel Girban und Mircea Popa. “WSN testing environment with energy consumption monitoring and simulation of sensed data”. In: *6th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*. Bd. 2. IEEE. 2011, S. 842–847.

[Had+13] Khaled Haddad u. a. “Applicability of Monte Carlo cross validation technique for model development and validation using generalised least squares regression”. In: *Journal of Hydrology* 482 (2013), S. 119–128.

[Har+08] Ivaylo Haratcherev u. a. “PowerBench: A scalable testbed infrastructure for benchmarking power consumption”. In: *Int. Workshop on Sensor Network Engineering (IWSNE)*. 2008, S. 37–44.

[Hur+11] Philipp Hurni u. a. “On the Accuracy of Software-Based Energy Estimation Techniques”. In: *Wireless Sensor Networks*. Bd. 6567. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 49–64. DOI: 10.1007/978-3-642-19186-2_4.

[Ins13] Texas Instruments. *CC120X Low-Power High Performance Sub-1 GHz RF Transceivers User’s Guide*. CC120X. Rev. SWRU346B. 2013.

[Jun+12] Wonwoo Jung u. a. “DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components”. In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’12. Tampere, Finland: ACM, 2012, S. 353–362. ISBN: 978-1-4503-1426-8. DOI: 10.1145/2380445.2380502.

[KB12] Mikkel Baun Kjærgaard und Henrik Blunck. “Unsupervised Power Profiling for Mobile Devices”. In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services: 8th International ICST Conference, MobiQuitous 2011, Revised Selected Papers*. Hrsg. von Alessandro Puiatti und Tao Gu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 138–149. ISBN: 978-3-642-30973-1. DOI: 10.1007/978-3-642-30973-1_12.

[Koh+95] Ron Kohavi u. a. “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In: *Proceedings of the 14th international joint conference on Artificial intelligence*. Bd. 2. 1995, S. 1137–1145.

[Lim+13] Roman Lim u. a. “Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems”. In: *International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE. 2013, S. 153–165.

[LPS07] Insup Lee, Anna Philippou und Oleg Sokolsky. “Resources in process algebra”. In: *The Journal of Logic and Algebraic Programming* 72.1 (2007), S. 98–122.

[McC+11] John C McCullough u. a. “Evaluating the effectiveness of model-based power characterization”. In: *USENIX Annual Technical Conf.* Bd. 20. 2011.

[Mur+12] Rahul Murmuria u. a. “Mobile application and device power usage measurements”. In: *Sixth International Conference on Software Security and Reliability (SERE)*. IEEE. 2012, S. 147–156.

[Pal+15] James Pallister u. a. “Data dependent energy modelling: A worst case perspective”. In: *Computing Research Repository, arXiv* (2015).

[Pat+11] Abhinav Pathak u. a. “Fine-grained power modeling for smartphones using system call tracing”. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, S. 153–168.

[Per+08] Enrico Perla u. a. “PowerTOSSIM z: realistic energy modelling for wireless sensor network environments”. In: *Proceedings of the 3rd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*. ACM. 2008, S. 35–42.

[Pra+10] Aggeliki Prayati u. a. “A modeling approach on the TelosB WSN platform power consumption”. In: *Journal of Systems and Software* 83.8 (2010), S. 1355–1363.

[Rei10] Z Reitermanova. “Data splitting”. In: *WDS 10 proceedings of contributed papers*. Bd. 1. 2010, S. 31–36.

[Sem08] Nordic Semiconductor. *Single Chip 2.4GHz Transceiver Preliminary Product Specification v1.0*. nRF24L01+. 2008.

[Sem15] NXP Semiconductors. *Digital temperature sensor and thermal watchdog*. LM75B. Rev. 6.1. Feb. 2015.

[SG16] Olaf Spinczyk und pure-systems GmbH. *Documentation: AspectC++ Language Reference*. Version 2.1. Aug. 2016. URL: http://www.aspectc.org/doc/ac-languageref.pdf (besucht am 03. 02. 2017).

[Shn+04] Victor Shnayder u. a. “Simulating the power consumption of large-scale sensor network applications”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, S. 188–200.

[Tan+02] Tat Kee Tan u. a. “High-level energy macromodeling of embedded software”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.9 (2002), S. 1037–1050.

[Ton14] David Tondorf. “Eine Experimentierumgebung zur Durchführung und Auswertung von Energiemessungen mit MIMOSA”. Bachelorarbeit. TU Dortmund, 2014.

[TRJ02] Tat Kee Tan, Anand Raghunathan und Niraj K Jha. “Embedded operating system energy analysis and macro-modeling”. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE. 2002, S. 515–522.

[VS08] Aneta Vulgarakis und Cristina Seceleanu. “Embedded systems resources: Views on modeling and analysis”. In: *32nd Annual International Conference on Computer Software and Applications (COMPSAC'08)*. IEEE. 2008, S. 1321–1328.

[XL01] Qing-Song Xu und Yi-Zeng Liang. “Monte Carlo cross validation”. In: *Chemometrics and Intelligent Laboratory Systems* 56.1 (2001), S. 1–11. ISSN: 0169-7439.

[YF09] Kenji R Yamamoto und Paul G Flikkema. “Prospector: Multiscale Energy Measurement of Networked Embedded Systems with Wideband Power Signals”. In: *International Conference on Computational Science and Engineering (CSE'09)*. Bd. 2. IEEE. 2009, S. 543–549.

[Zha+10] Lide Zhang u. a. “Accurate online power estimation and automatic battery behavior based power model generation for smartphones”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2010, S. 105–114.

[Zho+11] Hai-Ying Zhou u. a. “Modeling of node energy consumption for wireless sensor networks”. In: *Wireless Sensor Networks* 3.1 (2011), S. 18.

[ZO13] Nanhao Zhu und Ian O'Connor. “Energy measurements and evaluations on high data rate and ultra low power wsn node”. In: *10th International Conference on Networking, Sensing and Control (ICNSC)*. IEEE. 2013, S. 232–236.

[ZV16] Nanhao Zhu und Athanasios V. Vasilakos. “A generic framework for energy evaluation on wireless sensor networks”. In: *Wireless Networks* 22.4 (2016), S. 1199–1220. ISSN: 1572-8196. DOI: 10.1007/s11276-015-1033-x.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift