Applikationsspezifische Analyse und Optimierung der Energieeffizienz eingebetteter Hardware/Software-Systeme

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Jochen Zimmermann
aus Heilbronn-Neckargartach

Tübingen
2013

Tag der mündlichen Qualifikation:	21. Oktober 2013
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Oliver Bringmann
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Danksagung

Diese Arbeit entstand neben meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung „Systementwurf in der Mikroelektronik“ des Forschungsbereichs „Intelligent Systems and Production Engineering“ am FZI Forschungszentrum Informatik am Karlsruher Institut für Technologie (KIT).

Ich möchte an dieser Stelle allen danken, die mich während dieser Zeit auf alle erdenklichen Arten unterstützt und so die Entstehung dieser Arbeit ermöglicht haben.

Mein besonderer Dank gilt Herrn Prof. Dr. Oliver Bringmann für die intensive inhaltliche und fachliche Betreuung meiner Arbeit, sowie für die Übernahme des Referats dieser Dissertation. Herrn Prof. Dr. Wolfgang Rosenstiel danke ich für die Übernahme des Korreferats und seine sorgfältige Begutachtung der Arbeit, sowie für die Bereitstellung der guten Arbeitsbedingungen, die mir die Entwicklung und Umsetzung meiner Ideen erst ermöglicht haben.

Weiterhin danke ich allen Kolleginnen und Kollegen des FZI und der Universität Tübingen, sowie allen Studenten, die direkt oder indirekt an dieser Arbeit beteiligt waren. Insbesondere möchte ich hier nochmals Herrn Prof. Dr. Oliver Bringmann für die langjährige Zusammenarbeit, Führung und Anleitung, sowie für die gewinnbringenden Diskussionen danken, die wesentlich zum Gelingen dieser Arbeit beigetragen haben. Nicht zuletzt durch diese zahlreichen Gespräche und sein großes Fachwissen bekam ich wertvolle Hinweise und Anregungen, die mir bei der Erarbeitung und Durchführung meines Themas hilfreich waren.

Schließlich danke ich meiner Familie, die mich während meines Studiums und meiner gesamten Doktorandenzeit stets darin bestärkt und unterstützt hat, meinen Weg zu gehen. Ein herzlicher Dank gilt meiner Freundin Sarah für die fortwährende Unterstützung und ihr Verständnis während der arbeitsintensiven Phasen meiner Promotion.

Karlsruhe, Dezember 2013

Jochen Zimmermann

Inhaltsverzeichnis

1 Einleitung 1
1.1 Problemstellung . . . 3
1.2 Lösungsansatz und Ziele . . . 5
1.3 Gliederung der Arbeit . . . 6

2 Grundlagen 9
2.1 Allgemeine Definitionen . . . 9
2.2 Modellierung von Hardware/Software-Systemen . . . 10
2.2.1 Unified Modeling Language (UML) . . . 10
2.2.2 Systems Modeling Language (SysML) . . . 10
2.2.3 MARTE . . . 11
2.2.4 Modell-zu-Modell-Transformationen mit QVT . . . 12
2.3 Simulation von Hardware/Software-Systemen . . . 13
2.3.1 Systembeschreibungssprache SystemC . . . 13
2.3.1.1 Hierarchische Modellierung . . . 14
2.3.1.2 Kommunikation . . . 14
2.3.1.3 Nebenläufige Ausführung . . . 15
2.3.1.4 Transaction-Level Modeling (TLM) . . . 17
2.3.2 Instrumentierung von Simulationsmodellen . . . 18
2.3.3 MATLAB/Simulink . . . 19
2.3.3.1 Simulink S-Function . . . 19
2.4 Berechnungsmodelle und anwendbare Algebren . . . 20
2.4.1 Allgemeine Berechnungsmodelle . . . 20
2.4.2 Datenflussorientierte Berechnungsmodelle . . . 21
2.4.3 Synchrone Datenflussmodelle . . . 22
2.4.4 Max-Plus-Algebra . . . 23
2.5 Mehrkern-Architekturen . . . 24
2.5.1 Homogene Architekturen . . . 24
2.5.2 Heterogene Architekturen . . . 27
2.5.3 Parallelismus . . . 29
2.6 Leistungsaufnahme in integrierten Schaltungen . . . 30
2.7 Modelle zur Abbildung der Leistungsaufnahme . . . 33
2.7.1 Zustandsbasierte Leistungsmodelle . . . 33
2.7.2 Instruktionsbasierte Leistungsmodelle . . . 36

2.8 Strategien zur Reduzierung der Leistungsaufnahme 37
2.8.1 Dynamic Voltage and Frequency Scaling 37
2.8.2 Dynamic Power Management 39
2.9 Lösung von ganzzahligen Optimierungsproblemen 40
2.9.1 Branch-and-Bound-Verfahren 40
2.10 Lösung von nichtlinearen Optimierungsproblemen 41
2.10.1 Grundbegriffe 41
2.10.2 Optimalitätsbedingungen 42
2.10.2.1 Lineare Nebenbedingungen 42
2.10.2.2 Nichtlineare Nebenbedingungen 44
2.10.3 Quadratische Optimierungsprobleme 44
2.10.3.1 Newton-Verfahren 45
2.10.3.2 Quasi-Newton-Verfahren 45
2.10.3.3 Active-Set 46
2.10.4 Sequential Quadratic Programming 48

3 Stand der Technik 51
3.1 Simulation und Analyse eingebetteter Hardware/Software-Systeme . . . 51
3.1.1 Generierung und Verwendung virtueller Prototypen 52
3.1.2 Integration domänenspezifischer Simulationsumgebungen 54
3.2 Optimierung der Energieeffizienz 54
3.2.1 Optimierung zur Entwurfszeit 56
3.2.2 Optimierung zur Laufzeit 58
3.3 Zusammenfassung der offenen Probleme 59

4 Konzept zur Optimierung der Energieeffizienz auf Systemebene 61
4.1 Voraussetzungen und Herausforderungen 63
4.1.1 Einsatz von Mehrkern-Architekturen 63
4.1.2 Geeignetes Berechnungsmodell 64
4.1.3 Performanzanforderungen in Echtzeitsystemen 65
4.1.4 Anwendung von Low-Power-Strategien 65
4.1.5 Abbildung der Leistungsaufnahme 66
4.2 Vorgeschlagener Lösungsansatz 68
4.2.1 Definitionen 68
4.2.2 Analyse- und Optimierungsansatz 69

5 Modellierung und Simulation digitaler Hardware/Software-Systeme 75
5.1 Modellierung 76
5.1.1 Modellierung der Funktionalität 77
5.1.2 Modellierung der Hardware-Plattform 78
5.1.3 Modellierung von Abhängigkeiten und Deployment 79
5.1.4 Spezifizierung nicht-funktionaler Eigenschaften 81
5.2 Aufbau einer virtuellen Ausführungsplattform 83
5.2.1 Software-Schicht 83
5.2.2 Integrations- und Verwaltungsschicht 85

5.2.3 Hardware-Schicht . . . 87
5.3 Generierung eines ausführbaren Systemmodells . . . 91
5.3.1 Abbildung des Systemmodells auf ein Komponentenmodell . . . 93
5.3.2 Transformation des Komponentenmodells in ein AST-Modell . . 97
5.3.3 Verbindung zwischen Modellen und Quelltext . . . 100
5.4 Optimierung der Energieeffizienz in der Simulation . . . 102
5.4.1 Ausführung von Simulationsmodellen mit NFP-Annotationen . 103
5.4.2 Reaktives Energiemanagement in der Simulation . . . 109
5.4.3 Analyse des dynamischen Verhaltens in Applikationen . . . 113

6 Optimierung der Energieeffizienz digitaler Hardware/Software-Systeme 115
6.1 Optimierung zur Entwurfszeit . . . 116
6.1.1 Formulierung als mathematisches Optimierungsproblem . . . 116
6.1.1.1 Zielfunktion . . . 117
6.1.1.2 Nebenbedingungen . . . 124
6.1.2 Lösung des Optimierungsproblems . . . 127
6.1.2.1 Aufspaltung in Teilprobleme . . . 128
6.1.2.2 Lösung des relaxierten Optimierungsproblems . . . 130
6.1.3 Experimentelle Ergebnisse . . . 132
6.1.4 Abstraktion des Optimierungsproblems . . . 134
6.1.4.1 Formulierung als lineares Optimierungsproblem . . . 135
6.1.4.2 Fazit der Abstraktion des Optimierungsproblems . . . 138
6.2 Optimierung während der Laufzeit . . . 138
6.2.1 Modellierung gegenseitiger Abhängigkeiten . . . 139
6.2.2 Multi-Domänen-Szenarien zur Abbildung dynamischen Verhaltens 141
6.2.2.1 Software-basierte Eigenschaften . . . 142
6.2.2.2 Hardware-basierte Eigenschaften . . . 143
6.2.3 Mathematisches Modell zur Erstellung des Zustandsraums . . . 145
6.2.4 Exploration des Zustandsraums . . . 149
6.2.4.1 Regelbasierte Limitierung des Zustandsraums . . . 152
6.2.4.2 Heuristiken zur Strategieableitung während der Laufzeit 156

7 Anwendung und Ergebnisse 161
7.1 Konfiguration und Optimierung auf virtuellen Plattformen . . . 162
7.1.1 Automatische Generierung der Ausführungsplattform . . . 162
7.1.2 Ausführung und Energiemanagement in virtuellen Prototypen . 163
7.2 Simulative Analyse Cyber-Physischer Systeme . . . 167
7.2.1 Plattform zur Co-Simulation . . . 168
7.2.1.1 Aufbau der Co-Simulationsplattform . . . 168
7.2.1.2 Synchronisation . . . 169
7.2.1.3 Ablauf der Co-Simulation . . . 172
7.2.2 Interaktion von eingebettetem System und Umgebung . . . 173
7.2.2.1 Einfluss der E/E-Architektur auf Regelsysteme . . . 173
7.2.2.2 Exploration von Konfigurationsparametern . . . 176

7.2.2.3 Temperaturentwicklung auf Steuergeräte/Chip-Ebene . 177
7.3 Energiemanagement auf dem Intel SCC 180
7.3.1 Optimierung zur Entwurfszeit 183
7.3.2 Optimierung zur Laufzeit 187

8 Zusammenfassung und Ausblick 195
8.1 Beitrag dieser Arbeit 195
8.2 Ausblick 196

A Quellcode-Auszüge 199
A.1 Regelbasierte Modell-zu-Modell-Transformation 199
A.2 Template-basierte Modell-zu-Text-Transformation 201

Abkürzungen und Symbole 205

Abbildungsverzeichnis 209

Tabellenverzeichnis 213

Verzeichnis der Definitionen 215

Verzeichnis der Algorithmen 217

Literaturverzeichnis 219

Index 231

Kapitel 1

Einleitung

Elektronische Systeme spielen im täglichen Leben eine immer wichtigere Rolle. Insbesondere trifft dies für digitale eingebettete Systeme zu, die mikro- bzw. nanoelektronische Hardwarekomponenten integrieren, um bestimmte Funktionalitäten durch die Ausführung dedizierter Softwareprogramme zu erbringen. Eingebettete Hardware/Software-Systeme sind dadurch zum führenden Mittel für Innovationen und Wachstum in nahezu allen Anwendungsdomänen geworden. Besonders der Markt für mobile Kommunikationsendgeräte hat in jüngster Vergangenheit enorme Zuwachsraten erfahren. So wurden laut dem Marktforschungsunternehmen Gartner Inc. im ersten Quartal 2011 ca. 428 Millionen mobile Kommunikationsendgeräte verkauft, was einer Steigerung um 19% im Gegensatz zum ersten Quartal des Vorjahrs entspricht. Noch stärker war das Wachstum in der Sparte der sogenannten Smartphones, also Mobiltelefonen mit erweiterten und komplexen Funktionalitäten. Laut einer internationalen Studie der IDC[1] wuchs der Markt für Smartphones im Jahr 2011 um ca. 49% im Vergleich zum Vorjahr – umgerechnet entspricht dies etwa 450 Millionen Smartphones im Gegensatz zu 303 Millionen im Jahre 2010. Im Jahr 2012 hat sich diese Zahl auf 700 Millionen noch einmal drastisch gesteigert. Das liegt nicht zuletzt an den großen Entwicklungsschritten, die bei der Leistungsfähigkeit der verwendeten Hardware-Plattformen in naher Vergangenheit erzielt wurden. Den Marktforschern von Strategy Analytics nach betrug der Umsatz, der durch den Verkauf sogenannter Applikationsprozessoren auf ARM®-Basis wie der Snapdragon™ S4 von Qualcomm®, Samsung® Exynos™, Texas Instruments® OMAP™ oder Apple® A6™ im ersten Halbjahr 2012 insgesamt 5,5 Milliarden US-Dollar. Mit hohen Zuwachsraten werden eingebettete Hardware/Software-Systeme auch in der Industrieautomatisierung, der Luftfahrttechnik und im Automobilbereich eingesetzt – hier vor allem in Steuergeräten, die unter anderem in aktiven und passiven Sicherheitssystemen zum Einsatz kommen, sowie im Infotainment-Bereich. In letzterem wird eine Zuwachsrate der Umsätze der Halbleiterindustrie von 3–7% über die nächsten 5 Jahre prognostiziert. Der ebenfalls in der Anwendungsdomäne Automobil feststellbare Trend zu hybriden und vollelektrischen Antriebskonzepten dürfte zu einem weiteren Schub führen, da eine komplexe Verwaltung der zur Verfügung stehenden Energie und aller beteiligten Komponenten in Form eingebetteter Systeme benötigt wird.

[1] International Data Corporation

Eine derartige Entwicklung der zugrunde liegenden Hardware-Plattformen zu leistungsfähigen Rechensystemen ist jedoch mit nicht zu vernachlässigbaren Schwierigkeiten verbunden. Vor allem durch den großen Anteil an mobilen Systemen ist deren Energieversorgung aufgrund des limitierten Energiespeichers zu einem primären Problem und damit zu einem entscheidenden Wettbewerbskriterium im Markt geworden. Die eingebetteten Hardware/Software-Systeme müssen trotz strenger Anforderungen immer komplexere und zum Teil hochgradig vernetzte Aufgaben erfüllen. Bei mobilen Kommunikationsendgeräte oder Tablet-Computern erhöht die Nachfrage nach einem schmalen Formfaktor, einer hohen Konnektivität unter Nutzung mehrerer Übertragungstechnologien und einer funktionsreichen Benutzerschnittstelle die Sensitivität bezüglich der Leistungsaufnahme, die erheblichen Einfluss auf die Temperaturentwicklung der in den Systemen integrierten Schaltungen hat, und den Bedarf an elektrischer Energie zusätzlich. Sensoren sind hier zu einem wesentlichen Teil der Interaktion zwischen Mensch und Maschine geworden, da sie es dem System ermöglichen, sich an die Umgebung und den Kontext anzupassen. Prominente Beispiele hierfür sind situationsabhängige Benutzereingaben, lokalisationsabhängige Dienste, sowie Anwendungen zur visuellen, sprachlichen oder bewegungsorientierten Erkennung und Verarbeitung.

Energieeffizienz ist jedoch nicht mehr nur ein Aspekt, der mobile Kommunikationsgeräte betrifft, sondern ist in allen Anwendungsdomänen allgegenwärtig. Selbst Automobile entwickeln sich nicht zuletzt durch die Einführung komplexer Fahrerassistenzsysteme immer mehr zu mobilen vernetzten Rechensystemen, was besonders an der steigenden Anzahl der darin enthaltenen Berechnungsknoten bzw. Kommunikationsarchitekturen, aber auch am großen Anteil eingebetteter Software erkennbar ist. Die zur Ausführung dieser Software verwendeten Hardware-Plattformen können nicht mehr – wie noch in der Vergangenheit üblich – mehrere Technologieschritte hinter den aktuellen Entwicklungen der Halbleiterindustrie liegen, sondern müssen den neuesten Generationen entsprechen. Im Sinne der Energieeffizienz sind die technologisch-bedingten Potentiale ebenso weitgehend ausgeschöpft, wie dies in anderen Anwendungsdomänen mit starkem Bezug zur Halbleiterindustrie der Fall ist.

Die wachsenden Anforderungen der Kunden an die Qualität der Systemfunktionalität implizieren zusätzliche Herausforderungen an die Energieversorgung der Systeme. Können diese Herausforderungen auf Seiten der Systemhersteller nicht entsprechend bewältigt werden, kann es im Extremfall dazu führen, dass auf zusätzliche Funktionalität verzichtet werden muss, um ein spezifiziertes Budget für den Energieverbrauch einer oder mehrerer Komponenten nicht zu überschreiten. Im Umkehrschluss kann es auch vorkommen, dass Anwendungen die an sie gestellten und teilweise sicherheitskritischen Anforderungen nicht mehr erfüllen können, da einzelne Funktionseinheiten aufgrund der aktuellen Energiesituation nicht mit der maximal zur Verfügung stehenden Leistungsfähigkeit betrieben werden können – oder im schlimmsten Fall sogar ganz ausgefallen sind. Können gesetzte Budgets für Leistungsaufnahme und Energieverbrauch, also die Leistungsaufnahme über eine bestimmte Zeitspanne, nicht eingehalten werden, drohen umfangreiche und ressourcenintensive Entwurfsiterationen, welche gleichzeitig die Entwicklungszeit erhöhen und meist unweigerlich zu einer späteren Verfügbarkeit der Produkte am Markt führen. Gleichzeitig müssen die Systeme im

praktischen Einsatz die hohen Erwartungen der Kunden an deren Qualität, aber auch deren Energieeffizienz erfüllen.

Systemarchitekten, Zulieferer von Teilsystemen, sowie Systemintegratoren und Hersteller der Gesamtsysteme sind deswegen gezwungen, ihre Entwurfsprozesse so abzustimmen und zu modifizieren, dass der Energieverbrauch schon während der Spezifikation und Entwurf der eingebetteten Hardware/Software-Systeme berücksichtigt und dessen Minimierung als wichtiges Entwurfsziel in die folgenden Prozesse integriert wird. Dies kann nur anhand durchgängiger Entwicklungsabläufe erreicht werden, indem die Leistungsaufnahme des Systems als entscheidender Faktor zur Bestimmung des Energieverbrauchs bewertet und für dessen Nutzung beim Kunden optimiert wird.

1.1 Problemstellung

Die Leistungsaufnahme elektronischer Systeme ist zu einem Schlüsselfaktor der Performanz dieser Systeme geworden, da sie sowohl die Temperaturentwicklung, als auch den Bedarf an elektrischer Energie bestimmt. Diese Feststellung gilt offensichtlich für mobile Endgeräte wie Smartphones und Tablet-Computer, bei denen sich aufgrund des Formfaktors eine erhöhte Leistungsaufnahme besonders negativ auf die Temperaturentwicklung auswirkt, aber auch für integrierte Systeme wie Set-Top-Boxen und moderne Fernsehgeräte. Die bisherigen Lösungen der Halbleiterindustrie zur Verringerung der Leistungsaufnahme basierten auf der stetigen Reduzierung der Strukturbreiten, um die Versorgungsspannung senken zu können, die einen quadratischen Anteil an der Leistungsaufnahme der integrierten Schaltung hat. Kleine Änderungen führten somit schon zu einer deutlichen Minimierung. In den letzten Jahren hat man sich aber hier einer technisch-bedingten Grenze so weit angenähert, dass in naher Zukunft keine deutlichen Verbesserungen zu erwarten sind. Die Versorgungsspannung einer integrierten Schaltung bestimmt neben der Leistungsaufnahme auch deren maximale Taktfrequenz. Wird eine hohe Leistungsfähigkeit des Systems gefordert, wird diese wegen der eben genannten Limitierung der Frequenz einer Ausführungseinheit zumeist durch die Integration mehrerer Ausführungseinheiten in einem System oder sogar auf einem Chip – sogenannte System-on-Chip (SoC) bzw. Multiprocessor-System-on-Chip (MPSoC) im Falle mehrerer Berechnungskerne – erreicht. Diese Entwicklung setzt jedoch die effiziente Nutzung parallel arbeitender Ressourcen voraus. Die durch die steigende Leistungsausnahme verursachten Problem werden damit jedoch in die Software-Welt verlagert und sind aufgrund der Schwierigkeiten bei der Programmierung von parallel ausführbarer Software nicht vollständig zu lösen [28].

Gleichzeitig gelten aber auch immer häufiger Begrenzungen bezüglich des Energiebedarfs dieser Plattformen, sowohl durch Kundenwünsche als auch durch politische und soziale Vorgaben. Es gilt deshalb, das System an das Ungleichgewicht zwischen Phasen der Benutzung, in denen eine hohe Leistungsfähigkeit gefordert ist und denen, die diese nicht erfordern und damit Raum zur Minimierung des Energieverbrauchs bieten, gezielt anzupassen. Neueste technische Lösungen existieren in Form von heterogenen

Architekturen, in denen unterschiedliche Typen von ausführenden Einheiten eingesetzt werden können, um so eine möglichst gute Adaption an die situationsbedingten Anforderungen zu erreichen. Diese Verteilung kann allerdings nur so gut funktionieren, wie es die steuernde Hardware-nahe Software und das Energiemanagement erlauben. Zusätzlich zu diesen rein technischen Lösungen müssen die Applikationen so geplant und ausgeführt werden, dass deren Energieeffizienz maximiert wird. Die dadurch erreichte Optimierung der Energieeffizienz muss möglichst früh im Entwurfsprozess durchgeführt werden, da Änderungen im Energiemanagement schwieriger und eingeschränkter werden, je weiter der Entwurfsprozess vorangeschritten ist. Vor allem aber ist es nur auf hohen Abstraktionsebenen möglich, sowohl vielversprechende Entscheidungen bezüglich des Energiemanagements zu treffen als auch einzelnen Subsystemen oder Funktionen ein bestimmtes Budget am Energiehaushalt zuzuweisen. Analysen des Energieverbrauchs von Hardware/Software-Systemen und des daraus abgeleiteten Optimierungspotentials gehen davon aus, dass je nach Anwendungsfall bis zu 70% der Energieeinsparung auf Systemebene erreicht werden, wohingegen das Optimierungspotential auf niedrigeren und Technologie-nahen Abstraktionsebenen bei 10-20% anzusiedeln ist. Optimierungen auf Hardware-Ebene werden in der Regel durch die Integration von Mechanismen wie Power- oder Clock-Gating, was zum zeitweisen Abschalten bestimmter Teile der Schaltung führt, oder die Verwendung unterschiedlicher Versorgungsspannungen während der Synthese erreicht. Entscheidend ist jedoch die Frage, wie Strategien für den effizienten Einsatz dieser Mechanismen abgeleitet werden können.

Der Aufwand für die Entwicklung geeigneter Strategien des Energiemanagements steigt mit der Komplexität des zu steuernden Systems. Je nach Verwendung bzw. Art des Systems stecken bis zu 25% der Entwicklungszeit in der Bewertung und Reduzierung der Leistungsaufnahme. Durch die mitunter sehr verschiedenen Nutzungsszenarien eingebetteter Hardware/Software-Systeme, z.B. im Mobilkommunikationsbereich, ergibt sich entsprechend eine Vielzahl von Szenarien für das Energiemanagement. Als Paradebeispiel kann hier eine dauerhaft aktivierte Sprachsteuerung herangezogen werden, die bei absichtlichem Gebrauch eine hoch komplexe Funktionalität darstellt und deswegen hohe Anforderung an die Performanz des ausführenden Systems stellt, ansonsten aber möglichst energieeffizient implementiert sein sollte. Um möglichst viel Potential ausschöpfen zu können, sollte die Steigerung der Energieeffizienz basierend auf statischen und bereits zur Entwurfszeit feststehenden Eigenschaften mit einem dynamisches Energiemanagement während der Laufzeit kombiniert werden.

Die hier identifizierten Herausforderungen an ein Verfahren zur Optimierung der Energieeffizienz können also in folgenden Punkten zusammengefasst werden:

- Einhaltung der an das System gestellten Anforderungen
- Berücksichtigung der zugrunde liegenden Hardware-Plattform und der auszuführenden Applikationen
- Simulative Analyse der Leistungsaufnahme aller beteiligter Komponenten
- Berücksichtigung des dynamischen Verhalten während Simulation und Analyse

- Steigerung der Energieeffizienz basierend auf statischen Eigenschaften zur Entwurfszeit
- Dynamisches Energiemanagement während der Laufzeit zur Anpassung an das situationsbedingte Systemverhalten

Im nachfolgenden Abschnitt soll ein Ansatz zur Lösung dieser Herausforderungen bezüglich der Optimierung der Energieeffizienz eingebetteter Hardware/Software-Systeme skizziert werden, der analytische und simulative Ansätze kombiniert. Die in dieser Arbeit verwendeten Begriffe für *Energieeffizienz* und deren Optimierung beziehen sich dabei ausschließlich auf die zur Erbringung einer definierten Funktionalität innerhalb eines vorgegebenen Zeitraums benötigte Energie. Eine genaue Definition hierfür ist in Abschnitt 2.1 enthalten.

1.2 Lösungsansatz und Ziele

In dieser Arbeit soll durch eine modellbasierte Analyse der auszuführenden Funktionalität eine applikationsspezifische Optimierung der Energieeffizienz unter Berücksichtigung der zugrunde liegenden Hardware-Plattform durchgeführt werden. Voraussetzung hierfür ist, dass die Zielarchitektur über mehrere Betriebszustände verfügt, die verschiedene Charakteristika bezüglich der Leistungsaufnahme und Ausführungsperformanz besitzen. Wie in Abbildung 1.1 zu sehen ist, spielt die Optimierung der Energieeffizienz über mehrere Phasen hinweg eine entscheidende Rolle, angefangen von der Spezifikation, über die Entwicklung und die Verifikation, bis hin zum letztendlichen Betrieb der eingebetteten Hardware/Software-Systeme.

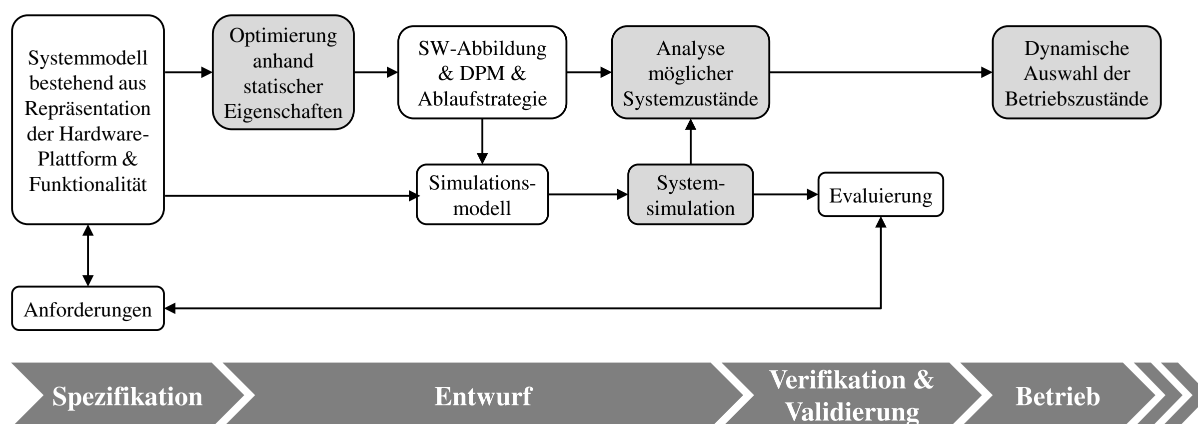


Abbildung 1.1: Lösungsansatz zur Analyse und Optimierung der Energieeffizienz

Ausgehend von einem Systemmodell, das aus der in Software zu realisierenden Funktionalität und der zugrunde liegenden Hardware-Plattform besteht, wird durch den gezielten Einsatz eines dynamischen Energie- bzw. Powermanagements (DPM) eine Optimierung der Energieeffizienz angestrebt, welche zunächst die statischen Eigenschaften des Systemmodells und der an das System gestellten Anforderungen einbezieht. Der Ansatz zur statischen Optimierung basiert auf der mathematischen

Formulierung eines Optimierungsproblems, das durch eine zielgerichtete Anpassung von bereits in Forschung und Industrie etablierten Algorithmen auf das hier adressierte Optimierungsproblem gelöst werden soll. Ziel dieser Optimierung ist sowohl die energieeffiziente Abbildung der spezifizierten Applikationen auf ausführende Berechnungseinheiten, als auch deren zeitliche Ablaufplanung, sowie die Zuordnung von auf der jeweiligen Hardware-Plattform zur Verfügung stehenden Betriebszuständen unter strikter Berücksichtigung der Performanzanforderungen an die Applikationen.

Weiterhin soll durch die Berechnung des energieeffizienten Ablaufs und des Systemmodells eine automatische und dadurch schnelle Generierung eines Simulationsmodells möglich sein. Dies erlaubt eine frühzeitige simulative Analyse der Leistungsaufnahme des Gesamtsystems, was die Umgebung des Hardware/Software-Systems einschließt und damit die Integration zusätzlicher Komponenten, Sensoren oder Aktoren erlaubt. Die Simulation des Gesamtsystems kann zusätzlich zur Evaluierung der funktionalen und nicht-funktionalen Eigenschaften wie Zeitverhalten und Leistungsaufnahme verwendet werden, womit sowohl die Anforderungen als auch eventuell spezifizierte Budgets dieser nicht-funktionalen Eigenschaften früh im Entwicklungsprozess überprüft werden können. Ebenso können mithilfe des Simulationsmodells die Auswirkungen der vorangegangenen Optimierungsschritte aufgezeigt werden.

Zusätzlich können während der Simulation typische Nutzungsszenarien der Applikationen extrahiert werden, falls diese nicht bereits in der Spezifikationsphase bekannt sind. Anhand dieser Szenarien soll die Energieeffizienz weiter optimiert werden, sodass während des Betriebs eine Strategie für das Energiemanagement angewandt bzw. eine dynamische Anpassung dessen vorgenommen werden kann, welche die dynamischen Eigenschaften des Hardware/Software-Systems zur Steigerung der Energieeffizienz verwendet. Insgesamt werden dadurch sowohl statische als auch dynamische Aspekte in den Optimierungsprozess integriert.

1.3 Gliederung der Arbeit

Die weiteren Inhalte dieser Arbeit sind in die nachfolgend beschriebenen Kapitel gegliedert. Theoretische Ergebnisse und entsprechende Schlussfolgerungen sind bereits in den jeweiligen Inhaltskapiteln enthalten.

In Kapitel 2 sind zunächst die für das weitere Verständnis benötigten Grundlagen enthalten. Hierbei werden insbesondere Techniken und Methodiken zur Modellierung und Simulation eingebetteter Hardware/Software-Systeme, aktuelle Architekturen mit mehreren Berechnungseinheiten, Techniken zur Reduzierung der Leistungsaufnahme dieser Architekturen, sowie mathematische Grundlagen zur Optimierungstheorie erläutert.

Kapitel 3 stellt bereits existierende Ansätze zur Verwendung virtueller Prototypen und der Optimierung der Energieeffizienz in Hardware/Software-Systemen vor. Insbesondere werden darin die Probleme dieser bestehenden Ansätze herausgearbeitet.

Kapitel 4 erläutert das in dieser Arbeit verfolgte Konzept zur Simulation und Analyse digitaler Hardware/Software-Systeme mit dem Ziel der Optimierung der Energieeffizi-

enz. Insbesondere werden hier auch die Voraussetzungen und die Herausforderungen bei der Verwendung moderner Mehrkern-Architekturen, sowie die in diesen Architekturen bereitgestellten Techniken zur Reduzierung der Leistungsaufnahme und deren Modellierung thematisiert.

In Kapitel 5 wird die Transformation eines in UML/SysML modellierten Systemmodells in einen virtuellen Prototypen bzw. in ein Simulationsmodell in SystemC beschrieben. Hierbei wird besonders die Umsetzung der Ausführungssemantik und die Steuerung der Simulation durch eine Integrations- und Verwaltungsschicht dargestellt, die den Ablauf und das Energiemanagement während der Ausführung mehrerer Applikationen beinhaltet.

Kapitel 6 präsentiert die in dieser Arbeit entwickelten Verfahren zur modellbasierten Optimierung der Energieeffizienz eingebetteter Hardware/Software-Systeme. In einem ersten Optimierungsprozess werden dabei die statischen Eigenschaften zur Berechnung eines energieeffizienten Ablaufs der Applikationen verwendet. In einem weiteren Schritt werden zusätzlich die dynamischen Aspekte des modellierten Systems zur Ableitung einer Strategie des Energiemanagements einbezogen, um eine Optimierung zur Laufzeit zu erlauben.

In Kapitel 7 werden die Anwendung der Systemsimulation und der Optimierung anhand mehrerer Anwendungsszenarien dargestellt und entsprechende Ergebnisse basierend auf einer realen Hardware-Plattform präsentiert.

Kapitel 8 bildet den Abschluss dieser Arbeit, indem zuerst eine Zusammenfassung der dargestellten Inhalte erfolgt und anschließend ein Ausblick auf weiterführende Aktivitäten in diesem Themengebiet geboten wird.

In Anhang A sind zur Vervollständigung einige Quellcode-Auszüge enthalten, um die im Verlauf der Arbeit gezeigten Vorgehensweisen und Algorithmen anhand deren Implementierung zu verdeutlichen.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die Grundlagen für ein erweitertes Verständnis dar und umfasst nicht nur fundamentale Definitionen von Begriffen, sondern liefert detaillierte Erklärungen zu Sachverhalten, die in späteren Kapiteln aufgrund des Umfangs und der daraus folgenden Komplexität ohne nähere Erläuterung verwendet werden. Bei Bedarf soll auf diese Grundlagen verwiesen werden, das Kapitel kann allerdings auch unabhängig als Nachschlagewerk dienen.

Nach allgemeinen Definitionen zu den Begriffen *Energieverbrauch* und *Energieeffizienz* sind zunächst einführende und erklärende Abschnitte zur Modellierung und Simulation von eingebetteten Hardware/Software-Systemen, sowie Techniken zur Transformation von verschiedenen Arten von Modellen enthalten. Es werden außerdem aktuelle Mehrkern-Architekturen vorgestellt. Zusätzlich werden sowohl grundlegende Berechnungsmodelle zur Analyse des Systemverhaltens als auch Modelle zur Repräsentation der Leistungsaufnahme in eingebetteten Systemen erklärt. Weiterhin wird die Leistungsaufnahme in integrierten Schaltungen erklärt und Mechanismen zur Reduzierung der Leistungsaufnahme vorgestellt. Abschließend wird eine grundlegende Einführung in die Theorie der ganzzahligen linearen, quadratischen und nicht-linearen Optimierung gegeben.

2.1 Allgemeine Definitionen

In diesem Abschnitt werden zunächst allgemeine und im Verlauf dieser Arbeit häufig verwendete Begriffe definiert. Es sollte beachtet werden, dass die hier dargestellten Definitionen auf unterschiedliche Domänen angewandt werden können, sodass eine Übertragung auf den hier adressierten Bereich digitaler Hardware/Software-Systeme notwendig ist. Weiterhin bauen die Definitionen teilweise aufeinander auf.

Definition 2.1 (Energieverbrauch)
Da rein physikalisch lediglich eine Umwandlung von Energie in unterschiedlichen Formen existiert, soll die Sichtweise bei Hardware/Software-Systemen auf die zugeführte elektrische Energie beschränkt sein. Der Energieverbrauch stellt somit den Bedarf an elektrischer Energie dar, der von der ausführenden Hardware-Plattform für die Erbringung einer bestimmten Funktionalität benötigt wird.

Definition 2.2 (Energieeffizienz)
Nach Auffassung der Europäischen Union[1] *beschreibt die Energieeffizienz das Verhältnis von Ertrag an Leistung, Dienstleistung, Waren oder Energie zu Energieeinsatz [96]. In dieser Arbeit wird zur Bewertung der Energieeffizienz ausschließlich die zur Erbringung einer definierten Funktionalität innerhalb eines vorgegebenen Zeitraums zugeführte elektrische Energie bzw. der damit verbundene Energieverbrauch herangezogen.*

Definition 2.3 (Optimierung der Energieeffizienz)
Unter einer Optimierung der Energieeffizienz wird die Reduktion des Energiebedarfs zur Erbringung einer definierten Funktionalität innerhalb eines vorgegebenen Zeitraums verstanden, welche durch die strategische Anwendung eines dynamischen Power- bzw. Energiemanagements erzielt wird.

2.2 Modellierung von Hardware/Software-Systemen

2.2.1 Unified Modeling Language (UML)

Die *Unified Modeling Language* ist eine durch die Object Management Group (OMG) standardisierte Sprache zur Modellierung von objektorientierten Softwarearchitekturen. UML ist momentan in der neuesten Version 2.4.1 verfügbar [92]. Das Ziel von UML ist die Unterstützung des gesamten Softwareentwicklungsprozesses sowohl in der Entwurfs- und Implementierungsphase, als auch zur Verifikation und Dokumentation von Software. Dazu werden in UML die Syntax und Semantik verschiedener Elemente wie *Klassen*, *Variablen* und *Operationen*, aber auch strukturelle Elemente wie *Ports*, sowie Beziehungen und Abhängigkeiten untereinander, wie z.B. Vererbungsbeziehungen oder Enthalten-sein-Relationen, definiert. Die semantische Bedeutung von Elementen kann durch die Verwendung von UML-Profilen und darin enthaltene UML-Stereotypen präzisiert werden. Beispielsweise können Klassen dadurch als Schnittstellen oder als *UML-Komponenten* mit wohldefiniertem Zugriff auf diese Schnittstellen durch UML-Ports definiert werden. Weiterhin werden grafische Notationen der strukturellen Einheiten zur Verwendung in standardisierten UML-Diagrammen, wie z.B. Klassen-, Objekt- oder Kompositionsstrukturdiagramme, spezifiziert. UML-Diagramme sind generell in Strukturdiagramme, die hauptsächlich den statischen Aufbau von Architekturen und Systemen vorgeben, und Verhaltensdiagramme unterteilt. Letzte repräsentieren das dynamische Verhalten der modellierten Objekte durch Aktivitäts-, Zustands- oder Sequenzdiagramme. Abbildung 2.1 zeigt eine Übersicht der in UML zur Verfügung stehenden Diagrammtypen.

2.2.2 Systems Modeling Language (SysML)

Die *Systems Modeling Language (SysML)* ist eine Profilerweiterung zur Unified Modeling Language (UML) und ist von der Object Management Group (OMG) in [93] standardisiert. Sie erweitert die UML in der Version 2.3 sowohl um Sprachkonstrukte

[1] Energiedienstleistungsrichtlinie der Europäischen Union (2006)

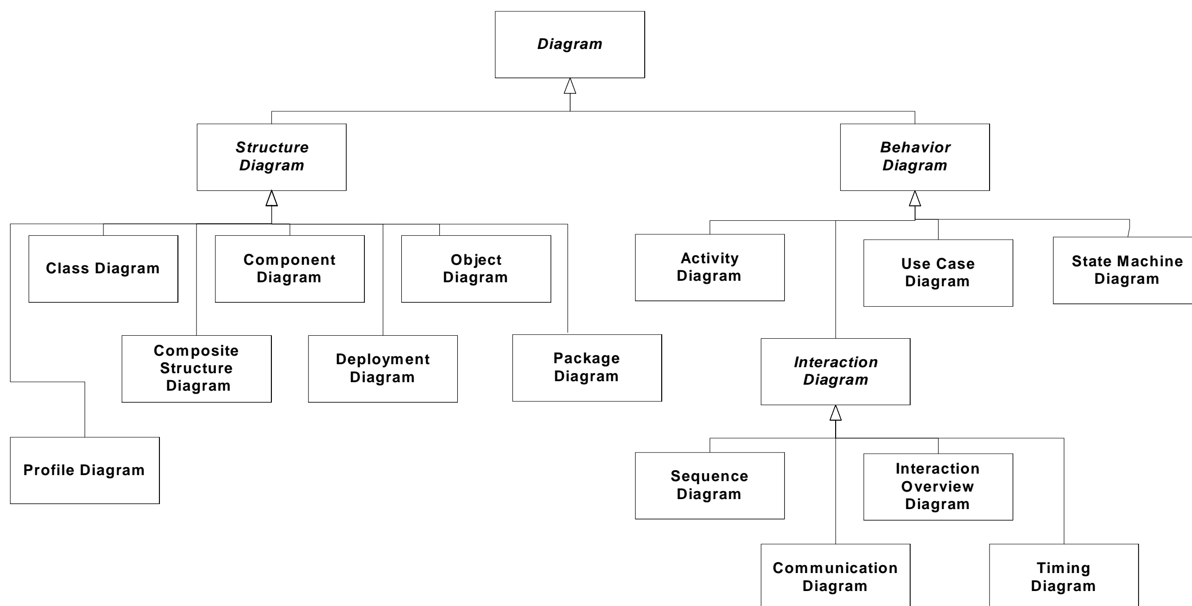


Abbildung 2.1: Diagramme der Unified Modeling Language (UML) (Bildquelle: [92])

als auch um Diagrammtypen zur Modellierung komplexer und zumeist auch realer Systeme, wohingegen UML zwar eine allgemeine Modellierungssprache darstellt, jedoch oftmals zur Modellierung des gedanklichen Aufbaus von Architekturen oder deren Verhaltensmuster verwendet wird. Dieser Unterschied wird auch durch die durch SysML definierten Sprachkonstrukte deutlich. SysML erweitert das Konzept von Klassen um die Notation von Blöcken, die die Hauptelemente des zu modellierenden Systems bilden. Ein *SysML-Block* kann wie eine UML-Klasse hierarchisch angeordnet sein, sodass dieser wiederum ein oder mehrere Blöcke enthalten kann. Als Schnittstellen bzw. Interaktionspunkte von Blöcken definiert SysML sogenannte *Flow-Ports* als Erweiterung von Ports in UML. Zur Verminderung der Komplexität in hierarchischen Modellen wird die Definition eines Blocks und dessen internen Aufbau in getrennten Diagrammen modelliert. Dafür werden in SysML das *Blockdefinitionsdiagramm* und das *Interne Blockdiagramm* spezifiziert. Abbildung 2.2 zeigt die Erweiterungen durch SysML gegenüber dem UML-2-Standard.

2.2.3 MARTE

Mit *MARTE* (Modeling and Analysis of Real Time and Embedded Systems) besteht ein Profil für den UML-2-Standard, das für die Modellierung von Echtzeitsystemen und eingebetteten Systemen entwickelt wurde [91]. Vorrangiges Ziel von MARTE ist die Unterstützung eines modellbasierten Entwurfs und der modellbasierten Entwicklung von Echtzeitsystemen anhand eines UML-Profils, das von der Object Management Group (OMG) standardisiert wurde. MARTE unterstützt dabei Entwurfs- und Spezifikationsprozesse, sowie die Analyse und Verifikation von Hardware/Software-Systemen, indem es einen Top-Down-Modellierungsablauf implementiert. Ausgehend von grundlegenden Kernelementen zur Beschreibung von Echtzeitsystemen werden diese sowohl zum Zweck der Modellierung – bis hin zum Anforderungsmanagement – als auch

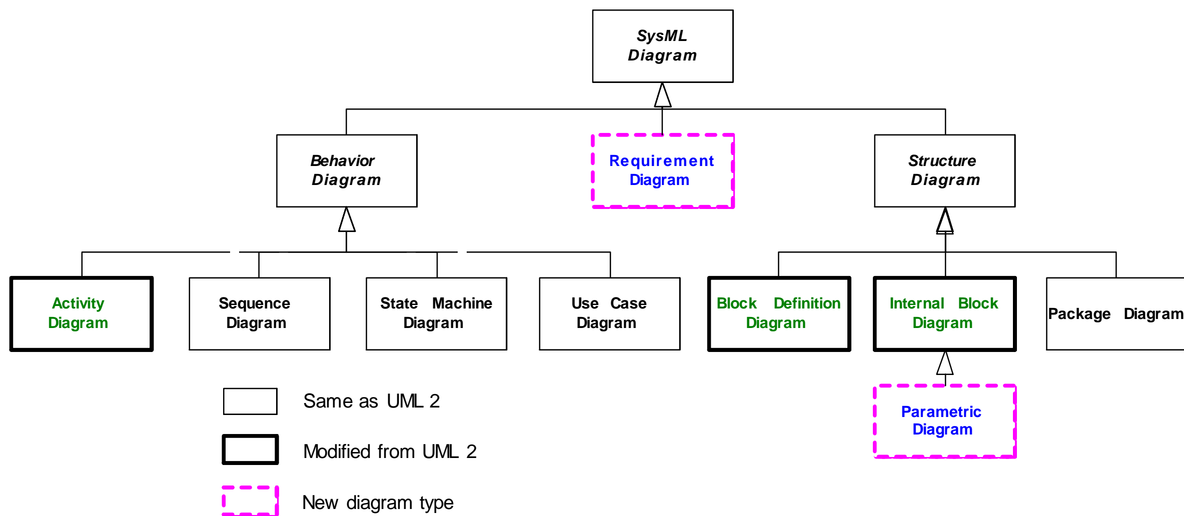


Abbildung 2.2: Erweiterung der UML-Diagrammstruktur durch SysML (Bildquelle: [93])

zur Analyse verfeinert. Dafür werden vor allem Stereotypen, aber auch funktionale und nicht-funktionale Eigenschaften spezifiziert, die die Anbindung von separaten Entwurfs- und Analysewerkzeugen ermöglichen.

2.2.4 Modell-zu-Modell-Transformationen mit QVT

Query-View-Transformations (QVT) stellt eine Sprache für Modell-zu-Modell-Transformationen basierend auf Abbildungsregeln dar und wurde von der *Object Management Group (OMG)* als Teil der *Meta Object Facilities (MOF)* formal spezifiziert [90]. Das Prinzip von QVT besteht darin, dass einzelne Elemente bzw. komplexe Elementmengen eines Modells durch formale Ausdrücke ausgewählt und durch anschließende Transformationen Beziehungen zwischen Modellen hergestellt werden. Modell-zu-Modell-Transformationen leisten damit einen wichtigen Beitrag zur modellgetriebenen Entwicklung (engl.: *Model-driven Engineering (MDE)*) bzw. stellen eine generelle Grundlage für modellgetriebene Architekturen (engl.: *Model-driven Architecture (MDA)*) dar, deren Ziel z.B. die klare Trennung zwischen plattformabhängigen und plattformunabhängigen Sichtweisen auf das System durch die Verwendung unterschiedlicher Abstraktionsebenen ist. Es existieren zwei grundsätzlich verschiedene Ansätze zur Formulierung und Umsetzung von QVT in Transformationswerkzeugen.

QVT-Relations (QVT-R) ist eine deklarative Sprache, die die Modelle auf definierte Vorbedingungen prüft und bei deren Einhaltung die damit verbundenen Transformationsschritte ausführt. Es existiert also keine bestimmte Reihenfolge, in denen die einzelnen Transformationen ausgeführt werden, sondern es werden alle entsprechenden Elemente und deren Kombinationen in den Eingabemodellen bezüglich der Vorbedingungen geprüft. Die Transformationsregeln können deshalb theoretisch auch bidirektional, also mit vertauschten Eingabe- und Ausgabemodellen angewandt werden. Die Programmierung in deklarativen Sprachen ist jedoch sehr komplex, da sie nicht der natürlichen Denkweise von Menschen und Entwicklern entspricht. Wesentlich

einfacher ist die Programmierung in imperativen Sprachen, da sie ein strukturiertes Vorgehen innerhalb der Programmstruktur ermöglichen.

Mit *QVT-Operational (QVT-O)* existiert eine imperative Sprache für QVT, die allerdings auf unidirektionale Transformationsschritte beschränkt ist. Dieser Umstand ist dadurch begründet, dass Transformationsregeln in QVT-O auf Elemente der Eingabemodelle aufgerufen werden, um die Zielmodelle zu generieren und somit keine unmittelbaren Auswirkungen auf die Eingabemodelle existieren.

2.3 Simulation von Hardware/Software-Systemen

2.3.1 Systembeschreibungssprache SystemC

Systembeschreibungssprachen wurden als Lösung für die steigenden Komplexität beim Entwurf eingebetteter Systeme entwickelt. Um den Entwurfsprozess komplexer Systeme zu vereinfachen, müssen diese unter Umständen auf sehr hohem Abstraktionsniveau betrachtet werden. Dies ist allerdings mit herkömmlichen Hardwarebeschreibungssprachen wie VHDL[2] [53] oder Verilog [54] nur in begrenztem Umfang möglich, da diese keine Softwarebeschreibungsmechanismen aufweisen. Aus dieser Notwendigkeit heraus wurde die Systembeschreibungssprache SystemC ursprünglich von der Open SystemC Initiative (OSCI) vorangetrieben, einem Zusammenschluss aus mehreren Firmen der Halbleiter-, EDA[3]- und IP[4]-Branche. Mittlerweile wird die Sprache von der Accellera Systems Initiative [3] entwickelt und durch IEEE[5] standardisiert [55]. Ziele dieser Entwicklung sind die Möglichkeit des Austauschs von IP-Modellen und die Unterstützung der Partitionierung von Hardware bzw. Software und des Co-Designs.

SystemC ist keine eigenständige Programmiersprache, sondern basiert als Erweiterungsbibliothek auf der Programmiersprache C++ [45]. So wird die Flexibilität einer objektorientierten Programmiersprache mit der Möglichkeit verbunden, aus Hardware und Software bestehende Systeme auf unterschiedlichen Abstraktionsebenen beschreiben und modellieren zu können. Die Variabilität der Abstraktionen reicht von der Beschreibung zeitbehafteter und zeitloser Prozesse, über Beschreibungen auf Transaktionsebene (TLM[6]), bis zur Register-Transfer-Beschreibung (RTL[7]) komplexer elektronischer Komponenten. Ein den Regeln entsprechendes Modell auf RTL-Ebene kann vergleichbar mit VHDL zur Hardwaresynthese eingesetzt werden.

Im Nachfolgenden werden die einzelnen Konstrukte der SystemC-Bibliothek erklärt, die zur Hardware- und Softwarebeschreibung benötigt werden, und die durch Einbinden der SystemC-Headerdatei in C++ verfügbar gemacht werden. Die in diesem Abschnitt dargelegte Beschreibung von SystemC ist teilweise aus [137] entnommen.

[2] Very High Speed Integrated Hardware Description Language
[3] Electronic Design Automation
[4] Intellectual Property
[5] Institute of Electrical and Electronics Engineers
[6] Transaction-Level Modeling
[7] Register Transfer Level

2.3.1.1 Hierarchische Modellierung

Um die Komplexität eines Systems beherrschen zu können, müssen einzelne Komponenten oder auch ganze Systeme in Hierarchien gegliedert werden. Dazu werden in SystemC sogenannte *Module* verwendet.

Module Module verbergen ihren internen Aufbau und ihre Funktionalität vor anderen Modulen und Komponenten, wodurch sie beliebig erweitert oder verändert werden können, ohne Einfluss auf das Zusammenspiel mehrerer Module zu haben. Die Kommunikation nach außen erfolgt über ihre Schnittstellen, die sogenannten *Ports* (siehe 2.3.1.2). Die Funktionalität innerhalb eines Moduls wird durch *Prozesse* (vgl. 2.3.1.3) realisiert. Module können auch hierarchisch aufgebaut sein, d.h. dass ein Modul wiederum aus verschiedenen Modulen und deren Kommunikationsverbindungen bestehen kann.

Module werden in der Hauptdatei `sc_main()` instanziiert, wodurch der Konstruktor dieses Moduls aufgerufen wird. Dieser kann durch Verwendung des SystemC-Makros `SC_CTOR` oder durch C++-Schreibweise gebildet, wenn Parameter übergeben werden sollen. In letzterer wird das entsprechende Modul von der Klasse `sc_module` abgeleitet. Für die Aktivierung der in diesem Modul enthaltenen Prozesse muss das Makro `SC_HAS_PROCESS([Modulname])` aufgerufen werden.

2.3.1.2 Kommunikation

In der reinen Hardwaremodellierung werden Hardwaresignale dazu benutzt, um die Kommunikation und Synchronisation zwischen unterschiedlichen Prozessen herzustellen. Da diese Ebene für eine Systemmodellierung meist zu niedrig ist, können in SystemC *Interfaces*, *Ports* und *Kanäle* zur Modellierung der Kommunikation verwendet werden, um diesem hohen Abstraktionsgrad gerecht zu werden.

Ports Ein Modul muss mit seiner Umgebung über wohldefinierte Schnittstellen kommunizieren können. *Ports* bilden diese Schnittstelle eines Moduls nach außen, über die die Kommunikation mit anderen Modulen erfolgt. Bei der Deklaration eines Ports muss sowohl ein Typ, als auch eine Kommunikationsrichtung (Eingang, Ausgang oder Ein- und Ausgang kombiniert) angegeben werden. Als Datentypen können hierfür C++-Typen, spezielle SystemC-Typen und benutzerdefinierte Typen verwendet werden.

Interfaces Alle in SystemC verwendeten *Interfaces* werden von der abstrakten Basisklasse `sc_interface` abgeleitet. Interfaces bestehen aus einer Menge von Operationen, von denen allerdings nur die Signatur spezifiziert ist, also die Namen der Operationen, verwendete Parameter und Rückgabewerte. Ein Interface dient als Schnittstelle für ein Kommunikationsmedium (siehe dazu 2.3.1.2), bietet dessen Funktionalität also nach außen an. Will ein Port kommunizieren, muss er über ein Interface mit dem gewünschten Kommunikationsmedium verbunden werden. Dabei ist darauf zu achten, dass die Typen von Port und dem entsprechenden Interface übereinstimmen.

Kanäle Interfaces und Ports beschreiben zwar die Funktionen, die kommunizierende Elementen nutzen können, explizit ausgeführt wird die Kommunikation aber über *Kanäle*. Kanäle implementieren also ihre zugehörigen Interfaces und geben ihnen somit Funktionalität, die diese dann nach außen hin den Ports zur Verfügung stellen können. Dadurch wird eine Trennung von Zugriff auf ein Kommunikationsmedium und dessen Transporteigenschaften geschaffen, was deren beider Wiederverwendung losgelöst vom jeweils anderen ermöglicht. So kann auf denselben Kanal durch mehrere Interfaces unterschiedlich zugegriffen werden oder etwa ein Kanal verändert oder erweitert werden, ohne dass sich an der Zugriffsart etwas ändert.

Kanäle werden in hierarchische Kanäle und primitive Kanäle unterschieden. Primitive Kanäle zeichnen sich dadurch aus, dass sie keine Hierarchien und keine Prozesse enthalten. Zu den primitiven Kanälen gehören Signale, FIFO[8]-Kanäle, sowie Mechanismen des gegenseitigen Ausschlusses wie Mutexe und Semaphoren. Hierarchische Kanäle sind vollständige SystemC-Module inklusive Prozessen, die es erlauben, komplizierte Kommunikationsprotokolle zu realisieren, deren Komplexität vor den kommunizierenden Einheiten verborgen wird. Dabei besteht die Verbindung nicht nur aus einer einfachen Übertragungsleitung, sondern es kann eine komplexe Datenkommunikation mit einbezogener Umformung oder Transaktion beschrieben werden.

2.3.1.3 Nebenläufige Ausführung

Modelle in Software sind an sich schlecht dafür geeignet, die Realität in einem echten System widerzuspiegeln. In einem solchen System laufen viele der durchgeführten Aktionen parallel ab, wohingegen Software diese geforderte Nebenläufigkeit nur durch eine Aufteilung dieser Aktionen in einzelne Threads nachbilden kann, die dann durch eine Scheduling-Politik ausgewählt, zur Ausführung gebracht und dann wieder verdrängt werden. Um Parallelität bei Ausführungen zu erhalten, muss ein dedizierter Simulationskern dafür sorgen, dass zwar mehrere Aktionen nacheinander ausgeführt werden, dabei allerdings keine Zeit innerhalb des Simulationsmodells vergeht.

SystemC-Simulationskern Der Simulationskern in SystemC arbeitet ereignisgesteuert und ist in drei Phasen unterteilt. In der ersten Phase werden die notwendigen Datenstrukturen initialisiert und die Verbindungen zwischen den einzelnen Elementen eingerichtet. Diese werden in der letzten Phase wieder bereinigt. Die eigentliche Simulation wird in der zweiten Phase durchgeführt und startet mit dem Aufruf der Funktion `sc_start()`, der als Parameter die Simulationsdauer übergeben werden kann. Zu diesem Zeitpunkt werden alle dem Simulationskern angemeldeten Prozesse gestartet. Deren Ausführung besteht wiederum aus zwei Teilen. Im ersten Teil wird die eigentliche Arbeit der Prozesse ausgeführt, im zweiten werden die Ausgabedaten bekanntgegeben oder aktualisiert. Dieser kombinierte Vorgang beschreibt einen *Delta-Zyklus*. Die Simulationszeit schreitet erst voran, wenn sich kein Prozess mehr im aktuellen Delta-Zyklus befindet, im Speziellen auch diejenigen nicht mehr, die erst durch die aktualisierten Ausgabewerte eines anderen Prozesses gestartet wurden.

[8] First In First Out

Intern werden dabei zwei Listen erstellt. In der einen werden die aktuell arbeitenden Prozesse gehalten, in der anderen die Prozesse, die auf ein Ereignis warten. Tritt so ein Ereignis ein, wird dieser Prozess in die Liste der arbeitenden Prozesse eingetragen und nach Beendigung der Arbeit wieder zurück. Die Simulationszeit schreitet solange nicht voran, bis sich keine Prozesse mehr in der Liste der arbeitenden Prozesse befinden.

Prozesse *Prozesse* sind in SystemC die eigentlich ausführenden Einheiten, also die Erbringer der Funktionalität eines Moduls, und bieten somit den Mechanismus, das nebenläufige Verhalten eines Systems zu simulieren. Prozesse müssen im Konstruktor des jeweiligen Moduls deklariert werden und registrieren sich damit beim Simulationskern, der die Prozesse bei Beginn der Simulation startet. Die Sensitivitätsliste eines Prozesses legt dabei fest, durch welche Signale der Prozess aktiviert werden kann. Es existieren drei verschiedene Arten von Prozessen:

Methode Eine Methode wird dann ausgeführt, wenn ein Ereignis eintritt, das in der zugehörigen Sensitivitätsliste spezifiziert wurde. Wird eine Methode ausgeführt, läuft diese von Beginn bis Ende ohne Unterbrechung. Eine Methode wird durch die Verwendung des Makros `SC_METHOD` im Konstruktor deklariert.

Thread Im Gegensatz zu einer Methode kann ein Thread während dessen Ausführung unterbrochen und in einen Wartezustand versetzt werden. Dies geschieht durch Aufruf der Funktion `wait()`. Je nachdem, wie die Wartefunktion aufgerufen wird, kann der Thread dann fortgesetzt werden, wenn die als Parameter übergebene Zeit abgelaufen ist oder wenn das als Parameter angegebene Ereignis eintritt, auf das gewartet wird. Da Threads in der Regel ständig laufen sollen, werden sie meist innerhalb einer Endlosschleife implementiert. Threads werden durch Verwendung des Makros `SC_THREAD` im Konstruktor des zugehörigen Moduls deklariert.

Clocked Thread Diese Art von Thread wurde entwickelt, um die Implementierung von voll-synchroner Logik zu erleichtern, denn hierbei wird die Aktivierung dieses Threads durch einen kontinuierlichen Zeitgeber hervorgerufen. Dynamische Sensitivität kann durch die Funktion `wait_until()` erreicht werden. Clocked Threads werden durch das Makro `SC_CTHREAD` im Konstruktor des Moduls deklariert.

Events *Events* bzw. *Ereignisse* sind ein Schlüsselelement einer ereignisbasierten Simulation, wie sie vom Simulationskern in SystemC durchgeführt wird. Die Hauptaufgabe von Events ist die Synchronisierung von Prozessen und Prozessabläufen. Events werden durch Instanziierung der Klasse `sc_event` deklariert. Ein Event ist ein Ereignis, das während einer Simulation auftritt und besitzt weder eine Länge noch einen Wert. Auf die Notifizierung eines Events reagieren nur diejenigen Prozesse, die sensitiv auf diesen Event sind. Um auf einen Event zu warten, wird die `wait()`-Funktion mit dem Event als Parameter benutzt. Will ein Prozess einen Event benachrichtigen bzw. *notifizieren*, auf den andere Prozesse potentiell warten, ruft er die Funktion `notify()` für dieses

Event auf. Events können nur dann bemerkt werden, wenn explizit und zu dieser Zeit darauf gewartet wird. Ist ein Prozess gerade beschäftigt, ignoriert er das Auslösen eines Events.

Sensitivität Events repräsentieren Ereignisse, die während einer Simulation auftreten können. Allerdings werden nicht zwingend alle Prozesse über das Eintreten eines solchen Ereignisses unterrichtet. Die Angabe, welcher Prozess auf welchen Event reagiert, wird in der sogenannten *Sensitivität* festgelegt. Dabei gibt es zwei unterschiedliche Ansätze:

Statische Sensitivität Bei einer statischen Sensitivität werden vor Beginn der Simulation diejenigen Signale spezifiziert, auf die der jeweilige Prozess sensitiv reagieren soll. Dazu wird nach der Registrierung des Prozesses im Simulationskern eine Liste der Signale mittels `sensitive<<[Signal 1]<<[Signal 2]<<...` angelegt. Diese kann nach dem Start der Simulation nicht mehr verändert werden, sodass die Sensitivität eines Prozesses statisch erhalten bleibt.

Dynamische Sensitivität Bei Verwendung von dynamischer Sensitivität kann die Sensitivitätsliste während der Laufzeit überschrieben werden. Dies ist dann der Fall, wenn durch Aufrufen der Funktion `wait([Event-Name])`auf Events gewartet wird und die Ausführung erst nach der Notifizierung dieses Events fortgesetzt wird. Events können innerhalb einer `wait()`-Anweisung durch logische Operatoren verknüpft werden.

2.3.1.4 Transaction-Level Modeling (TLM)

Um die Komplexität umfassender Modelle beherrschbar zu machen, können Simulationen auf unterschiedlichen Abstraktionsebenen durchgeführt werden. Das *Transaction-Level Modeling* [34] ist eine Entwurfsmethodik, die es erlaubt, von komplizierten Kommunikationsmechanismen zu abstrahieren. Durch TLM werden einzelne Kommunikationsschritte zu Transaktionen zusammengefasst, was die gesamte Simulationsdauer um viele Faktoren verkürzen kann. Dadurch kann zusätzlich das interne Verhalten eines kommunizierenden Elements von der Kommunikation an sich getrennt werden, um beide Teile unabhängig voneinander wiederverwenden oder optimieren zu können. Die Instrumente für die Modellierung auf diesem hohen Abstraktionslevel sind die schon früher in diesem Abschnitt erwähnten Interfaces, Kanäle und Events. Je nach Grad der Abstraktion können TLM-Modelle *timed*, *untimed* oder *cycle-accurate* sein, was sich direkt auf die Simulationsdauer auswirkt. Der TLM-2.0-Standard stellt hierfür eine Template-basierte Programmierschnittstelle zur Verfügung, die sowohl unidirektionale als auch bidirektionale Kommunikationsmechanismen anbietet und dabei die eigentlich zu übertragenden Daten in einer generischen Struktur kapselt. Auf diese Kommunikationsmechanismen kann über standardisierte Schnittstellen blockierend sowie nicht-blockierend zugegriffen werden. Durch die Standardisierung wird zudem die Interoperabilität zwischen modellierten Systemen bzw. Teilsystemen garantiert.

2.3.2 Instrumentierung von Simulationsmodellen

Simulationsmodelle auf unterschiedlichen Abstraktionsebenen modelliert werden. Dies gilt in gleichem Maße für die *Annotation nicht-funktionaler Eigenschaften (NFP)* innerhalb der Simulationsmodelle. Diese reichen von der Annotation von Taktzyklen in Hardware-Modellen, über die Annotation von Ausführungszeiten einzelner Instruktionen eingebetteter Software unter Berücksichtigung der ausführenden Hardware-Architektur, bis hin zu abstrakten Annotationen an Methoden oder Transaktionen in Modellen auf Systemebene. Vereint über alle Abstraktionsebenen hinweg ist eine manuelle Instrumentierung des Simulations-Codes durch nicht-funktionale Eigenschaften ein erheblicher Mehraufwand. In letzter Zeit haben sich jedoch Verfahren zur automatisierten Instrumentierung etabliert.

Erste Arbeiten in dieser Richtung wurden in [112] erforscht und anschließend in [115] erweitert. Dadurch werden komplexe Optimierungen des Quellcodes durch moderne Übersetzerwerkzeuge (engl.: *Compiler*), wie z.B. das Ausrollen von Schleifen oder das Auslagern von Schleifeninvarianten, in mehreren Analyseschritten eingebunden. Zunächst wird der Quellcode für die jeweilige *Zielarchitektur* übersetzt. Für jeden Basisblock des übersetzten Binärprogramms, also für jeden atomaren und somit verzweigungsfreien Block innerhalb des Kontrollflusses, wird eine Analyse des nicht-funktionalen Verhaltens durchgeführt, die Effekte der Mikroarchitektur auf der Zielarchitektur, wie Fließbandverarbeitung (engl.: *Pipelining*) und statische Sprungvorhersagen (engl.: *Branch Prediction*), einbezieht. Als Ergebnis entsteht ein NFP-annotierter Kontrollflussgraph des ausführbaren Binärprogramms mit annotierten Kanten, die die *Ausführungszeit* und die Anzahl der jeweils ausgeführten *Instruktionen* beinhalten. Basierend auf diesem binären Kontrollflussgraph wird der Kontrollfluss auf der Zielarchitektur analysiert, um ein Modell zur *Pfadsimulation* generieren zu können, der das Zielarchitektur-spezifische Verhalten des Programms modelliert. Durch die gleichzeitige Übersetzung des instrumentierten Quellcodes und des Codes zur Pfadsimulation für die *Simulationsplattform* entsteht ein Simulationsmodell des Programms, welches das nicht-funktionale Verhalten des Binärprogramms auf der Zielarchitektur modelliert. Anhand der Markierungen, z.B. Funktionsaufrufe, die während des Instrumentierungsprozesses in den Simulations-Code eingefügt werden, wird der Kontrollfluss des Binärprogramms durch die dynamische Ausführung der Pfadsimulation abgeschätzt. Da diese *Rekonstruktion* des Kontrollflusses auf Binärebene parallel zur Funktionalität des originalen Simulationsmodells auf der Simulationsplattform ausgeführt wird, werden die annotierten nicht-funktionalen Eigenschaften dynamisch ausgewählt, was unter anderem die Berücksichtigung von Datenabhängigkeiten ermöglicht. Nicht zuletzt dadurch wird die Abschätzung des nicht-funktionalen Verhaltens des Simulationsmodells sehr genau. Der entscheidende Vorteil des Instrumentierungsansatzes ist jedoch, dass die Simulation das nicht-funktionale Verhalten der Zielarchitektur widerspiegelt, aber auf der Simulationsplattform ausgeführt wird. Dadurch erhöht sich die Simulationsgeschwindigkeit um mehrere Größenordnungen im Vergleich zur Ausführung desselben Programms auf einem Instruktionssatz-Simulator. Entsprechende Simulationsergebnisse können aus [115] entnommen werden. Weiterhin ist das

Konzept der dynamischen Pfadsimulation unabhängig von möglicherweise falschen Hilfsinformationen des Übersetzerwerkzeugs, da nur gültige Pfade simuliert werden. Zusätzlich können Änderungen und Optimierungen der Programmstruktur, die durch das Übersetzerwerkzeug vorgenommen werden, mit hoher Genauigkeit berücksichtigt und simuliert werden, was allein durch statische Annotationen nicht-funktionaler Eigenschaften kaum möglich wäre. Während der Ausführung der Simulation können sogar Speicherzugriffe inklusive deren Effekte, wie erhöhte Zugriffszyklen bei Fehlzugriffen in Caches, modelliert werden, ohne dass Verbindungen zwischen Variablen im Simulations-Code und Speicherzugriffen im übersetzten Maschinen-Code hergestellt werden müssen [116].

2.3.3 MATLAB/Simulink

MATLAB ist eine Entwicklungsumgebung inklusive eigener Programmiersprache, die ursprünglich einem mathematischen Werkzeug zur Durchführung numerischer Berechnungen anhand von Matrizen entstammt. *Simulink* ist eine Erweiterung für MATLAB und wird hauptsächlich zur modellbasierten Simulationen verwendet. Nach [56] liegen die Vorteile von Simulink in der „schnellen, kostengünstigen Entwicklung dynamischer System inklusive Regelungstechnik, Signalverarbeitung und Kommunikation“. Die Modellierung erfolgt durch Simulink-Blöcke, die Gleichungen bzw. Differentialgleichungen bis hin zu komplexen Regelungssysteme enthalten können. Jeder Block besitzt Eingänge für die benötigten Daten und Signale, sowie Blockausgänge für die Ergebnisausgaben. Die periodische Abtastzeit kann entweder in jedem Block parametrisiert werden oder von verbundenen Blöcken geerbt werden, sodass sie mit diesen übereinstimmt. Der Austausch von Informationen kann visuell über die Verbindung durch Linien erfolgen, die eine Kommunikation für einzelne Daten oder einen Datenbus repräsentieren, was das natürliche Verständnis von Simulink-Modellen unterstützt.

Die Funktionalität von Simulink wird durch die Verwendung vorgefertigter und zum Teil domänenspezifischer Simulink-Blöcke erweitert. Eine domänenspezifische Menge an Blöcke wird auch als *Toolbox* bezeichnet. Die Integration von *S-Functions*, die in Abschnitt 2.3.3.1 detaillierter beschrieben werden, ermöglicht die Ausführung von C-Code innerhalb eines Simulink-Modells, sowie MATLAB-Function-Blocks das Ausführen von MATLAB-Code. Simulink unterstützt aber auch ein reziprokes Vorgehen, also die Generierung von C/C++-Quelltext aus einem Simulink-Modell heraus. Dies ermöglicht das Einsetzen von Simulink in Testszenarien für Software-in-the-Loop-Ansätze (SiL) und Hardware-in-the-Loop-Ansätze (HiL).

2.3.3.1 Simulink S-Function

Das Prinzip der *S-Function* erweitert Simulink um die Verwendung von C/C++- oder Fortran-Quelltext [56]. Bevor diese in das übergeordnete Simulink-Modell eingebunden werden können, werden sie für die jeweilige Architektur kompiliert. Für die Interaktion zwischen S-Function und Simulink existiert eine spezielle Programmierschnittstelle

(engl.: *Application Programming Interface (API)*), über welche die Ein-und Ausgänge der S-Function belegt und angesprochen werden. Der Ablauf einer S-Function besteht im Wesentlichen aus drei Teilen, die jeweils durch Ausführung spezifischer Methoden bestimmt werden:

(1) Initialisierungsphase, in der die Eingänge bzw. Ausgänge und deren Typ, sowie die Abtastzeit festgelegt werden. Die Abtastzeit kann vom vorangegangenen Block geerbt werden.

(2) Aktualisierung der Ausgänge pro Simulationsschritt anhand der durchgeführten Berechnungen

(3) Terminierungsphase, in der beliebige Operationen ausgeführt werden dürfen

2.4 Berechnungsmodelle und anwendbare Algebren

Insbesondere Anwendungen im Echtzeit- und Multimediabereich fordern ein robustes System mit garantierter Leistung. Deshalb muss jede Anwendung ein vorhersagbares Zeitverhalten haben. Mithilfe von Berechnungsmodellen können Zeitvorgaben und gegenseitige Abhängigkeiten zwischen verschiedenen Bestandteilen der Anwendungen modelliert werden. Außerdem existieren Analysetechniken auf diesen Berechnungsmodellen, mit denen zum Beispiel der Durchsatz oder der benötigte Speicherplatz berechnet werden kann. Dieser Abschnitt soll eine Einführung in Berechnungsmodelle geben und damit die Grundlage für spätere modellbasierte Analysen legen. Teile der hier enthaltenen Definitionen sind sinngemäß aus [124] entnommen.

2.4.1 Allgemeine Berechnungsmodelle

Allgemeine Berechnungsgraphen wurden erstmals von Karp und Miller beschrieben [64]. Ein Berechnungsgraph ist ein gerichteter Graph $G_B = (V_B, E_B)$ bestehend aus einer Menge von Knoten $v_1, ..., v_n \in V_B$, denen jeweils eine Funktion $f_1, ..., f_n$ zugeordnet ist. Eine Kante e_k aus der Menge der Kanten $e_1, ..., e_m \in E_B$ bezeichnet einen Datenfluss von einem festgelegten Knoten v_i zu einem festgelegten Knoten v_j, der nach dem FIFO[9]-Prinzip operiert. Jeder Kante sind dabei 4 Werte zugeordnet:

- A_k: Anzahl der Daten bzw. *Token*, die zu Beginn in der FIFO vorhanden sind,
- U_k: Anzahl der Token, die von n_i produziert werden,
- W_k: Anzahl der Token, die von n_j konsumiert werden,
- T_k: Anzahl der Token, die an e_k vorhanden sein müssen, bevor n_j auslöst.

Für diese Werte müssen immer folgende drei Eigenschaften gelten:

(1) $T_k \geq W_k \geq 0$,

[9]First In First Out

(2) $A_k \geq 0$,

(3) $U_k \geq 0$.

Ein Knoten v_j (bzw. dessen Funktion f_j) löst aus, wenn die Anzahl an Token, die an e_k anliegen, größer oder gleich T_k ist. Sobald v_j auslöst, werden W_k Token verbraucht und an jedem Ausgang e_l werden U_l Token erzeugt. Zu beachten ist hierbei, dass damit keine datenabhängigen Bedingungen modelliert werden können.

Ein *Kahn-Prozess-Netzwerk (KPN)* ist ein verteiltes Berechnungsmodell [63], das starke Ähnlichkeiten zu den allgemeinen Berechnungsgraphen ausweist. Es ist eines der gängigsten Modelle, um das Verhalten von Datenströmen zu beschreiben. Im Wesentlichen handelt es sich dabei um sequentiell ablaufende Prozesse, die über FIFO-Datenstrukturen unendlich großer Kapazität kommunizieren. Ein Schreiben in einen Kanal ist nicht-blockierend, das Lesen hingegen ist blockierend. Ein Lesezugriff entnimmt jeweils ein Element, wobei alle Zugriffe auf die Schlange atomar sind. Das Ergebnis des Netzwerks hängt von der Gestaltung der Prozesse, den Verknüpfungen dieser Prozesse und der Eingabe ab. Die Frage, ob ein Kahn-Prozess-Netzwerk terminiert ist unentscheidbar, da sich Kahn-Prozess-Netzwerke auf eine Menge von verbundenen *Turing-Maschinen* abbilden lassen [95]. Damit entspricht die Frage der Terminierung dem Halteproblem.

2.4.2 Datenflussorientierte Berechnungsmodelle

Datenflussgraphen stellen eine Erweiterung der allgemeinen Berechnungsgraphen dar. Ein Datenflussgraph G_D ist ein gerichteter Graph $G_D = (V_D, E_D)$. Die Knoten heißen *Aktoren*, die Kanten stellen die Datenflüsse zwischen den Aktoren dar. Der Zeitpunkt, zu denen ein Aktor auslöst, wird durch Regeln eindeutig festgelegt. Diese Regeln spezifizieren demnach, welche Bedingungen erfüllt sein müssen, um die Aktion auszulösen. Einem Aktor können eine oder mehrere solcher Regeln zugeordnet sein. Sobald eine Regel erfüllt ist, ist der Akteur in der Lage auszulösen. Die Regeln können wie folgt definiert werden, wobei sich die Darstellung an [76] anlehnt. Jeder Aktor mit $p \geq 1$ Eingängen kann n Auslöse-Regeln haben:

$$\mathfrak{R} = \{\bar{R}_1, \bar{R}_2, ..., \bar{R}_n\}.$$

Jede dieser Regeln $\bar{R}_i$ besteht wiederum aus einer Menge von endlichen Folgen $R_{i,j}$, die sämtliche p Eingänge abdecken. Ein Aktor mit keinem Eingang ($p = 0$) ist immer aktiv.

$$\bar{R}_i = \{R_{i,1}, R_{i,2}, ..., R_{i,p}\}.$$

Die Regel $\bar{R}_i$ löst nun jeweils dann aus, wenn alle $R_{i,j}$ erfüllt sind, d. h. die endlichen Folgen $R_{i,j}$ – die leere Folge $R_{i,j} = \perp$ ist ebenfalls zugelassen – ein Präfix der noch nicht konsumierten Eingabedaten am Eingang j sind. Eine leere Folge $\perp$ bedeutet nicht, dass keine Eingabe am Eingang j anliegen muss, sondern dass jede Eingabe eine gültige Eingabe ist. Weiterhin existiert ein Platzhalter $*$ für ein beliebiges Token an der Eingabe. Dieser Platzhalter ist ein Präfix jeder Folge und nur die leere Folge ist ein Präfix des Platzhalters. Unter Zuhilfenahme des Präfix-Operators $\sqsubseteq$ gilt somit:

(1) $\perp \sqsubseteq E$, für jede Eingabefolge E,

(2) $[*] \sqsubseteq E$, wobei E eine nichtleere Eingabefolge ist,

(3) $\perp \sqsubseteq [*]$.

Das Auslösen kann somit folgendermaßen beschrieben werden. Sei A_j für alle $j = 1, ..., p$ die Folge verfügbarer und noch nicht konsumierter Token am Eingang j. Dann ist die Auslöse-Regel $\bar{R}_i$ erfüllt, wenn gilt:

$$R_{i,j} \sqsubseteq A_j \quad \forall j = 1, ..., p.$$

2.4.3 Synchrone Datenflussmodelle

Ein Spezialisierung der in Abschnitt 2.4.2 eingeführten Datenflussmodelle bilden die *Synchronen Datenflussmodelle (SDF)* bzw. *Synchronen Datenflussgraphen*, die erstmals von Lee und Messerschmitt vorgestellt wurden [75]. Die Anzahl der konsumierten und produzierten Daten ist bei diesem Modell immer konstant, d.h. die Anzahl der konsumierten Elemente entspricht genau der Anzahl der Elemente, die zum Auslösen notwendig sind. Im Bereich der Synchronen Datenflussgraphen hat sich eine spezielle Syntax und Notation durchgesetzt, die im restlichen Teil dieser Arbeit verwendet werden soll, die sich jedoch auch teilweise mit den Begriffen der allgemeineren Berechnungsmodelle decken.

Ein SDF-Graph besteht demnach aus *Aktoren*, die in der Regel Funktionen oder *Tasks* repräsentieren. Diese sind über Ports und Kanten verbunden, um Datenabhängigkeiten darzustellen. Die Ausführung eines Aktors wird als „feuern“ bezeichnet [118]. Dabei wird eine gewisse Menge an Daten (*Token*) von allen Eingängen entsprechend den annotierten *Raten* entfernt und auf die Ausgänge gelegt. Auch für die Ausgänge sind Raten angegeben, wobei diese nicht gleich den Eingangsraten sein müssen. Der *Repetitionsvektor* beschreibt, wie oft ein Aktor im Bezug auf alle anderen Aktoren des Systems feuern muss, damit sich die Verteilung der Token über alle Kanten hinweg nicht ändert. Der Repetitionsvektor des SDF-Graphen in Abbildung 2.3 entspricht

$$(a_1, a_2, a_3) \rightarrow (3, 2, 1)$$

Dieser Vektor ist nicht trivial, da alle Werte ungleich Null sind. Damit gilt der SDF-Graph als *konsistent* und es existiert ein gültiger Ablaufplan, der den Graph wieder in seinen Ausgangszustand überführen kann. Ein nicht-konsistenter Graph würde zur Ausführung auf einem realen System unendlich viel Speicherplatz benötigen. Wenn in einem SDF-Graphen alle Raten gleich Eins sind, so handelt es sich um einen *homogenen* Graphen. Der Repetitionsvektor ist dann für alle Aktoren ebenfalls Eins. Jeder konsistente SDF-Graph kann in solch einen homogenen Graphen konvertiert werden. Bei der Transformation in einen homogenen SDF-Graph kann die Anzahl der Aktoren allerdings exponentiell steigen.

In der allgemeinen SDF-Semantik ist das Feuern eines Aktors ein atomarer Vorgang, d.h. dass dabei keine Zeit vergeht. Damit soll also primär die Reihenfolge der möglichen

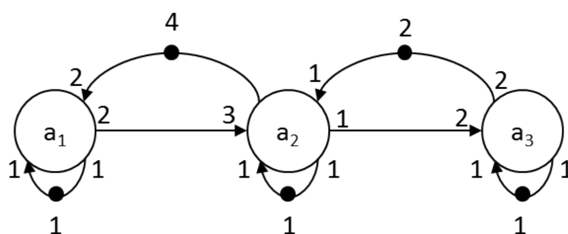


Abbildung 2.3: Beispiel eines SDF-Graphen

Ausführungen einzelner Aktoren festgelegt werden. Da aber ein Aktor auch einen Task repräsentieren kann, kann diesem sehr wohl eine Ausführungszeit zugeordnet werden. Diese wird an den Knoten annotiert – im Normalfall handelt sich hierbei um die *Worst-Case Execution Time (WCET)* des entsprechenden Tasks. Das dynamische Verhalten eines sogenannten *Timed SDF-Graphen* wird über Transitionen beschrieben. Die Zeit schreitet nur für Aktoren voran, die gerade ausgeführt werden. Dieser Zustand liegt zwischen Start und Ende des Feuerns. Dies führt zu einer Art der Ausführung, die *Self-Timed Execution* genannt wird, was bedeutet, dass ein Aktor sofort feuert, sobald alle nötigen Eingangs-Token vorhanden sind. Dieses Verhalten führt zu einer Maximierung des Durchsatzes. Wenn jeder Aktor so oft gefeuert hat, wie es ein minimaler Repetitionsvektor vorgibt, dann hat der SDF-Graph einen Zyklus durchlaufen und befindet sich wieder im Ausgangszustand. Dieser abgeschlossene und periodische Zyklus wird in der SDF-Syntax als eine *Iteration* bezeichnet.

SDF-Modelle mit Zeitverhalten erlauben neben der Ableitung eines Ablaufplans mithilfe des Repetitionsvektors auch die Berechnung und Analyse von Performanzeigenschaften [14] [15]. In [37] wird ein Ansatz zur Analyse des mindestens erreichbaren Durchsatzes eines SDF-Modells vorgestellt.

2.4.4 Max-Plus-Algebra

Die oftmals in verteilten Echtzeitsystemen auftretende Frage der Synchronisierung mehrerer zeitgesteuerter Ereignisse kann in der Regel auf das mathematische Modell einer Max-Plus-Algebra abgebildet werden, die daraus folgend eine analytische Herangehensweise an das Synchronisationsproblem erlaubt. Eine Max-Plus-Algebra ist eine Halbring-Algebra H, in der die Additionsoperation $\oplus_H$ durch die Bildung der Maximum-Funktion und die Multiplikationsoperation $\otimes_H$ durch eine Addition ersetzt wird [10]. Weiterhin existieren mit $-\infty$ für $\oplus$ und 0 für $\otimes$ jeweils neutrale Elemente zu beiden Operationen. Anwendung findet die Max-Plus-Algebra damit unter anderem in der Automatentheorie und der Behandlung zeitbehafteter Petri-Netze [97].

Der Nachweis der algebraischen Eigenschaften einer Max-Plus-Algebra erfolgt über die Erfüllung folgender Axiome eines Halbrings H mit $x, y, z \in H$:

Assoziativität

$$max(max(x, y), z) = max(x, max(y, z))$$
$$(x + y) + z = x + (y + z)$$

Kommutativität

$$max(x, y) = max(y, x)$$
$$x + y = y + x$$

Distributivität

$$max(x, y) + z = max((x + z), (y + z))$$

Neutrale Elemente

$$max(n_{\oplus_H}, x) = max(x, -\infty) = x$$
$$n_{\otimes_H} + x = x + 0 = x$$

Idempotenz

$$max(x, x) = x$$

2.5 Mehrkern-Architekturen

Mehrkern-Architekturen werden zumeist durch die anglizistischen Begriffe *Multicore-Architekturen* und *Manycore-Architekturen* differenziert. Prinzipiell unterscheiden sich diese in der Anzahl der Berechnungseinheiten bzw. Kerne, die genaue Trennung liegt aber an der jeweils verwendeten Definition und ist deswegen meist fließend. Im Folgenden sollen die hinsichtlich der Optimierungsziele entscheidenden Eigenschaften von Mehrkern-Architekturen erörtert werden. Dabei soll zwischen Architekturen mit identischen Kernen, also *homogene Architekturen*, und *heterogene Architekturen* unterschieden werden. Letztere verfügen über verschiedene Typen von Berechnungskernen, die je nach Anwendungsfall zur Abarbeitung einer Aufgabe eingesetzt werden können. Des Weiteren wird Parallelismus als eine Voraussetzung für den effizienten Einsatz von Mehrkern-Architekturen erläutert.

2.5.1 Homogene Architekturen

Die ursprüngliche Idee zum Einsatz von Mehrkern-Architekturen war die Verteilung der durchzuführenden Aufgaben auf mehrere Berechnungseinheiten. Eine gleichmäßige Verteilung über alle zur Verfügung stehenden Kerne wird als *Workload-Balancing* bezeichnet. Da die Auslastung jedes einzelnen Kerns möglichst gering gehalten wird, wird der Durchsatz des Gesamtsystems maximiert, was ein wichtiger Maßstab der Performanzeigenschaften darstellt. Gerade deshalb ist die gleichmäßige Verteilung der Aufgaben oftmals ein Ziel der Optimierung einer Ablaufstrategie auf Mehrkern-Architekturen. Um eine freie Verteilung ohne Anpassung der zu bearbeitenden Aufgaben und dadurch weniger komplex durchführen zu können, müssen die zur Verfügung stehenden

Kerne vom selben Typ, d.h. identisch sein. Solche Architekturen werden als *homogen* bezeichnet.

Üblicherweise können folgende Varianten und deren Mischform in homogenen Architekturen unterscheiden werden [66]:

Hierarchisch Die Kerne werden als eine Art Baumstruktur innerhalb der Speicherhierarchie (z.B. Caches) angeordnet, somit teilen sich mehrere Kerne die verfügbaren Speicherelemente. Normalerweise steigt die absolute Größe der Speicherelemente mit jeder Ebene. Alle Kernen können über die Speicherhierarchie hinweg auf den Hauptspeicher zugreifen.

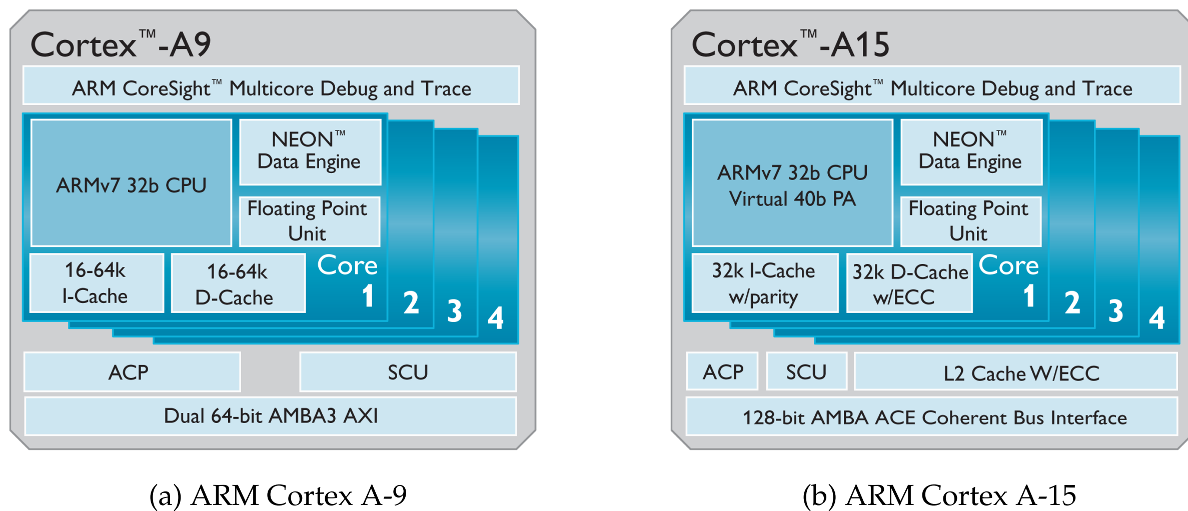
Pipeline Das Pipeline-Design wird insbesondere bei Grafikprozessoren (GPU) und Netzwerkprozessoren eingesetzt. Die jeweiligen Daten werden stufenweise von mehreren Prozessorkernen bearbeitet, bis sie schließlich nach Beendigung der Gesamtaufgabe im Speicher abgelegt werden.

Feld Bei einem Feld sind die Kerne sowie ihre lokalen Speicherelemente bzw. Caches über ein Verbindungsnetzwerk mit anderen Prozessorkernen des Chips verbunden. Die gesamte Kommunikation wird über das Verbindungsnetzwerk realisiert.

In den meisten Mobilgeräten werden nach heutigem Stand hierarchische Mehrkernprozessor-Architekturen verwendet. Aktuelle Beispiele dafür sind die ARM®-basierten System-on-Chip-Architekturen *Cortex™ A-9* bzw. *Cortex™ A-15*, die als Lizenzierung an verschiedene Hersteller verkauft werden. Deren struktureller Aufbau ist in Abbildung 2.4 dargestellt. Beide Architekturen verfügen über 1–4 Berechnungskerne, die über eine gemeinsame Verbindungsstruktur miteinander kommunizieren können. Größter Unterschied ist dabei der verwendete Bus zur Kommunikation und der vorhandene dedizierte Level-2-Cache als schneller Zwischenspeicher. Jeder Kern bzw. jede CPU verfügt über eine eigene Einheit zur Verarbeitung von Fließkommazahlen. Weitere aktuelle Beispiele sind die Snapdragon™-Reihe des Herstellers Qualcomm® oder Intel® Atom™-Architekturen aus der Medfield™-Serie.

Im Gegensatz zu einer hierarchischen Anordnung der Kerne werden Feld-Architekturen bevorzugt eingesetzt, wenn die Anzahl der Berechnungskerne einen gewissen Schwellwert überschritten hat. Durch die Kommunikation über ein Verbindungsnetzwerk anstatt über ein geteiltes Kommunikationsmedium wird die Gefahr eines gegenseitigen Blockierens und der damit verbundenen Verzögerung von Kommunikationsabsichten gesenkt. Jeder Kern verfügt deswegen über eine Schnittstelle zum Kommunikationsnetzwerk, das die Kerne in einer zumeist regelmäßigen Struktur untereinander verbindet. Da diese Art der Kommunikation der Verbindung von normalen Rechnersystemen über Netzwerke ähnelt, wird diese Technologie auch *Network-on-Chip (NoC)* genannt. Durch die Einführung eines Verbindungsnetzwerks auf Chip-Ebene können beim Entwurf die beiden Bestandteile Datenverarbeitung auf den Kernen und Kommunikation über das Netzwerk weitgehend getrennt voneinander betrachtet und optimiert werden [113].

Die *Tile64™*-Architektur des Herstellers Tilera® umfasst insgesamt 64 identische Berechnungskerne, die über ein gitterförmiges Kommunikationsnetzwerk verbunden



(a) ARM Cortex A-9

(b) ARM Cortex A-15

Abbildung 2.4: ARM Cortex™ A-9/15 Mehrkern-Architekturen (Bildquelle: ARM)

sind. Abbildung 2.5 zeigt den kompletten Chip inklusive der Schnittstellen zur Ein- bzw. Ausgabe und der Speicheranbindung. An einem Kern hervorgehoben ist die Anbindung eines Prozessors an das Verbindungsnetzwerk und den Cache. Für unterschiedliche Arten der Kommunikation, wie z.B. zwischen zwei Kernen oder zwischen einem Kern und dem externen geteilten Speicher, stehen mehrere separate Kanäle zur Verfügung. Dadurch soll die Latenz bei hohem Kommunikationsaufkommen nicht beeinträchtigt werden. Durch die große Anzahl zur Verfügung stehender Berechnungskerne spielt die Leistungsfähigkeit jedes einzelnen Kerns eine deutlich geringere Rolle als beispielsweise in eben genannten Architekturen von ARM oder Intel.

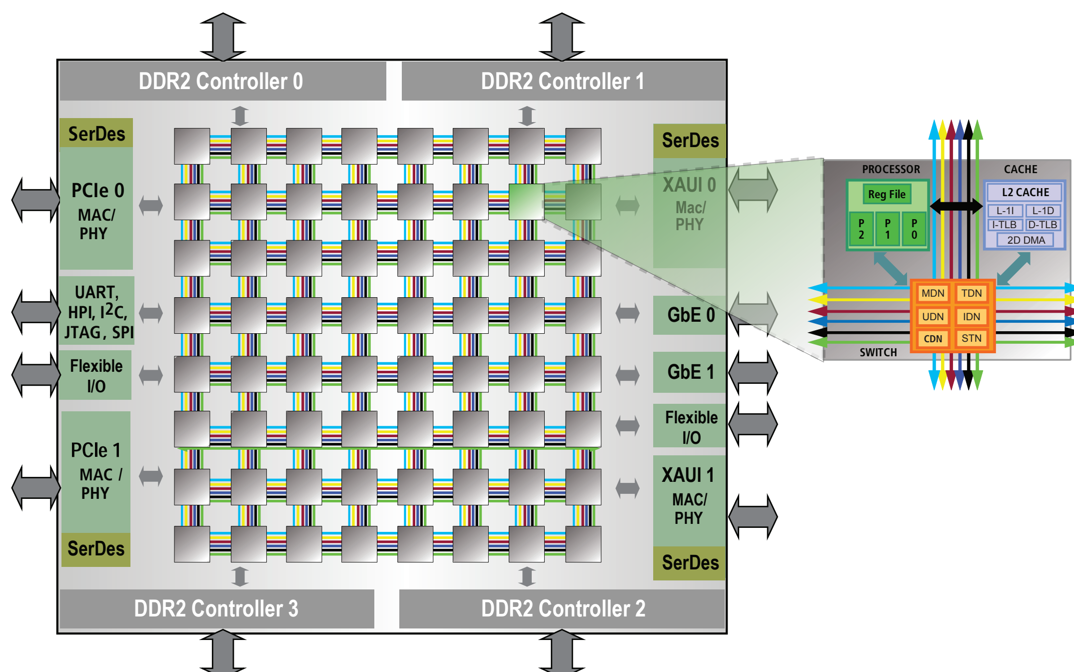


Abbildung 2.5: Tilera®Mehrkern-Architektur (Bildquelle: Tilera)

Eine weitere Feld-Architektur mit einer Vielzahl an Kernen stellt der von Intel zu Forschungszwecken bereitgestellte *Single-Chip-Cloud*™-Computer (SCC) dar. Diese experimentelle Architektur ist relativ ähnlich zur Tile64-Architektur, verfügt jedoch über insgesamt 48 Berechnungskerne. Alle Kerne sind vom Typ Pentium P54C und verfügen über FPU[10], MMU[11] und L1/L2-Caches[12], wobei immer zwei Kerne zu einer Einheit (engl.: *Tile*) zusammengefasst sind. Diese Tiles sind in der Anordnung 6×4 durch ein gitterförmiges Kommunikationsnetzwerk miteinander verbunden, das Daten als Nachrichten aus dem lokalen Speicher eines Tile zum lokalen Speicher des nächsten Tile überträgt. Abbildung 2.6 stellt die SCC-Architektur grafisch dar. Im Gegensatz zum Tile64 ist auf der SCC-Architektur ein durch den Benutzer steuerbares *Powermanagement* implementiert. Der komplette Chip ist in sechs Bereiche mit je acht Kerne eingeteilt, in denen die Versorgungsspannung angepasst werden kann. Ein solcher Bereich wird deswegen auch *Voltage Island* genannt. Die Verarbeitungsfrequenz kann hingegen in jedem Tile separat verändert werden, ein Tile bildet deshalb auch eine *Frequency Island*. Die Anwendung des Powermanagement des Intel SCC wird anhand einer Beispielapplikation noch detaillierter in Abschnitt 7.3 vorgestellt.

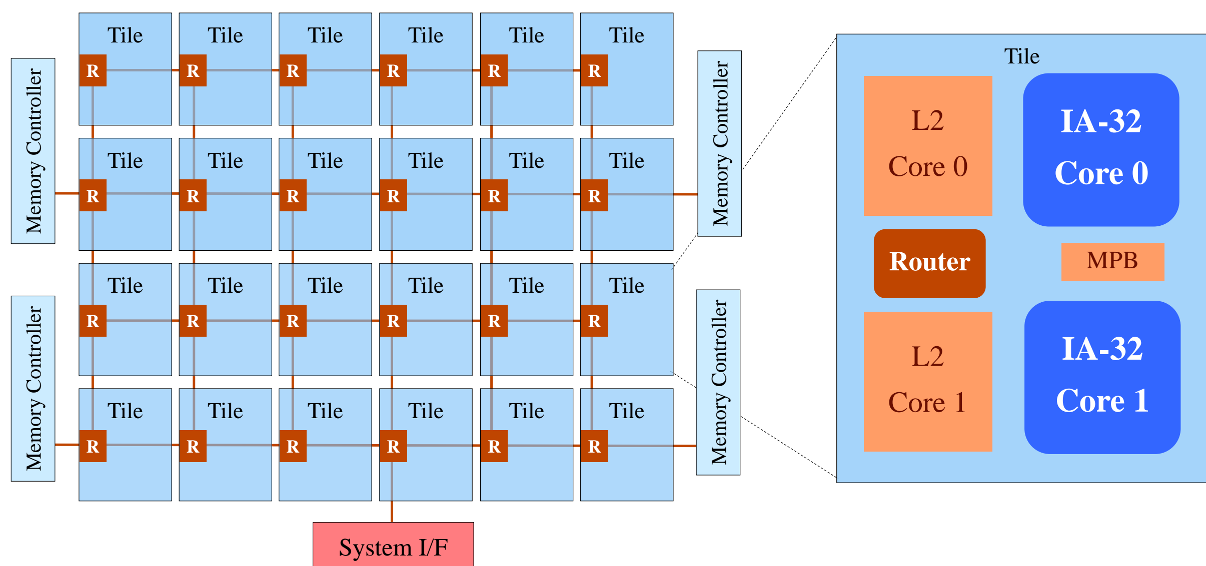


Abbildung 2.6: Intel Single-Chip-Cloud-Computer (Bildquelle: Intel)

2.5.2 Heterogene Architekturen

Die Notwendigkeit zur Steigerung der Energieeffizienz von Hardware/Software-Systemen führt auch bei den Paradigmen des Architekturentwurf zu entsprechenden Modifikationen. Hierbei wird die Tatsache ausgenutzt, dass unterschiedliche Typen von Berechnungskernen auch unterschiedliche Eigenschaften bezüglich der Leistungsfähigkeit und der Leistungsaufnahme besitzen. Im Gegensatz zu homogene Architekturen

[10]Floating Point Unit
[11]Memory Management Unit
[12]Cache-Speicher auf Level 1 bzw. Level 2

werden deshalb in *heterogenen Architekturen* verschiedene Typen von Kernen zu einer optimierten Architektur kombiniert. Auszuführende Aufgaben, die hohe Anforderungen an die Leistungsfähigkeit des Systems stellen, werden von dazu geeigneten Berechnungskernen ausgeführt, wohingegen Aufgaben mit geringen Leistungsanforderungen auf diejenigen Kerne ausgelagert werden können, die eine hohe Energieeffizienz aufweisen. Die Verteilung auf der Architektur kann so an die jeweilige Situation angepasst und damit ein bestimmtes Optimierungsziel verfolgt werden, da nicht-benötigte Kerne einzeln abgeschaltet werden können.

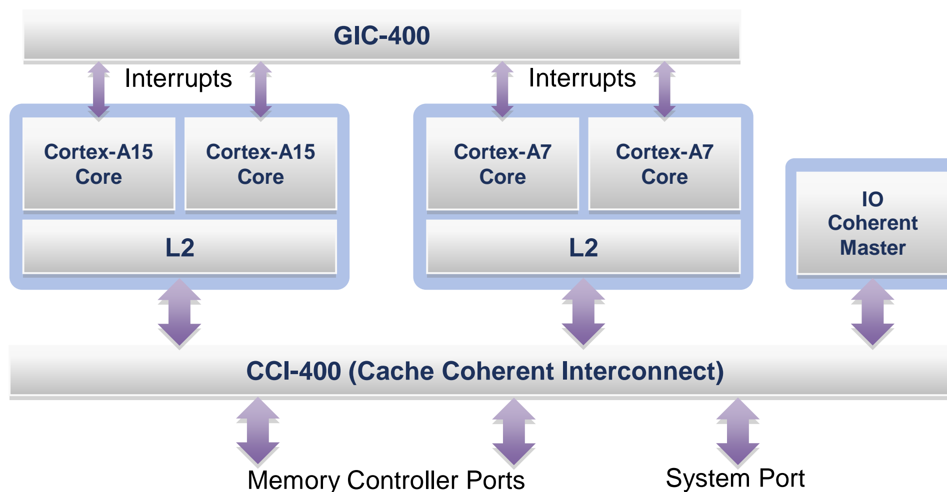


Abbildung 2.7: ARM big.LITTLE-Architektur (Bildquelle: [44])

Aktuelle Beispiele für die Kombination von leistungsoptimierten und Energie-optimierten Prozessoren in heterogenen Mehrkernsystemen sind die von NVIDIA® entwickelten *Tegra*™-*3*- bzw. *Tegra*™-*4*-Architektur und die *big.LITTLE*™-Architektur von ARM. In der letztgenannten Architektur werden bis zu vier Cortex A-15 mit bis zu vier Cortex A-7 kombiniert, was in Abbildung 2.7 ersichtlich ist. Die Kerne werden abhängig von den an das System gestellten Anforderungen eingesetzt. Die unterschiedlichen Eigenschaften der Berechnungskerne werden durch den in Tabelle 2.1 dargestellten Vergleich zwischen Kernen des Typs Cortex A-15 und Cortex A-7 anhand mehrerer Benchmarks verdeutlicht [44]. Auch in den *OMAP*™-*4*- und *OMAP*™-*5*-Plattformen von Texas Instruments® werden A-15-Kerne mit Cortex-Prozessoren der M-Serie kombiniert. In der Tegra-3- und Tegra-4-Architektur werden vier leistungsoptimierte Kerne von einem Energie-optimierten Kern unterstützt, der kontinuierlich anfallende Aufgaben ohne hohe Leistungsanforderungen bearbeitet, wohingegen die anderen vier Kerne nur für Multimedia-Anwendungen gebraucht werden.

Bei den bisherigen Beispielen sind nur diejenigen Architekturen erwähnt, die situationsbedingt Aufgaben auf allgemeine Prozessoren (engl.: *General Purpose Processors*) verteilen können. Neben diesen existieren heterogene Architekturen, die neben den allgemeinen Kernen noch spezialisierte Prozessoren besitzen. Dazu zählen typische *Grafikprozessoren* zur Bearbeitung von Vektoroperationen oder *digitale Signalprozessoren* (DSP) zur kontinuierlichen Bearbeitung von digitalen Signalen. Zu letzteren gehört die *Tricore*-Architektur von Infineon, die neben einem allgemeinen Prozessor über einen

Tabelle 2.1: Vergleich Performanz und Energieeffizienz von ARM Cortex A-15 und A-7

Benchmark	Performanz Cortex A-15 und A-7	Energieeffizienz Cortex A-7 und A-15
Dhrystone	1.9x	3.5x
FDCT	2.3x	3.8x
IMDCT	3.0x	3.0x
MemCopy L1	1.9x	2.3x
MemCopy L2	1.9x	3.4x

zusätzlichen Signalprozessor verfügt.

2.5.3 Parallelismus

In Mehrkern-Architekturen soll durch die Verteilung der auszuführenden Aufgaben auf mehrere Berechnungseinheiten die Leistungsfähigkeit des Gesamtsystems erhöht werden, ohne dass die Leistung eines einzelnen Kerns gesteigert werden muss. Grundvoraussetzung hierfür ist das Vorhandensein von *nebenläufigem* oder *parallelem* Verhalten innerhalb der Aufgaben. Nebenläufige Ereignisse zeichnen sich dadurch aus, dass mehrere dieser Ereignisse unabhängig voneinander bearbeitet werden können, also keine kausale Beziehung zwischen ihnen besteht. Parallele Ereignisse dagegen sind solche, die gleichzeitig bearbeitet werden.

Bezüglich Parallelismus wird meist zwischen Parallelismus auf Hardware-Ebene und Parallelismus auf konzeptioneller oder algorithmischer Ebene unterschieden. Auf Hardware-Ebene existiert der *Bit-Ebenen-Parallelismus*, also die Ausnutzung der Wortbreite, sowie der *Instruktionsebenen-Parallelismus (ILP)*. Letzterer bedeutet im Wesentlichen die gleichzeitige Ausführung und gegebenenfalls die Umordnung der Befehlsfolge mehrerer Instruktionen, ohne das Ergebnis gegenüber einem rein sequentiellen Programmablauf zu verändern.

Auf konzeptioneller bzw. algorithmischer Ebene wird zwischen *Aufgabenparallelismus* und *Datenparallelismus* unterschieden, die wie folgt definiert sind.

Definition 2.4 (Aufgaben-Parallelismus)
Man spricht von Aufgaben-Parallelismus (engl.: Task Parallelism), wenn mehrere verschiedene Aufgaben zu dem jeweiligen Zeitpunkt voneinander unabhängig bearbeitet werden können.

Definition 2.5 (Daten-Parallelismus)
Datenparallelismus (engl.: Data Parallelism) bedeutet, dass eine gegebene Menge an Daten so zerlegt wird, dass eine gegebene Aufgabe mehrfach auf diesen Daten arbeiten kann.

Ein einfaches Beispiel für Aufgabenparallelismus ist das Anwenden verschiedener Operationen auf ein- und denselben Daten, z.B. zum Zweck der Datenauswertung. Datenparallelismus tritt beispielsweise meist bei Operationen auf, die auf Vektoren arbeiten, da sie oftmals auf alle Komponenten des Vektors gleichzeitig angewandt werden können.

Die parallele Abarbeitung einer vorgegebenen Menge an Aufgaben bringt somit

meist einen relativen Geschwindigkeitsvorteil gegenüber der sequentiellen Abarbeitung und bestimmt den sogenannten *Speed-Up* durch Parallelisierung. Dabei existieren jedoch Grenzen. Die wohl bekannteste Schranke für Parallelisierbarkeit einer bestimmten Aufgabe wird durch *Amdahls Gesetz* [7] beschrieben, wonach der sequentielle und damit der nicht-parallelisierbare Teil der Berechnung eine Schranke für den maximal erreichbaren Speed-Up darstellt.

Der Speed-Up, der durch die parallele Ausführung erzielt werden kann, kann für die Optimierung der Energieeffizienz dahingehend benutzt werden, dass sie einen zusätzlichen Freiheitsgrad bezüglich der Ausführungsgeschwindigkeit und der Abbildung auf die zur Verfügung stehenden Berechnungseinheiten bietet. Beispielsweise kann eine bestimmte Anwendung durch die parallele Ausführung auf mehreren Kernen so verteilt werden, dass jeder einzelne Teil in einem Betriebsmodus mit geringerer Leistungsaufnahme (vgl. Abschnitt 2.8.1) ausgeführt werden kann.

2.6 Leistungsaufnahme in integrierten Schaltungen

Die gesamte Leistung, die durch eine integrierte Schaltung aufgenommen wird, ist die Summe aus *dynamischer Verlustleistung* bzw. *dynamischer Leistungsaufnahme*, auch „Schaltungsverlustleistung" genannt, und *statischer Verlustleistung* bzw. *statischer Leistungsaufnahme*, zum Teil auch „passive Verlustleistung" genannt. Im weiteren Verlauf kann der Begriff „Verlustleistung" je nach Gebrauch durch den allgemeineren Begriff „Leistung" ersetzt sein, behält aber die gleiche Bedeutung, wenn inhaltlich im weitesten Sinne von einer Leistungsaufnahme ausgegangen wird.

In CMOS-Technologie, die am Beispiel eines NMOS-Transistors in Abbildung 2.8a dargestellt ist, entsteht die dynamische Verlustleistung P_{dyn} während eines Schaltvorgangs der integrierten Schaltung [103]. Die Energie, die von der Energieversorgung bezogen wird, beträgt $C_L * V_{dd}^2$, wobei C_L die kapazitive Ladung einer integrierten Schaltung und V_{dd} die Versorgungsspannung darstellt. Die Hälfte der Energie wird im Kondensator gespeichert (siehe Abbildung 2.8b), die andere Hälfte wird im Transistor verbraucht. Im Übergang von 1 zu 0 wird keine zusätzliche Leistung von der Energieversorgung bezogen, jedoch wird die gehaltene Ladung im NMOS-Transistor umgewandelt. Wenn man nun annimmt, dass eine Schaltung mit der Frequenz f schaltet, dann ergibt sich die dynamische Leistung bei einer Schaltaktivität von α aus Gleichung 2.1.

$$P_{dyn} = \frac{1}{2} \cdot \alpha \cdot C_L \cdot V_{dd}^2 \cdot f \qquad (2.1)$$

Offensichtlich hat die Reduzierung der Versorgungsspannung einen großen Einfluss auf die Verringerung der dynamische Leistung, da diese als quadratischer Term in Gleichung 2.1 eingeht. Dabei ist allerdings zu beachten, dass die Versorgungsspannung V_{dd} nicht beliebig abgesenkt werden kann, da sich dadurch ebenfalls die Schaltgeschwindigkeiten der Transistoren reduziert. Die resultierende Verzögerung t_d kann laut [103] durch Gleichung 2.2 approximiert werden, wobei V_{th} die Schwellspannung

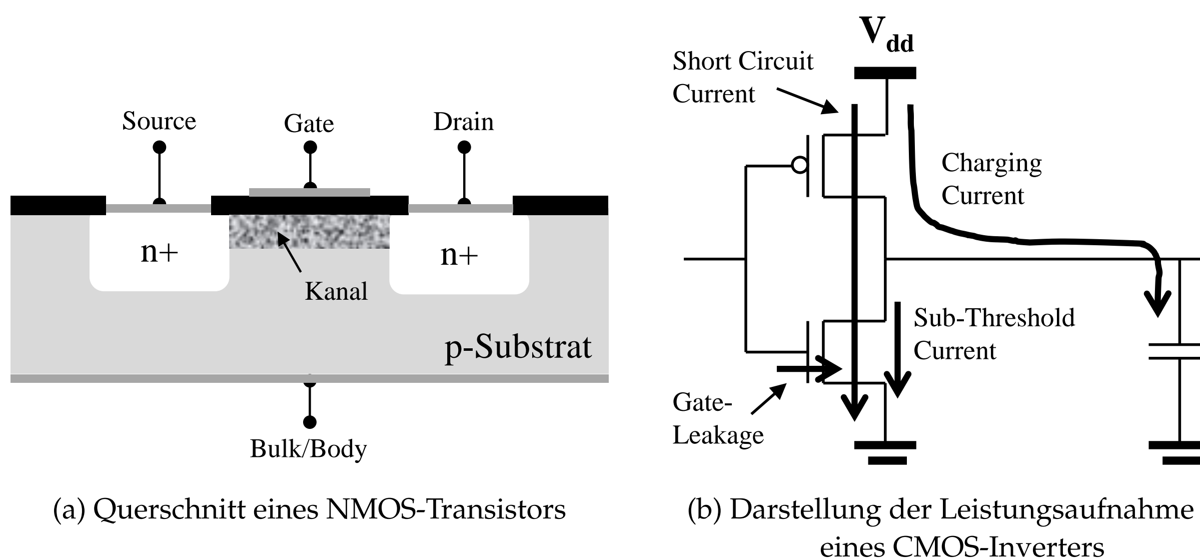
angibt, die die logische 0 von der logischen 1 trennt, also ab der ein Schaltvorgang stattfindet, und k konstant ist.

$$t_d = k \frac{V_{dd}}{(V_{dd} - V_{th})^2} \tag{2.2}$$

Um die Umschaltzeit bei reduzierter Versorgungsspannung wieder zu verkürzen, müsste also die Schwellspannung ebenfalls reduziert werden. Während eines Schaltvorgangs gibt es einen kurzen Zeitbereich, in dem sowohl NMOS-, als auch PMOS-Transistoren leitend sind. Dadurch entsteht ein kurzer Stromfluss zwischen Versorgung und Erdung. Diese Verlustleistung lässt sich durch Gleichung 2.3 beschreiben.

$$P_{sc} = K \cdot (V_{dd} - 2V_{th})^3 \cdot \tau \cdot \alpha \cdot f \tag{2.3}$$

Hier ist K eine Konstante, die sowohl von der verwendeten Technologie als auch von der Transistorgröße abhängt, und τ die Zeit, die das Eingangssignal für einen Schaltvorgang benötigt.



(a) Querschnitt eines NMOS-Transistors

(b) Darstellung der Leistungsaufnahme eines CMOS-Inverters

Abbildung 2.8: Funktionsweise und Leistungsaufnahme von CMOS-Transistoren

In Technologien mit großer Strukturbreite spielte die statische Verlustleistung eine unwesentliche Rolle. Jedoch nimmt mit abnehmender Strukturbreite auch die Dicke der Gate-Oxid-Schicht und die Länge der Gate-Elektrode ab, was zu einer größeren Wahrscheinlichkeit des *Tunneling*-Effekts führt. Dieser ist ein wesentlicher Bestandteil des Leckstroms und erhöht somit die statische Verlustleistung. Die statische Verlustleistung P_{stat} wird deshalb oft auch als *Leakage Power* bezeichnet und lässt sich durch Gleichung 2.4 berechnen. Die resultierenden Leckströme sind in Abbildung 2.8b grafisch dargestellt.

$$P_{stat} = (I_{diode} + I_{subthreshold}) \cdot V_{dd} \tag{2.4}$$

I_{diode} entsteht durch Diodenbildung zwischen Diffusionsregionen und Substrat in den Transistoren. Der Term $I_{subthreshold}$ bezeichnet den Strom, der trotz abgeschalteter Transistoren bezogen wird und durch Gleichung 2.5 berechnet werden kann. Dieser ist abhängig von der Eingangsspannung V_{in},der Schwellspannung V_{th} und der Kanalbreite der Transistoren W_{eff} [103]. K und S sind auch hier wiederum Parameter, die von der verwendeten Technologie abhängen.

$$I_{subthreshold} = K \cdot W_{eff} \cdot e^{\frac{V_{in}-V_{th}}{S}} \tag{2.5}$$

Insgesamt existieren in CMOS-Transistoren mehrere Bestandteile von Leckströmen, die im Folgenden aufgrund der weiten Verbreitung in englischsprachiger Bezeichnung aufgelistet sind:

Gate Oxide Tunneling Leakage In konventionellen CMOS-Techniken wird üblicherweise Siliziumdioxid als Dielektrikum eingesetzt. Bei kleinen Strukturbreiten können Elektronen selbst über diesen Isolator transportiert werden. Es besteht dabei eine exponentielle Beziehung zwischen der Dicke des Isolators und dem resultierenden Leckstrom. Wird also der Isolator halbiert, steigen die Leckströme um den Faktor 4.

Subthreshold-Leakage Um die dynamische Verlustleistung zu minimieren, wird die Versorgungsspannung herabgesetzt. Da dadurch die Transistoren aber langsamer schalten, muss auch die Schwellspannung herabgesetzt werden. Das führt jedoch dazu, dass diese nicht mehr vollständig gesperrt werden können und zwischen den beiden Elektroden *Drain* und *Source* ein Diffusionsstrom existiert.

Reverse-bias Junction Leakages Leckströme, die durch eine Sperrvorspannung zwischen Diffusionsregionen und Wannen bzw. zwischen Wannen und Substrat verursacht wird.

Gate Induced Drain Leakage Leckströme, die durch Feldeffekte an der *Drain*-Elektrode verursacht werden.

Gate Current due to Hot-carrier Injection Durch Löcher im Elektronengitter verursachte Leckströme an der *Gate*-Elektrode.

Durch die in diesem Abschnitt beschriebenen Verlustleistungen wird deutlich, dass die Reduzierung der Versorgungsspannung eine Reduzierung der dynamischen Verlustleistung, jedoch ein Ansteigen der statischen Verlustleistung zur Folge hat. Weiterhin verhält sich die Schaltgeschwindigkeit proportional zur Versorgungsspannung, was zu langsameren Schaltungen bei niedriger Versorgungsspannung führt.

Seit kurzer Zeit existiert mit *FD-SOI* (engl.: Fully Depleted Silicon-On-Insulator) eine neue Technologie, die eine reduzierte Leistungsaufnahme bei gleichzeitig höherer Schaltgeschwindigkeit auch in kleinen Strukturbreiten für Transistoren ermöglicht. Dabei wird zwischen Substrat und Elektronenkanal ein weiterer Isolator integriert, sowie der Abstand zwischen den Elektroden verringert, was zu einem schnelleren Elektronenfluss führt. Um herkömmliche CMOS-Schaltungen schneller zu machen,

wird das Substrat ebenfalls mit einer Spannung versorgt (engl.: Body Biasing), was den Elektronenfluss im Kanal erhöht, jedoch zu Leckströmen führt. Durch den zusätzlichen Isolator des FD-SOI-Transistors werden diese Leckströme auch bei hohen Spannungen des Substrats verhindert. Durch die Möglichkeit, neben der Gate-Elektrode auch das Substrat mit unterschiedlichen Spannungen versorgen zu können, kann die Charakteristik des Transistors angepasst werden, sodass schnell zwischen Zuständen mit hoher Leistungsfähigkeit und Zuständen mit niedriger Leistungsaufnahme gewechselt werden kann. Dies unterstützt die Anwendung von dynamischen Mechanismen zur Reduzierung der Leistungsaufnahme, wie sie in Abschnitt 2.8 vorgestellt werden. Da das Substrat bzw. der Kanal viel weniger dotiert werden muss, ist zudem die durch den Herstellungsprozess hervorgerufene Variabilität der Transistoren geringer, sodass notwendige Toleranzen verringert werden können. Dadurch können bei gleich bleibender Leistung die Schaltungen mit einer niedrigeren Versorgungsspannung betrieben werden.

2.7 Modelle zur Abbildung der Leistungsaufnahme

In diesem Abschnitt werden zwei grundlegende Modelle zur Abbildung der Leistungsaufnahme in digitalen Hardware/Software-Systemen vorgestellt.

2.7.1 Zustandsbasierte Leistungsmodelle

Nach [11] können Komponenten einer Hardware-Plattform in *Power-Manageable Components (PMC)* und *Non-manageable Components* unterschieden werden. Letztere bieten eine feste Leistung zu einem über die Zeit fixen Energieverbrauch. Der Vorteil bei PMCs ist die Möglichkeit, solche Komponenten auf die konkrete Auslastung anzupassen und zwischen Modi mit hoher Leistungsfähigkeit und hohem Energieverbrauch und Energiesparmodi mit geringerer Performanz zu wechseln. Dabei ist ein gewisser Kompromiss zu beachten: Je mehr Modi zur Verfügung stehen, umso genauer lässt sich die Leistung der Komponente anpassen und somit der Energieverbrauch optimieren. Dies führt allerdings zu einer stark erhöhten Hardwarekomplexität, sodass in der Praxis häufig nur eine begrenzte Anzahl von Modi zur Verfügung stehen. Diese Modi stellen die Zustände eines oder mehrerer PMCs dar, die diese während der Ausführung annehmen können, weshalb sie auch als *Ausführungs-* oder *Betriebsmodi* bzw. *Power-Modi* bezeichnet werden. Jedem Modus wird dabei eine meist gemittelte Leistungsaufnahme der korrespondierenden PMC in diesem Modus zugeordnet. Die Berechnung des Energieverbrauchs zur Durchführung einer bestimmten Aufgabe erfolgt über die entsprechende Laufzeit in einem bestimmten Betriebsmodus. Durch die Kopplung des Modells mit der Laufzeit eignet sich dieses Leistungsmodell nicht nur zur Modellierung der Leistungsaufnahme eines Berechnungsressource, z.B. eines Prozessors, sondern kann für beliebige Komponenten eines Hardware/Software-Systems angewandt werden.

Grundsätzlich können bei zustandsbasierten Leistungsmodellen drei wesentliche

Kategorien unterschieden werden:

(1) Aktive Zustände, in denen Energie sowohl wegen statischer, als auch wegen dynamischer Leistungsaufnahme verbraucht wird, z.B. *run*-Zustände

(2) Aktive Zustände, in denen Energie ausschließlich aufgrund statischer Leistungsaufnahme verbraucht wird, z.B. *idle*-Zustände

(3) Inaktive Zustände, in denen weder statische noch dynamische Leistungsausnahme existiert, und daher keine Energie verbraucht wird, z.B. *sleep*-Zustände

Es ist außerdem zu beachten, dass der Wechsel von einem Modus in einen anderen nicht unverzüglich erfolgt und so eine zusätzliche Leistungsaufnahme entsteht. Während eines Wechsels kann die Komponente nicht in ihrem nominalen Zustand arbeiten. Im Allgemeinen dauert der Wechsel von einem Zustand mit niedriger Leistungsaufnahme zu einem Zustand mit hoher Leistungsaufnahme länger als im umgekehrten Fall. Diese zusätzlichen Kosten dürfen nicht vernachlässigt werden, da unter gewissen Umständen manche Wechsel zwischen Betriebsmodi bezüglich des Gesamtenergieverbrauchs nicht mehr sinnvoll sind.

Eine PMC mit diesen Kosten lässt sich in Form einer *Power State Machine (PSM)* modellieren. In diesem Modell werden die verschiedenen Betriebsmodi durch Zustände repräsentiert, die jeweiligen Zustandsübergänge beschreiben die durch einen Moduswechsel hervorgerufenen Kosten in Leistungsaufnahme und benötigter Zeit. Abbildung 2.9 zeigt eine einfache PSM mit drei Zuständen und die jeweils möglichen Zustandsübergänge bzw. Transitionen.

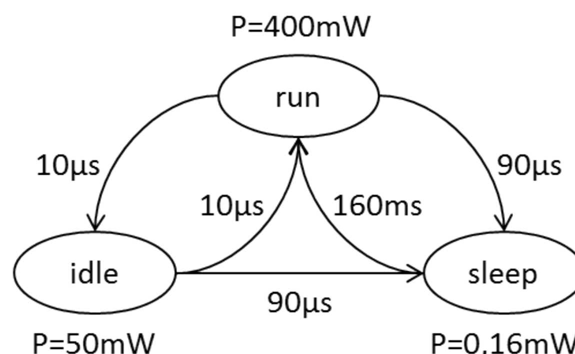


Abbildung 2.9: Power State Machine eines Strong-ARM-Prozessors SA-1100 [11]

Ein *Power-Manager* ist für die Kontrolle der Komponenten zuständig, dargestellt in Abbildung 2.10. Er besteht aus zwei Teilen: einem Observer, der Informationen über die PMCs sammelt, und einem Controller, der die Steuerung der Zustandsübergänge durchführt. Die Kontrollprozedur wird *(Power-)Policy* genannt und basiert auf Überwachung der derzeitigen Auslastung bzw. Annahmen über die zukünftige Auslastung. Ein einfaches Beispiel, das in den meisten kommerziellen Betriebssystemen implementiert ist, ist die *Timeout*-Policy: Eine Komponente wird nach einer definierten Zeit der Inaktivität abgeschaltet aufgrund der Annahme, dass sie wahrscheinlich auch in naher

Zukunft inaktiv bleiben wird. Neben den extern gesteuerten Komponenten, die vom Power-Manager überwacht werden, gibt es auch intern gesteuerte Komponenten. Diese haben keine Sicht auf das Gesamtsystem, sodass sie selbständig und unabhängig vom Gesamtsystemzustand die jeweiligen Zustandsübergänge auslösen.

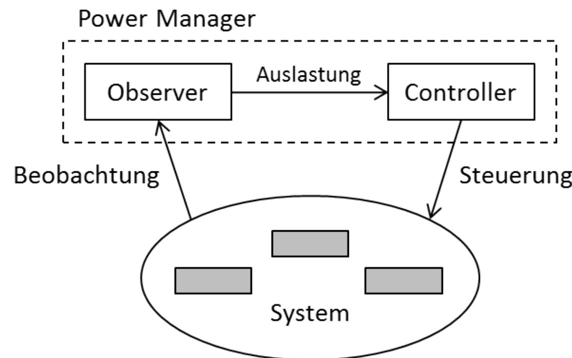


Abbildung 2.10: Schematischer Aufbau eines Power-Managers (nach [11])

Wenn eine *Power-Manageable Component (PMC)* in einen inaktiven Zustand z_{off} versetzt wird, entsteht eine Inaktivität der Dauer t_n [11]. Diese setzt sich aus der Zeit im Zustand z_{off}, nämlich t_{off}, und den Zustandsübergangszeiten für Hin- und Rückwechsel t_{tr} zusammen.

$$t_n = t_{off} + t_{tr}$$
$$t_{tr} = t_{on\to off} + t_{off\to on}$$

Zusätzlich kommt noch die Verlustleistung p_{tr} für die Zustandsübergänge hinzu.

$$p_{tr} = \frac{t_{on\to off} * p_{on\to off} + t_{off\to on} * p_{off\to on}}{t_{tr}}$$

Die *Break-Even-Zeit* $t_{be,off}$ ist definiert als die minimale Zeit der Inaktivität, die nötig ist, um die Kosten für die Zustandsübergänge zu amortisieren.

$$t_{be,off} = t_{tr} + t_{tr}\frac{p_{tr} - p_{on}}{p_{on} - p_{off}}$$

Für den Fall $t_n < t_{be,off}$ lohnt sich ein Wechsel zur Energieminimierung also nicht. Wenn $p_{tr} \leq p_{on}$ gilt, zum Beispiel für mechanische Komponenten, dann gilt $t_{be} = t_{tr}$. Für die PSM in Abbildung 2.10 wäre eine optimale Policy das Wechseln in den Zustand z_{off}, wenn die inaktive Zeit t_{idle} größer ist als t_{be}, und zurück zu z_{on}, wenn die Komponente rechtzeitig wieder bereit zur Ausführung sein muss. Die Verfolgung dieser Strategie bildet eine untere Schranke für den Energieverbrauch der PMC. Die so gesparte Energie $E_{save}(t_{idle})$ lässt sich berechnen durch

$$E_{save}(t_{idle}) = (t_{idle} - t_{tr})(p_{on} - p_{off}) + t_{tr}(p_{on} - p_{tr}).$$

2.7.2 Instruktionsbasierte Leistungsmodelle

Vor der Ausführung eines Programms muss dieses auf die Menge der auf der Zielarchitektur zur Verfügung stehenden *Instruktionen* abgebildet werden, was einem im Befehlssatz definierten Code für einen bestimmten Maschinenbefehl entspricht. Eine solche Instruktion ist hochgradig abhängig von der zugrunde liegenden Zielarchitektur, besteht aber im Wesentlichen aus dem kodierten Befehl zur Identifizierung der Instruktion und den benötigten Operanden. Die Art der Instruktion legt fest, welche Teile der Zielarchitektur aktiv sind, was wiederum einen maßgeblichen Anteil an der Leistungsaufnahme der Zielarchitektur hat.

Diese Abhängigkeit macht man sich in instruktionsbasierten Leistungsmodellen zunutze, indem jeder ausführbaren Instruktion $i \in I$ der Zielarchitektur durch eine Abbildung Φ eine bestimmte Leistungsaufnahme zugewiesen wird [125]:

$$\Phi : I \rightarrow \mathbb{R}$$

Ein Verfahren, wodurch die Abbildung Φ ermittelt werden kann, wird in [107] thematisiert. Der Ansatz basiert auf der simulativen Ausführung von mehreren einfachen Binärprogrammen auf dem Modell einer ARM-ähnlichen Architektur, die bis auf Gatterebene synthetisiert wurde. Die auf dieser Ebene analysierten Schaltaktivitäten werden daraufhin soweit abstrahiert, dass eine Charakterisierung der Leistungsaufnahme für einzelne funktionale Komponenten der Architektur, wie z.B. Pipeline-Stufen und Registereinheiten, vorgenommen werden kann. Basierend auf dieser Charakterisierung und der Zuordnung von Instruktionen zu funktionalen Komponenten kann die Leistungsaufnahme jeder Komponente bei Ausführung einer bestimmten Instruktion abgeleitet werden.

Die Analyse der Leistungsaufnahme wird in instruktionsbasierten Leistungsmodellen also auf die Analyse der ausgeführten Instruktionen bei einer bestimmten Programmausführung abgebildet, wodurch von der reinen Leistungsaufnahme abstrahiert wird. Durch die Abstraktion können instruktionsbasierte Leistungsmodelle in Entwurfsprozessen angewandt werden, die eine Sicht auf das auszuführende Binärprogramm bieten. Steht also fest, welche Instruktionen während einer Programmausführung auf der Zielarchitektur ausgeführt werden und wie oft diese jeweils ausgeführt werden, angegeben durch h_i kann die Leistungsaufnahme für n unterschiedliche Instruktionen i wie folgt berechnet werden:

$$p_{total} = \sum_{k=1}^{n} \Phi(i_k) * h_{i_k}$$

Das *Integral* über die für die Ausführung dieser Instruktionen benötigte Ausführungszeit bestimmt den *Energieverbrauch*.

Ein Leistungsmodell, das rein auf den ausgeführten Instruktionen basiert, abstrahiert jedoch von einigen Eigenschaften der Zielarchitektur, die einen Einfluss auf die Leistungsaufnahme haben. Moderne Rechenarchitekturen verfügen beispielsweise über begrenzten Zwischenspeicher für die Instruktionsdekodierung (engl.: Instruction-Cache), sodass der Aufwand der Dekodierung nicht nur von der aktuell ausgeführten

Dekodieraufgabe, sondern von den bereits zuvor ausgeführten Aufgaben abhängt. Dieser geltende Kontext kann aber einen signifikanten Einfluss auf die Leistungsaufnahme des Systems haben, was bedeutet, dass die Leistungsaufnahme nicht nur durch die Instruktion an sich, sondern auch durch die *Abfolge der Instruktionen* bei der Ausführung abhängt. Weiterhin spielt der Wertebereich, in dem sich die *Operanden* der Instruktion während der Ausführung befinden, eine Rolle. Die Belegung der Operanden entscheidet unter Umständen darüber, welche Teile der integrierten Schaltung umgeladen werden müssen. Wie sich die Leistungsaufnahme in integrierten Schaltungen zusammensetzt, kann in den Erläuterungen in Abschnitt2.6 nachgelesen werden. Die Auswirkung, die verschiedene Operanden und damit verschiedene Eingabedaten auf die Leistungsaufnahme einer ausgeführten Instruktion haben, hängt dabei maßgeblich von der Instruktion selbst ab und kann nicht pauschal angegeben werden. Erste Arbeiten zur Erforschung von instruktionsbasierten Leistungsmodells, die dynamische Effekte wie Instruktionsabfolgen und datenabhängige Belegungen der Operanden integrieren, wurden im Projekt *EASY*[13] [57] für eine ARM-Architektur durchgeführt, es fehlen jedoch weiterführende Arbeiten.

2.8 Strategien zur Reduzierung der Leistungsaufnahme

Nach [11] existieren zwei grundsätzliche Mechanismen zur Reduzierung der Leistungsaufnahme in Hardware/Software-Systemen – statische und dynamische Mechanismen. Wie die Leistungsaufnahme bestimmt werden kann, wird in Abschnitt 2.6 erläutert.

Für die Optimierung der Energieeffizienz auf hoher Abstraktionsebene sollen zwei grundlegende Techniken zur Reduzierung der Leistungsaufnahme untersucht und angewandt werden: *Dynamic Voltage and Frequency Scaling (DVFS)* und *Dynamic Power Management (DPM)*. Diese werden durch die folgenden Abschnitte im Detail vorgestellt.

2.8.1 Dynamic Voltage and Frequency Scaling

Die Technik des *Dynamic Voltage and Frequency Scaling (DVFS)* wurde ursprünglich für mobile Systeme entwickelt [98]. Denn besonders bei diesen Systemen gibt es aufgrund der begrenzten zur Verfügung stehenden Energie einen Konflikt zwischen Batterielebensdauer auf der einen und Rechenleistung auf der anderen Seite. Prominente Beispiele hierfür sind u.a. Mobiltelefone, Tablet-Computer, digitale Multimediageräte und verteilte Sensor/Aktor-Systeme. Trotz der stetigen Entwicklung sowohl in der Halbleiter-, als auch in der Batterietechnik wird dieser Konflikt auch in Zukunft bestehen bleiben.

Die DVFS-Technik versucht, den Energieverbrauch dieser Systeme zu minimieren, indem es zwei wichtige Systemeigenschaften ausnutzt. Zum einen wird die volle Rechenleistung nur für kurze Zeit benötigt, da die gestellten Anforderungen über die Laufzeit normalerweise deutlich niedriger sind. Für die meiste Zeit wäre also eine oder mehrere Verarbeitungseinheiten mit weniger Leistung ausreichend. Zum anderen sind

[13]Energy-Aware SYstem-on-chip design of the HIPERLAN/2 standard

diese Verarbeitungseinheiten meist in CMOS-Technologie realisiert, sodass nicht nur die maximale Frequenz und damit die Verarbeitungsgeschwindigkeit, sondern auch die Versorgungsspannung der integrierten Schaltung reduziert werden kann. Durch die quadratische Abhängigkeit von benötigter Energie und Versorgungsspannung, ausgedrückt durch

$$E \propto V_{dd}^2 \tag{2.6}$$

, bietet sich ein großes Energiesparpotential. Ein DVFS-System besteht aus mindestens einem programmierbaren DC-DC Schaltnetzteil, einem programmierbaren Taktgenerator und einem Prozessor mit einem möglichst großen Frequenzbereich. Um mögliche Spitzenlasten bewältigen zu können, arbeitet dieser meist standardmäßig mit maximaler Frequenz. Wenn die Auslastung des Systems niedrig ist, wird die Frequenz der Verarbeitungseinheit so reduziert, dass die Rechenleistung noch ausreicht, um die an das System gestellten Anforderungen bedienen zu können. Dadurch kann unter Umständen die Versorgungsspannung abgesenkt und der Energieverbrauch für die Ausführung der Aufgabe reduziert werden. DVFS-Techniken werden also angewandt, um die Energieeffizienz während der Ausführung zu erhöhen, weshalb hauptsächlich der dynamische Anteil der Leistung P_{dyn} beeinflusst wird.

Aufgrund des Freiheitsgrades, mit dem die Versorgungsspannung geändert werden kann, können verschiedene DVFS-Systeme in der Theorie wie folgt definiert werden [102]:

Ideal: Die Spannung kann unverzüglich gewechselt werden.

Erfüllbar: Die Spannung kann zwischen den Werten V_{dd}^{min} und V_{dd}^{max} mit der maximalen Rate R wechseln.

Praktisch: Wie erfüllbar, aber während der Spannungsänderung muss die Ressource pausieren.

Mehrfach: Es steht eine Menge von diskreten Spannungen zur Verfügung, zwischen denen unverzüglich gewechselt werden kann.

Durch diese Einschränkungen ergeben sich verschiedene Strategien, um einen möglichst energieeffizienten Ablaufplan zu erstellen. Ein ideales System kann direkt zur optimalen Ausführungsgeschwindigkeit wechseln. Für ein Mehrfachsystem muss zuerst die Spannung gefunden werden, mit der die Anforderung an Verarbeitungsperformanz noch eingehalten werden kann – der Wechsel zu dieser erfolgt jedoch unverzüglich. Bei einem erfüllbaren oder praktischen System kann die Spannung kontinuierlich angehoben bzw. gesenkt werden. Dazu muss zuerst eine Zielspannung gefunden werden, die mit der Rate R in der verfügbaren Zeit auch erreichbar ist.

In der Realität wird meist eine Kombination aus dem Mehrfach- und praktischen Modell verwendet. Einer Ressource stehen endlich viele diskrete Versorgungsspannungen zur Verfügung. Um die Spannung zu wechseln, muss die Verarbeitungseinheit der Ressource aber solange angehalten werden, bis ein stabiler Zustand angenommen wird. Diese Einschränkungen sind auch in vielen realen Prozessoren vorhanden. Im Zusammenhang mit Echtzeitsystemen ergeben sich allerdings Probleme, da sich

mit der Änderung der Frequenz auch die Ausführungszeit einer durchzuführenden Aufgabe ändert. Somit besteht die Gefahr, dass diese die an die Aufgabe gestellten Anforderungen nicht mehr einhalten kann.

Eine ausführliche Übersicht über verschiedene DVFS-Scheduling-Algorithmen, die Echtzeitbedingungen erfüllen, ist in [18] gegeben.

2.8.2 Dynamic Power Management

Wie beim *Dynamic Voltage and Frequency Scaling* (siehe Abschnitt 2.8.1) ist auch beim *Dynamic Power Management (DPM)* die Motivation, den Energiebedarf eines Systems zu minimieren und eine Verlängerung der Batterienutzungsdauer sowie eine Reduzierung der Umweltbeeinflussung, z.B. durch erhöhte Temperaturentwicklung und der zur aktiven und passiven Kühlung benötigte Aufwand (z.B. durch einen Prozessorkühler), zu ermöglichen. Ein elektronisches System kann dabei als Menge von Komponenten modelliert werden, wobei einzelne Teile auch mechanischer Natur sein können [11]. Als Beispiel stelle man sich ein Mobiltelefon vor, bestehend aus Display, mehreren Funkmodulen für die jeweiligen Technologien und mehreren Berechnungsressourcen. Analog zu DVFS muss ein solches System in bestimmten Fällen den maximalen Funktionsumfang abrufen können, dies ist jedoch nur selten nötig. Abhängig von der jeweilig betrachteten Komponente gibt es durchaus Zeitbereiche, in denen einige dieser Komponenten komplett abgeschaltet werden können, falls sie nicht benötigt werden, oder die Aufgabe von einer anderen Komponente übernommen werden kann. Dadurch kann also die zur Aufgabenerfüllung benötigte Energie aufgrund statischer Leistungsaufnahme

$$E_{stat} = P_{stat} \cdot t \tag{2.7}$$

optimiert werden. DPM versucht deshalb, durch dynamische Rekonfiguration des Systems die geforderte Leistung mit einer möglichst geringen Menge aktiver Komponenten, oder geringer Auslastung dieser, zu erbringen. Im Gegensatz zu DVFS sind die Techniken bei DPM auf die untätigen Bereiche der Komponenten ausgerichtet. Aber auch hier sind wieder Informationen über die aktuelle und zukünftige Auslastung notwendig – vor allem in einem Echtzeitsystem, um die Einhaltung aller strikten Anforderungen zu gewährleisten.

Bei der Anwendung von DPM-Techniken muss stets beachtet werden, dass sowohl der Vorgang des Abschaltens als auch des Anschaltens von Komponenten mit einem gewissen Zeitaufwand verbunden ist, in dem die Komponente nicht für die eigentliche Aufgabe genutzt werden kann. Während des Abschalt-Übergangs muss meist der aktuelle Zustand des Systems gesichert werden, welcher beim Anschalt-Übergang wieder hergestellt werden muss, bevor eine weitere Ausführung möglich ist. Weiterhin müssen dabei alle benötigten Signale wieder stabil an den Eingängen anliegen, was z.B. bei Taktsignalen besonders wichtig ist.

Ein weitergehender Überblick über verschiedene DPM-Techniken ist in [11] gegeben.

2.9 Lösung von ganzzahligen Optimierungsproblemen

In diesem Abschnitt soll zunächst eine Einschränkung auf rein lineare Optimierungsprobleme erfolgen, zur Lösung nichtlinearer Optimierungsprobleme sei auf Abschnitt 2.10 verwiesen.

Lineare Optimierungsprobleme bilden eine Problemklasse, für die bereits relativ effiziente und schnelle Lösungsverfahren existieren. Diese können im Fall von Variablenbelegungen durch reelle Zahlen durch *Innere-Punkte-Verfahren* oder durch das *Simplex-Verfahren* gelöst werden. In der linearen Optimierung, die eine Spezialisierung der konvexen Optimierung darstellt, kann die Lösungsmenge geometrisch durch einen Polyeder beschrieben werden. Die jeweiligen Optima liegen aufgrund der gegebenen Konvexität auf dem Rand bzw. den Eckpunkten dieses Polyeders und gelten global. Das Simplex-Verfahren läuft beim Prozess der Lösungsfindung diese Ecken ab und hat in der Regel polynomiale Komplexität. Durch lineare Nebenbedingungen kann der Polyeder weiterhin beschränkt werden, da die Nebenbedingungen einen Teil des Polyeders abtrennen und somit den Suchraum einschränken. Ist der Suchraum nicht beschränkt, können prinzipiell unendlich viele optimale Lösungen existieren.

Bei ganzzahligen linearen Optimierungsproblemen sind zumindest ein Teil der Lösungsvariablen aus dem ganzzahligen Raum. Dies führt zu einer steigenden Komplexität bei der Lösungsfindung, sodass diese Probleme oftmals in die Klasse der NP-harten Probleme fallen. Dies liegt mitunter daran, dass es nicht mehr ausreicht, die Eckpunkte Lösungsmengen-Polyeders zu betrachten, da diese nicht zwangsläufig ganzzahligen Werten entsprechen, sodass im schlimmsten Fall alle möglichen Lösungen untersucht werden müssen.

Aus diesem Grund werden ganzzahlige Optimierungsprobleme oftmals durch Verfahren nach dem Teil-und-Herrsche-Prinzip gelöst. Das bekannteste unter diesen Verfahren ist das *Branch-and-Bound-Verfahren*, das im folgenden Abschnitt erklärt wird.

2.9.1 Branch-and-Bound-Verfahren

Da die Evaluierung des gesamten zulässigen Lösungsraum meist zu inakzeptablen Rechenzeiten führt, wird der Lösungsraum beim Branch-and-Bound-Verfahren in einem ersten Schritt in mehrere Teilbereiche aufgespalten. Dieser Vorgang wird auch *Branch*-Schritt genannt, da sich durch rekursives Ausführen eine Baumstruktur ergibt, in der sich jeder Zweig von der Wurzel bis zum Blatt als Festlegung einer bestimmten Lösungsvariablen deuten lässt. Somit sind in den Blättern alle Lösungsvariablen festgelegt und der Funktionswert kann anhand dieser Festlegung berechnet werden. Ein Spezialfall bei der ganzzahligen Optimierung nehmen binäre Lösungsvariablen ein, die nur den Wert 0 oder 1 haben dürfen, da diese zumeist als Entscheidungsvariablen interpretiert werden können. Eine bestimmte Untermenge dieser binären Lösungsvariablen, in der nur genau eine der Lösungsvariable den Wert 1 haben darf, wird als *Special-Ordered-Set-1* bezeichnet. Diese Bedingung muss während des Branching beachtet werden.

Für das eingeschränkte und dadurch vereinfachte Problem werden in einem zweiten

Schritt obere bzw. untere Schranken gesucht. Durch Interpretation dieser Schranken wird versucht, Rückschlüsse auf eine Menge von Teilproblemen zu ziehen, sodass diese eventuell frühzeitig aus der Menge an möglichen Lösungen ausgeschlossen werden kann. Dieser Schritt wird als *Bound*-Schritt bezeichnet. Können keine Teilprobleme frühzeitig ausgeschlossen werden, führt das Verfahren allerdings zu einer vollständigen Betrachtung aller Lösungen.

2.10 Lösung von nichtlinearen Optimierungsproblemen

Mathematische Optimierungsprobleme und existierende Lösungsverfahren durch lineare, nichtlineare und ganzzahlige Programmierung werden grundlegend in [89] und [86] erklärt. In [121] wird insbesondere deren Anwendung bei der Synthese und Optimierung digitaler Hardware/Software-Systeme vorgestellt. Hierbei ist die Grundidee, das zu optimierende System in Form mathematischer Funktionen und zu geltende Randbedingungen in Form von (Un-)Gleichungen so zu beschreiben, dass eventuelle Optima durch die Anwendung mathematischer Verfahren ermittelt werden können. Beim Lösen von nichtlinearen Optimierungsproblemen zählt das *Sequential Quadratic Programming (SQP)* zu den am besten geeigneten Verfahren [78]. Dies gilt sowohl aus Sicht der zur Lösungsfindung benötigten Berechnungszeit als auch für die Qualität der Ergebnisse. Im Folgenden sollen Schritt für Schritt Verfahren zur Lösung von Optimierungsproblemen hergeleitet werden. Darin enthaltene Erklärungen und Definitionen sind zum Teil aus [30] entnommen.

2.10.1 Grundbegriffe

Ein allgemeines *nichtlineares Optimierungsproblem (NLP)* hat die Form

$$\begin{aligned} \text{minimiere} \quad & f(x), & \\ \text{sodass gilt} \quad & g_i(x) \geq 0, & i \in I, \\ & g_i(x) = 0, & i \in E. \end{aligned}$$

Die Funktionen f und g sind (ausreichend oft) stetig differenzierbar. I steht für die Menge der Indizes der Ungleichheitsbedingungen, E für die Indizes der Gleichheitsbedingungen. Ein Punkt x^* ist ein globales Minimum, wenn x^* die Bedingungen g_i erfüllt und außerdem gilt: $f(x^*) \leq f(x)$. Wenn es hingegen ein $\epsilon > 0$ gibt, sodass $f(x^*) > f(x)$ nur dann gilt, wenn $\| x - x^* \| < \epsilon$, dann spricht man von einem lokalen Minimum.

Eine Menge F ist konvex, wenn gilt:

$$(1 - \alpha)x + \alpha y \in F \quad \forall x, y \in F, \alpha \in [0, 1].$$

Dies lässt sich auch auf Funktionen übertragen. f ist eine konvexe Funktion auf F, wenn gilt:

$$f((1 - \alpha)x + \alpha y) \leq (1 - \alpha)f(x) + \alpha f(y) \quad \forall x, y \in F, \alpha \in [0, 1].$$

Solche Funktionen haben die Eigenschaft, dass alle lokalen Minima zugleich globale Minima sind.

Als letztes soll noch die positive Definitheit für Matrizen definiert werden, was in kommenden Abschnitten für die Optimalitätsbedingungen benötigt wird. Eine symmetrische Matrix H heißt positiv definit, abkürzend mit $H > 0$ bezeichnet, wenn gilt:

$$p^T Hp > 0 \quad \forall p \in R^n, p \neq 0.$$

Eine symmetrische Matrix H heißt hingegen positiv-semidefinit, auch als $H \geq 0$ dargestellt, wenn gilt:

$$p^T Hp \geq 0 \quad \forall p \in R^n, p \neq 0.$$

2.10.2 Optimalitätsbedingungen

Eine Vielzahl von Verfahren zur Lösung von Optimierungsproblemen basieren auf Suchverfahren, die sich schrittweise in einer bestimmten Richtung dem jeweils gesuchten Optimum annähern. Dabei heißt eine Richtung p *erfüllbar* für $x^* \in F$, wenn es ein $\alpha' > 0$ gibt, sodass gilt:

$$x^* + \alpha p \in F \quad \forall \alpha \in [0, \alpha'].$$

Wenn sich an der Stelle x^* ein lokales Minimum befindet, dann gibt es keine erfüllbare Richtung p, sodass gilt:

$$\triangledown f(x^*)^T p < 0.$$

Hierbei handelt es sich um eine Optimalitätsbedingung ersten Grades. Für eine Bedingung zweiten Grades muss zusätzlich noch gelten, dass es keine erfüllbare Richtung p gibt, sodass gilt:

$$\triangledown f(x^*)^T p < 0 \quad \text{und} \quad p^T \triangledown^2 f(x^*)^T p < 0.$$

Dies folgt aus einer Taylorreihenentwicklung von f in x^*. Für den unbeschränkten Fall muss also $\triangledown f(x^*) = 0$ und $\triangledown^2 f(x^*) \geq 0$ sein.

2.10.2.1 Lineare Nebenbedingungen

Diese Optimalitätsbedingungen werden zuerst für lineare Nebenbedingungen verfeinert. Das Problem $P_=$ enthält dabei nur Gleichheitsbedingungen:

$$\begin{array}{lll} (P_=) & \text{minimiere} & f(x), \\ & \text{sodass gilt} & Ax = b. \end{array}$$

Optimierungsprobleme mit Nebenbedingungen können mithilfe der Lagrange-Multiplikatorenregel zu einem Problem ohne Nebenbedingungen umformuliert werden. Dazu

wird eine neue skalare Variable für jede Nebenbedingung eingeführt – den sogenannten *Lagrange-Multiplikator* – und eine Linearkombination definiert, die diese Lagrange-Multiplikatoren als Koeffizienten enthält. Die Lagrange-Funktion L kombiniert die ursprünglich zu optimierende Funktion mit der neu gebildeten Linearkombination und ist dann definiert als

$$L(x, \lambda) = f(x) - \lambda^T(Ax - b),$$

womit sich die Optimalitätsbedingung erster Ordnung formulieren lässt als

$$\begin{pmatrix} \nabla_x L(x^*, \lambda^*) \\ \nabla_\lambda L(x^*, \lambda^*) \end{pmatrix} = \begin{pmatrix} \nabla f(x^*) - A^T \lambda^* \\ b - Ax^* \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Dabei wird der bereits erwähnte Lagrange-Multiplikator mit λ^* bezeichnet, für den gilt:

$$\nabla f(x^*) = A^T \lambda^*.$$

Für lineare Ungleichheitsbedingungen ist das Problem $P_\geq$ definiert als

$$\begin{array}{lll} (P_\geq) & \text{minimiere} & f(x), \\ & \text{sodass gilt} & Ax \geq b. \end{array}$$

Hierbei lässt sich A unterteilen in A_A und A_I, wobei gilt:

$$\begin{aligned} A_A x^* &= b_A \quad \text{und} \\ A_I x^* &> b_I. \end{aligned}$$

Die Nebenbedingungen A_A werden dabei als *aktiv* bezeichnet. Damit lauten die Optimalitätsbedingungen

$$\begin{aligned} Ax^* &\geq b \quad \text{und} \\ \nabla f(x^*) &= A_A^T \lambda_A^*, \quad \lambda_A^* \geq 0. \end{aligned}$$

Diese Bedingungen werden auch *Karush-Kuhn-Tucker*-Bedingungen genannt. Aus den vorgestellten Teilproblemen $P_=$ und $P_\geq$ lässt sich das Problem P mit Ungleichheits- und Gleichheitsbedingungen formulieren:

$$\begin{array}{lll} (P) & \text{minimiere} & f(x), \\ & \text{sodass gilt} & a_i^T x \geq b_i, i \in I, \\ & & a_i^T x = b_i, i \in E. \end{array}$$

Wenn x^* ein lokales Minimum für P ist, dann gibt es ein λ^*, sodass gilt:

$$\begin{aligned} a_i^T x &\geq b_i, i \in I, \\ a_i^T x &= b_i, i \in E, \\ \nabla f(x^*) &= A^T \lambda^*, \\ \lambda_i^*(a_i^T x^* - b^i) &= 0, \quad \lambda_i^* \geq 0 \quad \forall i \in I. \end{aligned}$$

2.10.2.2 Nichtlineare Nebenbedingungen

Die Funktion $g(x)$ beschreibt die nichtlinearen Nebenbedingungen. Diese können analog zu den linearen Nebenbedingungen als Matrix

$$A(x) = \begin{pmatrix} \nabla g_1(x)^T \\ \vdots \\ \nabla g_m(x)^T \end{pmatrix}$$

dargestellt werden.

Ein Punkt x^* heißt *regulär* für P, wenn die Matrix $A(x^*)$ vollen Rang hat, das heißt die $\nabla g_i(x^*)$ sind linear unabhängig. Für das nichtlineare Problem

$$\begin{aligned} (P) \quad & \text{minimiere} && f(x), \\ & \text{sodass gilt} && g_i(x) \geq 0, i \in I, \\ &&& g_i(x) = 0, i \in E \end{aligned}$$

können somit analog zum linearen Fall die Optimalitätsbedingungen ersten Grades formuliert werden: Wenn x^* ein regulärer Punkt und lokales Minimum für P ist, dann existiert ein λ^*, sodass gilt:

$$\begin{aligned} g_i(x) &\geq 0, i \in I, \\ g_i(x) &= 0, i \in E, \\ \nabla f(x^*) &= A(x^*)^T \lambda^*, \\ \lambda_i^* g_i(x^*) &= 0, \quad \lambda_i^* \geq 0 \quad \forall i \in I. \end{aligned}$$

2.10.3 Quadratische Optimierungsprobleme

Im Gegensatz zu einem (nicht-)linearen Optimierungsproblem ist bei einem *Quadratischen Optimierungsproblem (QP)* die Kostenfunktion quadratisch. Auf Nebenbedingungen wird bei der folgenden Formulierung vorerst verzichtet:

$$(QP) \quad \text{minimiere } \frac{1}{2} x^T H x + c^T x.$$

Wenn $H > 0$, dann gibt es ein eindeutiges globales Minimum x^* für das QP, das anhand $Hx^* = -c$ berechnet werden kann. Ansonsten können mehrere globale Minima existieren. Ein QP kann durch einen *Line-Search*-Algorithmus gelöst werden. Dabei handelt es sich um ein iteratives Verfahren, bei dem in jedem Schritt k zunächst eine Suchrichtung p_k berechnet wird, sodass gilt:

$$\nabla f(x_k)^T p_k < 0.$$

Die Schrittweite α_k wird bestimmt, indem

$$\min_{\alpha \geq 0} f(x_k + \alpha p_k)$$

approximiert wird. Der nächste Punkt x_{k+1} setzt sich dann zusammen aus

$$x_{k+1} = x_k + \alpha_k p_k.$$

2.10.3.1 Newton-Verfahren

Das *Newton-Verfahren* ist ein iteratives Lösungsverfahren, das auch auf quadratische Optimierungsprobleme angewandt werden kann. Ausgehend von einem gewählten Startwert x_0 konvergiert das Verfahren über eine Folge von x_k gegen den Punkt x^*, sodass gilt: $\nabla f(x^*) = 0$. Dies entspricht der Optimalitätsbedingung ersten Grades für ein Problem ohne Nebenbedingungen. Das Newton-Verfahren ist wohldefiniert, wenn $\nabla^2 f(x_k) \succ 0$. Die Suchrichtung p wird durch eine quadratische Approximation der Kostenfunktion $f(x_k)$ bestimmt:

$$\min_p \nabla f(x_k)^T + \frac{1}{2} p^T \nabla^2 f(x_k) p.$$

Mithilfe einer Taylor-Entwicklung zweiten Grades kann der nächste Punkt $x_{k+1} = x_k + p$ berechnet werden durch:

$$f(x_k + p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T \nabla^2 f(x_k) p + o(\| p \|^2).$$

Dabei gibt $o(\| p \|^2)$ den Fehler der Approximation an. Der Extremwert wird erreicht, wenn gilt:

$$\nabla^2 f(x_k) p_k = - \nabla f(x_k).$$

Daraus lässt sich auch der nächste Wert $x_{k+1} = x_k + p_k$ bestimmen:

$$x_{k+1} - x_k = p_k = -\frac{\nabla f(x_k)}{\nabla^2 f(x_k)}.$$

Die Schrittweite α kann beim Newton-Verfahren konstant sein, also z.B. auf $\alpha = 1$ gesetzt werden:

$$x_{k+1} = x_k + \alpha p_k.$$

Um die Konvergenz des Verfahrens zu verbessern, ist es aber auch möglich, α durch einen Line-Search-Algorithmus zu bestimmen.

2.10.3.2 Quasi-Newton-Verfahren

Wie in Abschnitt 2.10.3.1 dargestellt, muss beim Newton-Verfahren die Inverse der Hesse-Matrix berechnet werden, welche auch wieder positiv definit sein muss. Das *Quasi-Newton-Verfahren* umgeht diesen Schritt, indem $\frac{1}{\nabla^2 f(x_k)}$ durch die Matrix B_k ersetzt wird:

$$\begin{aligned} x_{k+1} - x_k = p_k &= - \nabla f(x_k) B_k \quad \text{und} \\ B_k &= B_k^T \succ 0. \end{aligned}$$

Als nächstes wird die Schrittweite α_k durch

$$\min_{\alpha \geq 0} f(x_k + \alpha p_k)$$

approximiert. Somit kann das nächste $x_{k+1} = x_k + \alpha_k p_k$ berechnet werden. Zum Schluss muss noch B_k aktualisiert werden. Das Newton-Verfahren liefert für $\nabla f(x_k)$ und $\nabla f(x_{k+1})$

$$\begin{aligned}\nabla f(x_k) &= -\nabla^2 f(x_k)(x - x_k) \quad \text{und}\\ \nabla f(x_{k+1}) &= -\nabla^2 f(x_{k+1})(x - x_{k+1}).\end{aligned}$$

y_k sei definiert als

$$\begin{aligned}y_k &= \nabla f(x_{k+1}) - \nabla f(x_k)\\ &= -\nabla^2 f(x_{k+1})(x - x_{k+1}) + \nabla^2 f(x_k)(x - x_k).\end{aligned}$$

Unter der Annahme $\nabla^2 f(x_k) \approx \nabla^2 f(x_{k+1})$ lässt sich y_k vereinfachen zu

$$y_k \approx -\nabla^2 f(x_{k+1})(x_k - x_{k+1}).$$

Somit gilt:

$$B_{k+1} y_k = x_{k+1} - x_k = s_k.$$

Für B_{k+1} wird ein Korrekturterm C_k gewählt, für den gilt:

$$\begin{aligned}B_{k+1} &= B_k + C_k \quad \text{und somit}\\ B_{k+1} y_k &= B_k y_k + C_k y_k = s_k.\end{aligned}$$

Durch Umstellen der Gleichung ergibt sich für C_k:

$$C_k = \frac{s_k - B_k y_k}{y_k}.$$

2.10.3.3 Active-Set

Das Problem *IQP* sei definiert als ein quadratisches Optimierungsproblem mit Ungleichheitsbedingungen:

$$\begin{aligned}(IQP) \quad \text{minimiere} \quad & \frac{1}{2} x^T H x + c^T x,\\ \text{sodass gilt} \quad & Ax \leq b.\end{aligned}$$

Wenn $x^* = x + p^*$ und λ^* das gleiche Problem mit Gleichheitsbedingung $Ax = b$ anstatt der Ungleichheitsbedingung optimal lösen und $H > 0$ gilt, dann lösen sie auch IQP optimal, wenn $\lambda^* \geq 0$.

Angenommen x' sei eine Lösung für IQP und die aktiven Bedingungen seien auch in x^* aktiv:

$$A = \{I : a_I^T x' = b_I\}.$$

Außerdem sei $W \subseteq A$, sodass A_W vollen Zeilenrang hat. Dann kann das Problem EQP_W formuliert werden:

$$(EQP_W) \quad \text{minimiere} \quad \frac{1}{2}(x' + p)^T H (x' + p) + c^T (x' + p),$$

$$\text{sodass gilt} \quad A_W p = 0.$$

Die optimale Lösung von EQP_W ist p^* mit Lagrange-Multiplikator λ_W^* und wird berechnet durch Lösen des linearen Gleichungssystems

$$\begin{pmatrix} H & A_W^T \\ A_W & 0 \end{pmatrix} \begin{pmatrix} p^* \\ -\lambda_W^* \end{pmatrix} = - \begin{pmatrix} Hx' + c \\ Ax - b \end{pmatrix},$$

was der Optimalitätsbedingung erster Ordnung entspricht. Das ursprüngliche Problem IQP wird dann durch $s^* = x' + p^*$ optimal gelöst.

Dabei werden aber die inaktiven Nebenbedingungen nicht beachtet, nämlich $a_i^T x \geq b_i, i \notin W$. Wenn $A(x' + p^*) \geq b$, dann erfüllt $x' + p^*$ alle Bedingungen. Sonst kann die maximale Schrittweite α_{max} berechnet werden, sodass $A(x' + \alpha_{max} p^*) \geq b$ gilt:

$$\alpha_{max} = \min_{i: a_i^T p^* < 0} \frac{a_i^T x' - b_i}{-a_i^T p^*}.$$

Hierbei sind zwei Fälle zu unterscheiden. Wenn $\alpha_{max} \geq 1$, dann sind die Bedingungen erfüllt und $\tilde{x} = x' + p^*$. Wenn $\alpha_{max} < 1$, dann wird $\tilde{x}$ auf $\tilde{x} = x' + \alpha_{max} p^*$ gesetzt und $W \leftarrow W \cup I$ für $a_I^T (x' + \alpha_{max} p^*) = b_I$.

Es wird nach wie vor ignoriert, dass aktive Bedingungen eigentlich Ungleichheitsbedingungen sind. Anstatt $A_W x \geq b_W$ wurde $A_W x = b_W$ gefordert. Wenn $\lambda_W^* \geq 0$, dann ist $\tilde{x}$ optimale Lösung für das Problem IQP_W und somit auch optimale Lösung für IQP:

$$\begin{aligned} (IQP_W) \quad \text{minimiere} \quad & \frac{1}{2} x^T H x + c^T x, \\ \text{sodass gilt} \quad & A_W x \geq b_W. \end{aligned}$$

Wenn ein k existiert, sodass $\lambda_k^* < 0$, dann muss k aus W entfernt werden: $W \leftarrow W \setminus \{k\}$. Denn angenommen $A_W p = e_k$ (e_k ist der Einheitsvektor für k), dann gilt:

$$\begin{aligned} -Hpp + A_W^T \lambda_W^* &= (Hx' + c)p \quad \text{und somit} \\ -Hpp + \lambda_k^* &= (Hx' + c)p. \end{aligned}$$

Für $\lambda_k^* < 0$ kann die linke Seite nicht positiv sein, die Bedingung k ist somit nicht aktiv und muss entfernt werden.

Eine Iteration des Active-Set-Verfahrens verläuft wie folgt:

(1) Löse $\begin{pmatrix} H & A_W^T \\ A_W & 0 \end{pmatrix} \begin{pmatrix} p^* \\ -\lambda_W^* \end{pmatrix} = - \begin{pmatrix} Hx' + c \\ Ax - b \end{pmatrix}$,

(2) $I \leftarrow$ Index der Bedingung, die zuerst entlang p^* verletzt wird,

(3) $\alpha_{max} \leftarrow$ maximale Schrittlänge entlang p^*.

(4) Wenn $\alpha_{max} < 1$, dann setze $x' \leftarrow x' + \alpha_{max} p^*$ und $W \leftarrow W \cup I$. Nächste Iteration.

(5) Sonst ($\alpha_{max} \geq 1$) setze $x' \leftarrow x' + p^*$.

(6) Wenn $\lambda_W^* \geq 0$ dann ist x' optimal. Ende.

(7) Sonst existiert ein k, sodass $\lambda_k^* < 0$. Setze $W \leftarrow W \setminus \{k\}$. Nächste Iteration.

2.10.4 Sequential Quadratic Programming

Das Ziel beim *Sequential Quadratic Programming (SQP)* ist die Approximation eines nichtlinearen Optimierungsproblems durch ein quadratisches Optimierungsproblem. Dieses wird allerdings nicht direkt für x, sondern für die Suchrichtung p gelöst, um dann iterativ $x_{k+1} = x_k + p_k$ zu bestimmen.

Als erstes sei das Problem $P_=$ gegeben, bei dem nur Gleichheitsbedingungen vorkommen:

$$\begin{aligned} (P_=) \quad \text{minimiere} \quad & f(x), \\ \text{sodass gilt} \quad & g_i(x) = 0. \end{aligned}$$

Die Lagrange-Funktion sei außerdem definiert als

$$L(x, \lambda) = f(x) - \lambda^T g(x).$$

Dann ist die Optimalitätsbedingung erster Ordnung

$$\begin{aligned} \nabla L(x, \lambda) &= 0 \\ \begin{pmatrix} \nabla_x L(x, \lambda) \\ - \nabla_\lambda L(x, \lambda) \end{pmatrix} &= \begin{pmatrix} \nabla f(x) - A(x)^T \lambda \\ g(x) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ A(x)^T &= (\nabla g_1(x) \dots \nabla g_m(x)). \end{aligned}$$

Eine Newton-Iteration hat dann die Form

$$\begin{pmatrix} x^+ \\ \lambda^+ \end{pmatrix} = \begin{pmatrix} x \\ \lambda \end{pmatrix} + \begin{pmatrix} p \\ v \end{pmatrix},$$

wobei gilt:

$$\begin{pmatrix} \nabla^2_{xx} L(x, \lambda) & -A(x)^T \\ A(x) & 0 \end{pmatrix} \begin{pmatrix} p \\ v \end{pmatrix} = \begin{pmatrix} - \nabla f(x) + A(x)^T \lambda \\ -g(x) \end{pmatrix}.$$

Diese Gleichung kann umgeformt werden zu

$$\begin{pmatrix} \nabla^2_{xx} L(x, \lambda) & A(x)^T \\ A(x) & 0 \end{pmatrix} \begin{pmatrix} p \\ -\lambda^+ \end{pmatrix} = \begin{pmatrix} - \nabla f(x) \\ -g(x) \end{pmatrix}.$$

Verglichen mit dem quadratischen Problem *EQP*

$$\begin{aligned} (EQP) \quad \text{minimiere} \quad & \frac{1}{2} p^T H p + c^T p, \\ \text{sodass gilt} \quad & Ap = b, \end{aligned}$$

mit Optimalitätsbedingung

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} p \\ -\lambda^+ \end{pmatrix} = - \begin{pmatrix} -c \\ b \end{pmatrix}$$

ist erkennbar, dass sich $P_=$ als quadratisches Problem $QP_=$ darstellen lässt:

$$\begin{aligned} (QP_=) \quad \text{minimiere} \quad & \frac{1}{2} p^T \nabla^2_{xx} L(x, \lambda) p + \nabla f(x)^T p, \\ \text{sodass gilt} \quad & A(x) p = -g(x). \end{aligned}$$

x und λ werden mit der Lösung von $QP_=$ aktualisiert:

$$\begin{aligned} x \leftarrow x + p \quad & \text{und} \\ & \lambda \leftarrow \lambda^+, \end{aligned}$$

und man erhält eine iterative Lösung für das nichtlineare Problem $P_=$. Um Konvergenz zu erreichen, kann ein Parameter α eingeführt werden, der durch einen Line-Search-Algorithmus bestimmt wird, sodass sich der Wert einer bestimmten Gütefunktion ausreichend verringert. Eine solche Funktion besteht normalerweise aus einer Gewichtung für Erfüllbarkeit und Optimalität, z.B. kann die erweiterte Lagrange-Gütefunktion verwendet werden:

$$M_\mu(x) = f(x) - \lambda(x)^T g(x) + \frac{1}{2\mu} g(x)^T g(x)$$

mit $\mu > 0$.

In $QP_=$ werden die Nebenbedingungen von $P_=$ durch Linearisierung um x approximiert:

$$g_i(x) + \nabla g_i(x)^T p = 0, \quad i = 1, \ldots, m.$$

Für Ungleichheitsbedingungen $g_i(x) \geq 0$ kann diese Näherung verallgemeinert werden:

$$b_i(x) + \nabla g_i(x)^T p \geq 0.$$

Der SQP-Algorithmus liefert dann in jeder Iteration eine Vorhersage der aktiven Bedingungen in P abhängig von den aktiven Bedingungen in QP.

Zusammengefasst kann in einer Iteration für P mit x, λ, sodass $\nabla^2_{xx} L(x, \lambda) \succ 0$ gilt, das Problem QP anstatt P gelöst werden:

$$\begin{aligned} (QP) \quad \text{minimiere} \quad & \frac{1}{2} p^T \nabla^2_x x L(x, \lambda) p + \nabla f(x)^T p, \\ \text{sodass gilt} \quad & \nabla g_i(x)^T p \geq -g_i(x), \quad i \in I, \\ & \nabla g_i(x)^T p = -g_i(x), \quad i \in E. \end{aligned}$$

Kapitel 3

Stand der Technik

In diesem Kapitel soll der aktuelle Stand der Technik vorgestellt werden, indem bereits existierende Ansätze zur Analyse und Simulation von digitalen eingebetteten Hardware/Software-Systemen sowie deren Verwendung als virtuellen Prototypen und in Kombination mit domänenspezifischen Simulationsumgebungen diskutiert werden. Weiterhin werden verschiedene Ansätze zur Optimierungen der Energieeffizienz untersucht, die in Optimierungsprozesse während der Entwurfszeit und in Optimierungen zur Laufzeit untergliedert werden.

3.1 Simulation und Analyse eingebetteter Hardware/ Software-Systeme

Die Aspekte *Simulation* und *Analyse* werden in diesem Abschnitt gemeinsam diskutiert, da Simulationen oftmals zu Analysezwecken eingesetzt werden und bestehende Ansätze deshalb beide Aspekte zumeist in gegenseitiger Abhängigkeit betrachten.

Formale Analyse- und Bewertungsansätze basieren in der Regel auf einem Berechnungsmodell, in dem die Eigenschaften des zu analysierenden Systems semantisch repräsentiert sind. Eine umfassende Übersicht über allgemeine Berechnungsmodelle wird in [62] gegeben. Typischerweise werden formale analytische Ansätze zur Bewertung von Hardware/Software-Systemen und Echtzeitsystemen in *White-Box*-Ansätze und *Black-Box*-Ansätze unterschieden. Modellbasierte White-Box-Ansätze, wie z.B. in [5] [131] [29], zeichnen sich dadurch aus, dass das zugrunde liegende Modell eine direkte Verbindung mit der funktionalen Implementierung aufweist, was die Modellierung und Berücksichtigung komplexer Interaktionsmuster und Primitiven zur Synchronisation mit einschließt. Sie agieren damit in der Regel auf niedrigerer Abstraktionsebene als Black-Box-Ansätze [123] [48], deren Modelle hauptsächlich auf festgelegten Spezifikationen oder Verhaltensabschätzungen basieren, die zielgerichtet kombiniert werden. Dies ermöglicht zwar eine effizientere Analyse, abstrahiert aber meist von komplexen Interaktionen und gegenseitiger Beeinflussung. Beide Ansätze haben gemeinsam, dass sie wegen des übermäßigen Wachstums des Analyseraums bei komplexen Systemmodellen nur noch schwer anwendbar sind. Weiterhin berücksichtigen sie aus Komplexitätsgründen keine dynamischen Aspekte der beteiligten Hardware-

und Software-Komponenten, wie z.B. datenabhängige Schleifen und Bedingungen oder komplexe Hardware-Plattformen.

Um diese Beschränkungen wenigstens teilweise aufzuheben, existieren neben den gerade genannten formalen Ansätze, denen in der Regel ein mathematisches Modell zugrunde liegt, auch Analyseansätze, die auf einer ganzheitlichen Simulation des Systemmodells basieren [33]. Der größte Nachteil dieser Ansätze liegt in der Abdeckung der simulierten Testfälle. Durch diese Ungewissheit können speziell für Echtzeitsysteme mit harten Anforderungen nicht zwangsläufig alle relevanten Eigenschaften garantiert werden, da eventuell nicht alle Grenzfälle durch entsprechende Simulationsdurchläufe abgedeckt sind. Nichtsdestotrotz erlauben sie eine frühzeitige Evaluierung und Verifikation der modellierten Systeme während des Systementwurfs, indem sie zum Test bestimmter Anwendungsfälle benutzt werden. Sie unterstützen den Systementwurf durch die Simulation unterschiedlicher Konfigurationen mit dem Ziel der Exploration des zur Verfügung stehenden Entwurfsraums oder durch die Möglichkeit, frühzeitig im Entwurfsprozess mit der Entwicklung der auszuführenden Software starten zu können. Da simulationsbasierten Ansätzen ein ausführbares Modell zugrunde liegt, erlauben sie die frühzeitige Entwicklung der auf dem Zielsystem auszuführenden Software basierend auf einem virtuellen Modell der Zielplattform. Weiterhin erlauben sie die Berücksichtigung dynamischer Aspekte der simulierten Systeme, soweit diese im Simulationsmodell repräsentiert sind. Zudem werden Verfahren sowohl zur Abstraktion als auch zur Verfeinerung der verwendeten Modelle eingesetzt, wodurch eine Abwägung zwischen der Simulationsgeschwindigkeit und der Simulationsgenauigkeit abhängig der zum Ziel gesetzten Aspekte ermöglicht wird [2].

Die weitere Betrachtung simulativer Ansätze wird im Folgenden in zwei Abschnitte unterteilt. In Abschnitt 3.1.1 werden zunächst existierende Arbeiten zur Erstellung und Verwendung von Simulationsmodellen in Form von virtuellen Prototypen vorgestellt. Danach werden in Abschnitt 3.1.2 weiterführende Ansätze diskutiert, die eine Erweiterung der Sichtweise virtueller Prototypen bieten, indem Modelle unterschiedlichster Domänen in die Simulation integriert werden.

3.1.1 Generierung und Verwendung virtueller Prototypen

Virtuelle Plattformen bzw. virtuelle Prototypen werden als ausführbares Simulationsmodell für allgemeine Hardware/Software-Systeme bis hin zu SoC- und MPSoC-Architekturen [99] eingesetzt. Um dabei neben den funktionalen Eigenschaften des ständig wachsenden Anteils eingebetteter Software auch deren nicht-funktionale Eigenschaften (NFP) berücksichtigen zu können, existieren mehrere unterschiedliche Ansätze. Zur Ausführung von Quellcode, der bereits für die Zielarchitektur übersetzt ist, können Instruktionssatz-Simulatoren (ISS) in das Simulationsmodell eingebunden werden, welche die Befehle der Zielarchitektur auf Befehle des Simulationsrechners abbilden und diese somit ausführbar machen [9]. Zumeist werden hierfür C/C++-basierte ISS benutzt, da sie nahtloser in die Simulationssprache SystemC (siehe Abschnitt 2.3.1 im Grundlagenkapitel) eingebettet werden können [32]. Die Simulationsgeschwindigkeit wird durch die Verwendung der ISS jedoch erheblich beeinträchtigt, sodass diese

Ansätze in der Regel nicht zur Simulation komplexer Systeme und zur Systemexploration verwendet werden können. Auch kommerzielle Werkzeuge im Bereich virtueller Prototypen, wie z.B. [119], basieren zumeist auf der Interpretation von Befehlen, die für die Zielarchitektur übersetzt sind. Bei diesen Werkzeugen liegt der Fokus auf der Entwicklung der eingebetteten Software auf einem virtuellen Modell der späteren Zielplattform, sodass die Wiederverwendung dieser Software im Vordergrund steht und damit die Simulationsgeschwindigkeit eher zum Sekundärziel wird.

Für die Simulation virtueller Prototypen entwickelte Simulationssprachen wie SystemC und SpecC [26] ermöglichen die Annotation nicht-funktionaler Eigenschaften direkt im Simulationsmodell. Dieser Ansatz erlaubt im Gegensatz zu Simulationen unter Verwendung eines ISS eine deutlich höhere Simulationsgeschwindigkeit, bedarf jedoch einer teilweise komplexen Analyse der zu annotierenden Eigenschaften, um deren Genauigkeit so weit wie möglich zu garantieren. Erste Ansätze dieser Analysen auf Ebene von Binärcode und der Rückführung in das Simulationsmodell wurden in [112] vorgestellt. Erweiterungen hinsichtlich der Berücksichtigung von Optimierungen, die durch den Übersetzer (engl.: Compiler) durchgeführt werden, sind in [115] enthalten. Beide Arbeiten konzentrieren sich auf die Mikroarchitektur der Zielplattform, wie z.B. Instruktionen, Pipeline-Abarbeitung, und Sprungvorhersage, die Eigenschaften der Makroarchitektur, wie z.B. die Ausführung auf Mehrkern-Architekturen mit unterschiedlichen Betriebsmodi, bleiben weitestgehend unberücksichtigt. In [39] wird die Integration eines Modells eines Echtzeit-Betriebssystems (engl.: Real-Time Operating System (RTOS) in SpecC vorgestellt, in [100] die Integration eines RTOS in SystemC. Diese Arbeiten berücksichtigen lediglich das Zeitverhalten des simulierten Systems, weitere nicht-funktionale Eigenschaften wie die Leistungsaufnahme werden nicht beachtet. Außerdem nimmt die Simulationsperformanz durch die Integration der RTOS-Modelle stark ab. Für eine frühe Evaluierung des Systemverhaltens wäre eine abstrakte Repräsentation eines Betriebssystems ausreichend, die neben der Ablaufplanung auch die Aspekte Leistungsausnahme und Energiemanagement in der Ablaufplanung einschließen würde. In [38] wird ein Ansatz zur Simulation von NFP-annotierten Modellen unter Verwendung eines abstrakten Betriebssystemmodells vorgestellt, darin existiert jedoch keine energieeffiziente Ablaufplanung. Die Integration eines dynamischen Managements der Leistungsaufnahme wird in [117] und [134] präsentiert, die Leistungsaufnahme basiert aber auf Profilen und ist nicht im Simulationsmodell bzw. im SystemC-Quellcode annotiert, was eine dynamische bzw. datenabhängige Bewertung der Leistungsaufnahme zur Simulationszeit erschwert.

Neben der Verwendung virtueller Prototypen zur Evaluierung des Systemverhaltens ist auch die Erstellung bzw. Generierung dieser virtuellen Prototypen von großer Bedeutung. Da die Simulation eines Hardware/Software-Systems die Existenz eines entsprechenden Simulationsmodells voraussetzt, gibt es viele Bestrebungen und aktuelle Arbeiten in Richtung einer automatischen Generierung virtueller Prototypen. Diese Ansätze unterscheiden sich zumeist grundlegend anhand der für den Generierungsprozess verwendeten Eingangsmodelle. In [69] wird ein Ansatz vorgestellt, in dem virtuelle Prototypen in SystemC bzw. TLM-2.0 ausgehend von datenflussorientierten Modellen der Anwendung generiert werden können. Nach [42] kann

die Simulationsperformanz dieser virtuellen Prototypen erhöht werden, indem das Wissen über die Kommunikation bzw. Interaktion der zugrunde liegenden formalen datenflussorientierten Modelle auf die Simulationsmodelle übertragen und dadurch komplexe ereignisbasierte Interaktionen ersetzt werden.

3.1.2 Integration domänenspezifischer Simulationsumgebungen

Durch die Integration eingebetteter Hardware/Software-Systeme in immer komplexeren Gesamtsystemen und der daraus resultierenden zunehmenden Abhängigkeiten des eingebetteten Systems von dessen Umgebung, steigt auch die Notwendigkeit der Berücksichtigung dieser umgebenden Systeme in einer ganzheitlichen Simulation unter Verwendung virtueller Prototypen. Nach [23] existieren hierbei jedoch viele Herausforderungen. Zum einen müssen kontinuierliche physikalische Modelle auf diskrete, zum Teil ereignisbasierte Modelle abgebildet werden. Zum anderen steigt durch die rapide wachsende Komplexität solcher sogenannter Cyber-Physischer Systeme die Wahrscheinlichkeit des Vorkommens nicht-kompatibler Semantiken, was eine fehlerlose Interaktion verschiedener Domänen nahezu ausschließt. Als mögliche Lösungen werden ein plattformbasierter Entwurf [109] unter strikter Einhaltung einer eindeutigen Semantik und die Definition einer abstrakten Modellsemantik zur Integration verschiedener Berechnungsmodelle [27] [77] vorgeschlagen.

Mit [20] und [126] existieren Simulationsplattformen, die die Integration und wechselseitige Simulation domänenspezifischer Simulationsumgebungen wie MATLAB/-Simulink, ModelSim[1], Modelica[2]/Dymola[3] und teilweise auch C/C++-Code erlauben. Speziell für die Anwendungsdomäne Automobil wurde in [61] eine Schnittstelle[4] zur funktionalen Kopplung heterogener Simulationswerkzeuge entwickelt. In diesen Arbeiten werden jedoch virtuelle Prototypen eingebetteter Hardware/Software-Systeme und speziell SystemC als geeignetes Werkzeug zur Simulation von Hardware/Software-Systemen nicht berücksichtigt.

3.2 Optimierung der Energieeffizienz

Zur Betrachtung und Optimierung der Energieeffizienz in eingebetteten Hardware/Software-Systemen muss zunächst die Leistungsaufnahme der zugrunde liegenden Hardware-Plattform analysiert werden, da der Energieverbrauch durch das Integral der Leistungsaufnahme über die relevante Zeit der Aktivität ermittelt wird. Grundsätzlich können existierende Arbeiten zur Abschätzung der Leistungsaufnahme nach der Granularität der Analyse und der verwendeten Abstraktionsebene unterschieden werden. Zur Erhöhung der Genauigkeit wird oftmals eine Simulation der zugrunde liegenden Hardware-Plattform auf niedrigen Abstraktionsebenen durchgeführt, wie z.B. auf Ebene der einzelnen Gatter, auf Register-Transfer-Ebene oder auf Architekturebene

[1] Simulationsumgebung für Hardwarebeschreibungssprachen (HDL) von Mentor Graphics
[2] Objektorientierte Beschreibungssprache für physikalische Modelle (Modelica Association)
[3] Grafische Entwicklungsumgebungen für Modelica
[4] Functional Mockup Interface (FMI)

[136]. Da diese Simulationen aufgrund der niedrigen Abstraktionsebene und der dadurch inhärenten Komplexität jedoch auch eine niedrige Simulationsgeschwindigkeit aufweisen, sind sie auf komplexe und Software-intensive Systeme zumeist kaum anwendbar.

Abstraktere Modelle, die für die Abschätzungen der Leistungsaufnahme der auf einer bestimmten Zielarchitektur ausgeführten Software verwendet werden, basieren in der Regel auf den letztendlich ausgeführten Instruktionen [125]. Zur Betrachtung der Leistungsaufnahme von kompletten System-on-Chip-Architekturen wird in [41] eine instruktionsbasierte Abschätzung der Leistungsaufnahme für Peripherie-Komponenten durchgeführt. Weitere Arbeiten berücksichtigen ebenfalls die Leistungsaufnahme der Kommunikationsstruktur bzw. dedizierter Kommunikationskomponenten, wie z.B. die auf dem Chip vorhandenen Bussysteme [72]. Jedoch existieren hier nur Lösungen für separat ausgeführte Programme, sodass modernen Architekturen mit mehreren nebenläufigen Funktionalitäten nicht berücksichtigt werden.

Zur weiteren Erhöhung der Abstraktionsebene können zustandsbasierte Modelle zur Bewertung der Leistungsaufnahme von Gesamtsystemen verwendet werden. In [12] werden die zur Modellierung der Leistungsaufnahme verwendeten Zustandsautomaten (engl.: *Power State Machine (PSM)*) der einzelnen Hardware-Komponenten durch Produktbildung der enthaltenen Zustände zu einer einzigen PSM vereint, die die Leistungsaufnahme das Gesamtsystem repräsentiert. Durch eine formale symbolische Simulation können dann die minimale und maximale Leistungsausnahme des Systems analysiert werden. In [94] wird ein Ansatz vorgestellt, der die Ableitung von Modellen der Leistungsaufnahme auf verschiedenen Ebenen der Granularität erlaubt, z.B. von funktionalen Komponenten auf der Hardware-Plattform, über die Betrachtung von Instruktionen bis hin zur Berücksichtigung einzelner Pipeline-Stufen. Dadurch wird eine Abwägung zwischen der Genauigkeit des Modells und dessen Verwendung in schnellen Simulationen ermöglicht.

Eine Übersicht über die Modellierung und Abschätzung der Leistungsaufnahme, aber auch deren Optimierung in frühen Entwurfsphasen wird in [81] präsentiert. Im Entwurfsprozess von Hardware-Plattformen existieren sowohl Möglichkeiten zur Optimierung während der Synthese [17] als auch zur optimierten Platzierung der Architekturkomponenten [52]. Dabei müssen insbesondere die Auswirkungen der entwickelten Strategien zur Minimierung der Leistungsaufnahme auf das Verhalten der integrierten Schaltungen berücksichtigt werden [50] [83], da eine Reduzierung der Leistungsaufnahme oftmals mit Einbußen bei der Performanz gekoppelt ist. Diese Arbeiten befinden sich allerdings auf niedrigerer Abstraktionsebene und befassen sich deshalb nicht mit dynamischen Mechanismen zur Reduzierung der Leistungsaufnahme.

Weitergehende Arbeiten, die hier jedoch nicht näher beschrieben werden sollen, betrachten die Leistungsaufnahme nicht als Faktor der Minimierung des Energieverbrauchs, sondern setzen diese in Verbindung mit technologisch-bedingten Einflüssen, wie z.B. Prozessvariabilitäten, die zur Bewertung der Zuverlässigkeit des Systems eingesetzt werden. So soll z.B. darüber entschieden werden, ob bestimmte Ausführungszeiten von Anwendungen garantiert werden können [82].

Die steigende Komplexität der betrachteten Hardware/Software-Systeme und der

stetig wachsende Anteil eingebetteter Software bedingt auch bei der Optimierung des Energieverbrauchs eine Erhöhung der verwendeten Abstraktionsebene. Ein Überblick von Strategien zum Energiemanagement auf Systemebene, wie z.B. die dynamische Anpassung der Frequenz an die Auslastung des Systems und eine damit verbundene Reduzierung der Versorgungsspannung in CMOS-Schaltungen, ist in [11] und [59] enthalten. Hier werden allerdings nicht nur verschiedene Strategien diskutiert, sondern auch unterschiedliche Phasen des Einsatzes identifiziert. Ist das Verhalten des Systems vorhersagbar, können Strategien schon zur Entwurfszeit entwickelt werden, was einem Offline-Verfahren entspricht. Ist dies nicht der Fall, müssen Strategien implementiert werden, die auf den momentanen Eigenschaften des Systems reagieren und damit dynamisch zur Laufzeit eine Strategie ableiten. Dieses Vorgehen wird durch ein Online-Verfahren geregelt. In [104] erfolgt eine grundsätzliche Gegenüberstellung von Online- und Offline-Algorithmen. Diese Unterscheidung spiegelt sich auch in den nachfolgenden Abschnitten 3.2.1 und 3.2.2 wider, die jeweils existierende Arbeiten bezüglich der Energieoptimierung zur Entwurfszeit bzw. zur Laufzeit thematisieren.

3.2.1 Optimierung zur Entwurfszeit

Vergleichbar mit der Analyse der Leistungsaufnahme unterscheiden sich auch die Strategien zur Optimierung der Energieeffizienz in der jeweils verwendeten Abstraktionsebene. In [132] wird beispielsweise eine Optimierung bei der Transformation eines Verhaltensmodells in Quellcode, z.B. durch die Verwendung bestimmter Datentypen oder Instruktionen durchgeführt. Eine grundsätzliche Übersicht über Optimierungen auf höheren Ebenen, die Verfahren der Ablaufplanung und deren Einsatz zur Minimierung des Energieverbrauchs mit einschließen, wird in [18] dargestellt. [133] präsentiert einen Ansatz zur Entwicklung eines energieeffizienten Ablaufplans für datenflussorientierte Anwendungen. In [88] wird dagegen lediglich das Teilproblem der Auswahl eines energieeffizienten Betriebsmodus für eine gegebene Abbildung untersucht. In [110] wird eine Anpassung der Versorgungsspannung mittels DVS anhand von Profilen der Leistungsaufnahme einzelner Komponenten durchgeführt, jedoch können Ressourcen nicht komplett abgeschaltet werden, es erfolgt also keine Berücksichtigung von DPM-Strategien.

Zur Zuordnung von auszuführenden Tasks auf Betriebsmodi der zugrunde liegenden Verarbeitungsressourcen und deren Ausführungsreihenfolge werden oftmals mathematische Modelle benutzt, die als ganzzahliges lineares Optimierungsproblem (ILP) formuliert sind [43]. Im Allgemeinen sind ILP-Probleme jedoch aufgrund ihrer Komplexität ohne eine geeignete Abstraktion bzw. durch die Verwendung einer Heuristik schwer zu lösen. In [19] wird deshalb ausgehend von einer ILP-Formulierung die Transformation in ein lineares Optimierungsproblem vorgeschlagen, das zunächst keine ganzzahligen Lösungen benötigt. Um letztendlich einen eindeutige Zuordnung in Form einer ganzzahligen Lösung zu ermitteln, wird ein iteratives Verfahren angewandt, das durch einen Bin-Packing-Algorithmus bestimmte Zuordnung fixiert und anschließend basierend auf einer Heuristik Umverteilungen vornimmt. Dieses Verfahren liefert aber aufgrund der Heuristik in der Regel kein optimales Ergebnis. In [114] wird ein ähnliches

Verfahren verwendet, zur Modellierung der Anwendung wird hier ein Task-Graph mit annotierten Wahrscheinlichkeiten benutzt. Die Tasks besitzen keine Ausführungszeiten, sondern es wird die Auswirkung der Tasks auf die Auslastung der zugrunde liegenden Ressourcen verwendet. Die Betrachtung von Auslastungen anstatt Startzeitpunkten und Ausführungszeiten von Tasks verhindert allerdings eine genaue und energieeffiziente Steuerung der Ablaufplanung. So können z.B. keine Optimierungen über Periodengrenzen hinweg vorgenommen werden, um ebenso die Wechsel zwischen unterschiedlichen Betriebsmodi zu minimieren. Generell fokussieren Ansätze, die Task-Graphen zur Anwendungsspezifikation verwenden, auf eine sequentielle Abfolge der Tasks, sodass eine freie Ablaufplanung oder eine etwaige Parallelisierung von Tasks nicht möglich ist. In [108] wird deswegen eine energieeffiziente Ablaufplanung untersucht, die den Speed-Up durch eine mögliche parallele Ausführung der Tasks auf mehreren Berechnungskernen einbezieht. Allerdings sind dabei lediglich solche Tasks zugelassen, die es erlauben, dass sich die Menge der an dem Task parallel arbeitenden Ressourcen über die Zeit ändern kann.

Auf Analysen basierende Optimierungsverfahren abstrahieren weitestgehend von der Implementierung der realen Hardware-Plattform und berücksichtigen deshalb keine daraus resultierenden Einschränkungen. In [49] und [128] werden deshalb Ansätze für eine gemeinsamen Betrachtung von DVFS-Strategien und der notwendigen Voraussetzungen auf Chip-Ebene bzw. auf Ebene der Verbindungsstruktur in Form von Voltage/Frequency Islands vorgestellt.

Durch eine ganzheitliche Betrachtung der Hardware-Plattform ergeben sich zusätzliche Optimierungsmöglichkeiten. So wird in [79] der Einfluss von vorhandenen Caches auf den Energieverbrauch des Gesamtsystems untersucht. In [101] wird die Nutzung von Cache-Effekten dahingehend betrachtet, dass eine Abwägung zwischen der durch die Caches möglichen Steigerung der Ausführungsgeschwindigkeit und deren Energieverbrauch erfolgt. Weiterhin existieren Ansätze [129], welche die Kommunikation zwischen den einzelnen Tasks als Grundlage für eine Anpassung der Versorgungsspannung in den Verarbeitungsressourcen verwenden. In [24] werden zusätzlich benötigte Komponenten in die Minimierung des Energieverbrauchs einbezogen, die gleichzeitig mit der Ausführung eines Tasks aktiv sein müssen. Durch einen Vergleich zwischen einer DVS-Strategie auf der Verarbeitungsressource, was eine längere Ausführungszeit des Tasks und damit eine längere Aktivierung der zusätzlich Komponente zur Folge hätte, und einer Abschaltstrategie für die Komponente bei einer schnelleren Ausführung des Tasks und kürzerer Aktivierung der Komponente wird ein plattformabhängiger Break-Even-Zeitpunkt identifiziert, der für die Auswahl einer Strategie zur Optimierung der Energieeffizienz verwendet wird.

In [60] wird schließlich ein orthogonaler Ansatz zu den bisherigen Optimierungen gewählt, indem nicht der Energieverbrauch minimiert, sondern die Performanz unter der Berücksichtigung eines gegebenen Budgets der Leistungsaufnahme maximiert wird.

3.2.2 Optimierung zur Laufzeit

Ein entscheidendes Kriterium für die Anwendbarkeit von Optimierungen zur Laufzeit ist zum einen die Beobachtbarkeit des Systems. Zum anderen spielt der für die Optimierung benötigte Aufwand eine wichtige Rolle, da dieser die Ausführung der eigentlichen Funktionalität beeinflusst.

In [58] wird als Metrik zur Bewertung von Online-Algorithmen das Verhältnis der Kosten des Online-Algorithmus zu den besten Ergebnissen, die durch ein Offline-Verfahren zu erzielen wären, vorgeschlagen. Dazu werden verschiedene Strategien miteinander verglichen und ein Energiemanagement vorgeschlagen, das auf Wahrscheinlichkeiten basiert. Zusätzlich wird zwischen adaptiven und nicht-adaptiven Verfahren zur Laufzeit unterschieden. Nicht-adaptive Verfahren reagieren zwar auf die dynamischen Eigenschaften des Systems, tun dies aber immer nach einer vorgegebenen Strategie. Ein Beispiel hierfür wäre ein System, das auf eine Leerlaufphase wartet und sich allein aus dieser Tatsache heraus in einen anderen Betriebsmodus versetzt. Adaptive Verfahren benutzen das Wissen über die Vergangenheit und passen sich dementsprechend an. So wird im vorherigen Beispiel versucht, die Zeit vorherzusagen, die das System in der Leerlaufphase sein wird, um basierend darauf eine Entscheidung zu treffen, ob der Betriebsmodus gewechselt wird oder nicht.

In [25] wird ein Verfahren zum ständigen Lernen während der Laufzeit für die Steuerung des Energiemanagements benutzt, allerdings muss nach jedem Durchlauf eine Funktion zur Auswertung der erfolgten Reaktion durchgeführt werden, was einen nicht zu vernachlässigenden Aufwand hervorruft. Ein ähnlicher Ansatz wird in [105] verfolgt. Darin werden zunächst in einem Offline-Verfahren die Abschaltzeiten von Verarbeitungsressourcen basierend auf Markov-Prozessen optimiert. Die zur Auswahl der Abschaltstrategie benutzen Parameter werden dann zur Laufzeit angepasst. Jedoch ist das Ergebnis dieses Vorgehens stark von den tatsächlichen dynamischen Eigenschaften des Systems abhängig und kann nur dann sinnvoll verwendet werden, wenn z.B. statistische Annahmen getroffen werden können.

Weiterhin existieren mehrere Ansätze zur Optimierung, die auf der Auswertung bestimmter Eigenschaften während der Laufzeit basieren. In [8] werden anhand der Online-Auswertung der Ausführungszeiten von Tasks die notwendige Versorgungsspannung und Vorspannung aus einer zuvor angelegten Datenstruktur ausgewählt und entsprechend angepasst. Dabei ist aber keine Veränderung des Ablaufs von Tasks möglich, sodass eine eventuell vorhandene Zeitspanne (Slack), die zwar im Moment nicht für eine DVFS-Strategie genutzt werden kann, aber prinzipiell Potential für eine Optimierung bietet, nicht propagiert werden kann. In [40] wird eine dynamische Abschätzung der maximalen Ausführungszeit zur Laufzeit vorgenommen und das Energiemanagement anhand eines rudimentären Modells des Energieverbrauchs angepasst. Ein Ansatz zur dynamischen Auswertung von aktuellen Eigenschaften wird auch in [6] vorgestellt. Darin werden Versorgungsspannung und Frequenz anhand einer Rückkopplung von Warteschlangen an den Ausgangskanälen so angepasst, dass diese Warteschlangen einen zuvor definierten Füllgrad erreichen bzw. nicht überschreiten. Da dieser Ansatz auf einer Rückkopplung basiert, kann er jedoch nur eingesetzt werden,

wenn die Latenz der Daten eine untergeordnete Rolle spielt.

In [21] werden für die Anwendung benötigte Speicherzugriffe aus Speicherbausteinen, die sich außerhalb des Chips befinden, durch eine Monitoreinheit beobachtet und mit den Ausführungszeiten der Anwendung auf der Berechnungseinheit in ein Verhältnis gesetzt. Die DVFS-Strategie wird anhand der Abweichung dieses Verhältnisses angepasst. Dieses Online-Verfahren kann somit nicht in Systemen mit harten Echtzeitanforderungen eingesetzt werden, da es lediglich nach der Feststellung eingetretener Abweichungen nachregelt.

3.3 Zusammenfassung der offenen Probleme

Insgesamt kann festgestellt werden, dass bisher lediglich Einzellösungen existieren, um eine Optimierung der Energieeffizienz in digitalen eingebetteten Hardware/Software-Systemen zu realisieren. Ansätze, die auf statischen Mechanismen zur Reduzierung der Leistungsaufnahme basieren, werden zumeist technologieabhängig und deswegen nur auf niedrigen Abstraktionsebenen angewandt. Damit fehlt eine applikationsspezifische Steuerung des Energiemanagements. Existierende Ansätze auf höheren Abstraktionsebenen basieren zumeist auf stark vereinfachten Modellen der Leistungsaufnahme oder der Repräsentation der Funktionalität. Sie sind somit nur sehr eingeschränkt auf reale Hardware-Plattformen und Anwendungen übertragbar. Zumeist werden auch nur entweder DVFS oder DPM zur Reduzierung der Leistungsaufnahme angewandt, eine Abwägung beider Techniken findet damit nicht statt. Zusätzlich konzentrieren sich bisherige Arbeiten entweder auf Optimierungsansätze, die in abstrakter Form zur Entwurfszeit angewandt werden, oder auf solche, die ein Energiemanagement zur Laufzeit durchführen. Die zu erzielenden Ergebnisse sind dadurch erheblich eingeschränkt. Eine Kombination, wie sie in der vorliegenden Arbeit adressiert wird, erhöht das Optimierungspotential, indem eine auf statischen Eigenschaften basierende Optimierung, sowie eine Analyse des dynamischen Verhaltens zur Entwurfszeit durchgeführt, sodass auch zur Laufzeit eine Steigerung der Energieeffizienz erfolgen kann.

Auch existieren bisher keine Ansätze zur Integration eines Energiemanagement in ausführbaren Modellen auf Systemebene. Dies ist allerdings dringend notwendig, um eine frühzeitige Abschätzung des Energieverbrauchs durchführen zu können und Auswirkungen von strategischen Entscheidungen bei der Steigerung der Energieeffizienz integrieren zu können. Weiterhin gibt es bisher keine Arbeiten, die sich mit der automatischen Generierung einer Ausführungsplattform ausgehend von einer abstrakten Beschreibung der Funktionalität beschäftigen, um diese unter Berücksichtigung des Zeitverhaltens auf der Zielarchitektur und eines Energiemanagements ausführbar zu machen, ohne dass ein ausführbares Modell manuell und dadurch kostenintensiv implementiert werden muss. Dadurch können weder geltende Anforderungen während des Energiemanagements überprüft, noch das dynamische Verhalten anhand eines virtuellen Prototyps analysiert werden. Der in dieser Arbeit vorgestellte Ansatz zur automatischen Generierung einer Ausführungsplattform vereint diese Eigenschaften und erlaubt eine frühzeitige Analyse des funktionalen und nicht-funktionalen Verhaltens

eingebetteter Hardware/Software-Systeme auf hoher Abstraktionsebene.

Kapitel 4

Konzept zur Optimierung der Energieeffizienz auf Systemebene

Die typischen Einsatzmöglichkeiten von Rechensystemen haben sich in naher Vergangenheit grundsätzlich gewandelt. Wohingegen noch vor einigen Jahren leistungsfähige Hardware/Software-Systeme hauptsächlich auf stationäre Rechensysteme beschränkt waren, haben sich durch die Miniaturisierung der Technologie zusätzliche Einsatzmöglichkeiten im mobilen bzw. batteriebetriebenen Bereich ergeben – selbst dann wenn ein hohes Maß an Performanz und Funktionalität gefordert ist. Die Reichweite geht dabei von der Unterhaltungselektronik über tragbare Computersysteme bis hin zu mobilen Anwendungen in autonomen Systemen und der Fahrzeugtechnik. Aus diesem Grund hat sich mittlerweile die Maximierung der Energieeffizienz unter Berücksichtigung der an das System gestellten Anforderungen zum primären Optimierungskriterium beim Entwurf dieser Systeme entwickelt. Dem allgemeinen Trend in nahezu allen Anwendungsdomänen folgend entsteht ein bedeutender Teil der zukünftigen Funktionalität in eingebetteten Hardware/Software-Systemen nicht mehr nur durch Realisierung einzelner Anwendungen, sondern durch eine verstärkte Interaktion dieser Anwendungen, was willkürlich zu Systemen führt, die aus vielen Subsystemen aufgebaut sein können. Dies bedeutet eine mitunter drastische Erhöhung der Komplexität in vielen Bereichen, besonders aber bei der Berechnung und Verarbeitung der Daten, sowie dem Kommunikationsverhalten, was wiederum neue Anforderungen an eine zugrunde liegende Architektur bzw. Hardware-Plattform stellt. Parallel dazu verlangt der enorme globale Konkurrenzdruck immer kürzere Entwicklungszeiten und vor allem sinkende Entwicklungskosten. Dies stellt Entwickler und Ingenieure vor neue Herausforderungen, insbesondere bei der Analyse und Überprüfung funktionaler und nicht-funktionaler Eigenschaften des Gesamtsystems, wie z.B. das Zeitverhalten und den Energieverbrauch. Die Kosten für notwendige Änderungen im Systementwurf steigen dramatisch, je größer der schon erreichte Fortschritt innerhalb des Entwicklungsprozesses des Systems ist, weshalb Entscheidungen bezüglich der Systemarchitektur möglichst frühzeitig im Entwurfsprozess getroffen werden müssen.

Zur Realisierung möglichst kurzer bzw. kostengünstiger Entwicklungszyklen und hochqualitativen Endprodukten ist daher sowohl eine frühzeitige abstrakte Analyse, als auch eine frühe Verifikation und Validierung des zu entwickelnden Systems unerläss-

lich. Hinsichtlich der Überprüfung funktionaler und nicht-funktionaler Eigenschaften gewinnt der Aspekt der *ganzheitlichen Simulation* an immer größerer Bedeutung. Diese verkürzt Entwicklungszyklen und erlaubt weiterhin ein frühzeitiges Evaluieren von Entwicklungsschritten zur Prävention von Fehlentwicklungen innerhalb des Entwurfs. Mithilfe der Simulation können zum einen Eigenschaften, wie das Zeitverhalten und der Leistungsverbrauch des modellierten Hardware/Software-Systems, auch bei komplexen Systemen ermittelt werden. Zum anderen können aber auch Entscheidungen, die Ergebnis eines dedizierten Prozesses zur Optimierung der Energieeffizienz sind, überprüft und deren Auswirkungen sichtbar gemacht werden.

Eine grundlegende Anforderung ist die frühzeitige und kostengünstige Verfügbarkeit von Modellen zur Analyse und Simulation, die das zu entwickelnde System möglichst genau abbilden. Ein viel versprechender Ansatz hierfür ist die Anwendung eines Entwurfsablauf, der durch geeignete Abstraktion auf die wesentlichen Eigenschaften fokussiert und in einen modellbasierten Entwicklungsprozesses integriert ist. Für die Anwendung eines modellbasierten Entwurfs- und Entwicklungsprozesses gibt es mehrere wichtige Gründe:

- Modelle bieten verschiedene Sichtweisen auf das dargestellte System. Komplexe Systeme können dadurch abstrahiert werden, wodurch deren Komplexität innerhalb der verwendeten Modelle erheblich reduziert wird.

- Modelle können einen entscheidenden Beitrag zur Beschleunigung des Entwicklungsprozesses liefern, indem sie als ein ausführbares Referenzmodell interpretiert werden können, das zur Simulation und damit zur frühzeitigen Verifizierung bestimmter Eigenschaften herangezogen werden kann.

- Modelle können zur Parallelisierung von Entwicklungsphasen verwendet werden, indem Teile der Software auf simulierter bzw. virtueller Hardware entwickelt und getestet werden können, bevor ein realer Prototyp der Hardware existiert.

- Ausführbare Modelle erlauben eine schnelle Konfiguration des modellierten Systems und ermöglichen deshalb die Simulation vieler unterschiedlicher Testszenarien und Explorationsparameter in kurzer Zeit.

- Modelle können auf die jeweils wesentlichen Eigenschaften des Systems beschränkt werden, was Analyse- und Optimierungsprozesse auf die adressierten Ziele fokussieren lässt.

Insbesondere der letzte Punkt ermöglicht *Analyse- und Optimierungsprozesse* auf hohen Abstraktionsebenen, die neben einer frühzeitigen Anwendbarkeit eine erhebliche Reduzierung der Analysekomplexität bei einem gleichzeitig hohen Optimierungspotential versprechen.

Es sollen zunächst die Voraussetzungen und Herausforderungen der in dieser Arbeit adressierten Analyse und Optimierung der Energieeffizienz eingebetteter Hardware/ Software-Systeme unter Berücksichtigung einer spezifizierten Hardware-Plattform und gegebener Anforderungen an die Funktionalität und die Performanz des Systems

dargestellt werden. Anschließend wird der dafür vorgeschlagene Lösungsweg skizziert, wobei für genauere Details auf die Kapitel 5 und 6 verwiesen wird.

Bezüglich der in dieser Arbeit verwendeten Definitionen für die Energieeffizienz eines Hardware/Software-Systems und deren Optimierung sei an dieser Stelle auf Abschnitt 2.1 verwiesen.

4.1 Voraussetzungen und Herausforderungen

In diesem Abschnitt werden die Herausforderungen beim Einsatz aktueller Architekturen im Bereich eingebetteter Hardware/Software-Systeme – verstärkt unter Verwendung mehrerer Berechnungseinheiten – untersucht. Weiterhin werden grundlegende Performanzanforderungen von eingebetteten Systemen beim Einsatz in Echtzeitsystemen erörtert. Des Weiteren werden verschiedene Mechanismen zur Reduzierung der Leistungsaufnahme sowie die zugrunde liegende Powermodelle unter Berücksichtigung dieser Performanzanforderungen bewertet. Diese Mechanismen werden zudem hinsichtlich ihres Einsatzes während des Systementwurfs bzw. während der Ausführungszeit der Funktionalität untersucht, was Auswirkungen auf die grundsätzlichen Entscheidungen bei der Optimierung der Energieeffizienz hat.

4.1.1 Einsatz von Mehrkern-Architekturen

Der Einsatz mehrerer Kerne in eingebetteten Hardware/Software-Systemen ist eine notwendige Reaktion auf die Tatsachen, dass immer komplexere Aufgaben erfüllt werden sollen, es aber technologisch bedingt eine obere Schranke für die Leistungsfähigkeit einzelner Kerne gibt. Nicht zuletzt liegt diese Beschränkung an der stark steigenden Leistungsaufnahme und der damit verbundenen Temperaturentwicklung. Die dadurch entstehende Grenze wird oftmals auch als *Power-Wall* bezeichnet. Weiterhin sind Mehrkern-Architekturen eine Lösung für Probleme, die sich dadurch ergeben, dass der Parallelismus auf Instruktionsebene weitestgehend ausgeschöpft ist (*ILP-Wall*) und die Speicherlatenz sich in den letzten Jahren nicht mehr signifikant verbessert hat (*Memory-Wall*).

Die Verwendung von Mehrkern-Architekturen bedeutet für Software- und Hardwareentwicklern aber auch, dass neue Denkweisen, Methoden und Paradigmen entwickelt werden müssen, die dem Wissen und der Intuition der ursprünglich sequentiellen Herangehensweise zum Teil widersprechen, zumindest jedoch ungewohnt erscheinen. Dieser notwendige Paradigmenwechsel umfasst sowohl die verwendeten Berechnungsmodelle als auch die Programmiermodelle und Entwicklungsprozesse, anhand deren die zukünftige Funktionalität entwickelt wird.

Die wahrscheinlich größte Herausforderung liegt jedoch in der effizienten Nutzung der zur Verfügung stehenden Ressourcen. Bei der Verwendung von Mehrkern-Architekturen treffen Entwickler und Systemdesigner somit zwangsläufig auf das Abbildungsproblem. Unter dem Problem der Abbildung von Funktionalität auf die ausführende Hardware-Plattform versteht man die Suche nach einer möglichst optima-

len Abbildung von Berechnungsaufgaben auf Rechenknoten unter Berücksichtigung verschiedener möglicher Optimierungsziele [124]. In klassischer Sichtweise ist dies primär die Performanz, also die Geschwindigkeit des Programmablaufs, oder aber der Kommunikationsaufwand, welcher die Performanz in vielen Fällen maßgeblich beeinflusst. Dadurch sollen insbesondere in eingebetteten Architekturen die Einhaltung der an das System gestellten Anforderungen garantiert werden. In jüngster Zeit ist jedoch die Optimierung der Energieeffizienz immer weiter in den Vordergrund gerückt, was mit dem verstärkten Einsatz eingebetteter Hardware/Software-Systeme in mobilen Anwendungen begründet ist.

Die Abbildung von Funktionalität auf ausführende Einheiten ist naturgemäß von der Anwendung selbst, den Eingaben, dem Zustand der Anwendung, der zugrunde liegenden Architektur, dem verwaltenden Betriebssystems und der Umgebung abhängig. So kann es vorkommen, dass eine Abbildung, die zu einem Zeitpunkt optimal war, durch veränderte Bedingungen zu einem anderen Zeitpunkt unbrauchbar wird. In eingebetteten Systemen besteht jedoch der Vorteil, dass die Umgebung bzw. das umgebende System weitestgehend bekannt sind, die Eingaben häufig eingeschränkter und vorhersagbarer sind als im allgemeinen Fall und teilweise genaue Vorhersagen über das Berechnungs- und Kommunikationsverhalten der einzelnen Komponenten möglich sind.

4.1.2 Geeignetes Berechnungsmodell

Eine wichtige Anforderung an ein geeignetes Berechnungsmodell als Grundlage zur Analyse des Verhaltens eingebetteter Hardware/Software-Systeme ist die Eigenschaft, sowohl für die Modellierung einer sequentiellen als auch für eine mögliche parallele Ausführung der Funktionalität geeignet zu sein. Dies trifft speziell dann zu, wenn der durch die Parallelität zusätzlich gewonnene Freiheitsgrad zur Optimierung der Energieeffizienz genutzt werden soll. Insbesondere wird dies durch die Verwendung von Mehrkern-Architekturen als zugrunde liegende Hardware-Plattform möglich, wie sie in Abschnitt 4.1.1 bereits vorgestellt wurde.

Eine datenflussorientierte Modellierungsmethodik bietet die Möglichkeit, eine mögliche parallele Ausführung der Funktionalität abzubilden, da sie auf die Darstellung der gegenseitigen Abhängigkeiten fokussiert und deshalb inhärent auch die maximale parallele Sicht auf die Funktionalität eines Hardware/Software-System bietet. In Abschnitt 2.4.3 des Grundlagenkapitels wurden bereits die *synchronen Datenflussmodelle (SDF)* als Spezialisierung allgemeiner Datenflussmodelle mit Zeitverhalten vorgestellt. Durch ihre Semantik erlauben SDF-Modelle die Beschreibung des Systemverhaltens mittels der Berechnung eines periodischen Zyklus, sodass die nebenläufige Ausführung einer modellierten Funktionalität auf die Durchführung von Transaktionen im Modell und auf abgeschlossene und periodische Iterationen abgebildet werden kann. Im Hinblick auf eine Optimierung der Energieeffizienz muss dieses periodische Verhalten mit den an das System gestellten Anforderungen an dessen Leistungsfähigkeit verbunden werden. Die in dieser Arbeit verwendeten Performanzanforderungen werden im nachfolgenden Abschnitt 4.1.3 definiert.

4.1.3 Performanzanforderungen in Echtzeitsystemen

Eingebettete Hardware/Software-Systeme besitzen im Allgemeinen hohe Anforderungen an die *Performanz* des betrachteten Systems. Diese können in harte und weiche *Echtzeitanforderungen* unterteilt werden. Harte Echtzeitanforderungen müssen vom System immer und in vollem Umfang eingehalten werden, wohingegen weiche Echtzeitanforderungen eine abgeschwächte Form darstellen und möglichst eingehalten werden sollten. Deswegen müssen sie in der Regel nur zu einem bestimmten Prozentsatz garantiert werden. Ohne Beschränkung der Allgemeinheit werden in dieser Arbeit aufgrund der anvisierten Anwendungsdomänen harte Echtzeitanforderungen gefordert. Diese können in der Regel auch konkreter formuliert werden.

Im Folgenden soll der *logische Durchsatz* und die *maximale Periode* eines Hardware/Software-Systems im Gegensatz zu Standarddefinitionen bezüglich Durchsatz und Periode definiert werden. Zu beachten ist hierbei, dass es sich bei einem Hardware/Software-System auch um ein hierarchisch kompositioniertes System handeln kann, d.h. jedes System kann Teilsystem eines anderen Systems sein. Die Definition des logischen Durchsatzes kann somit auch für jedes einzelne Teilsystem angewandt werden.

Definition 4.1 (Logischer Durchsatz)
Der logische Durchsatz eines Hardware/Software-Systems wird definiert durch den Quotienten aus der Menge an Datenpaketen, die das System verlassen und den für die Berechnung der Datenpakete benötigten Zeitschritten.

$$\lambda = \frac{\#\, Datenpakete}{Zeitabschnitt} \tag{4.1}$$

Der logische Durchsatz wird bei dessen Verwendung als Anforderung an die Leistungsfähigkeit eines Systems meist auf ein einziges Datenpaket normiert.

Definition 4.2 (Maximale Periode)
Die maximale Periode eines Hardware/Software-Systems bestimmt die Zeit, die maximal vergehen darf, damit das System die gestellten Anforderungen noch einhalten kann. Es wird deshalb als Kehrwert des logischen Durchsatzes mit $1/\lambda$ *definiert.*

Beide Performanzanforderungen spielen in eingebetteten Hardware/Software-Systemen ein maßgebliche Rolle. Sie entscheiden unter anderem darüber, ob ein System eine gestellte Aufgabe, z.B. in Form einer sicherheitskritischen Funktionalität, unter Einhaltung aller geltenden Bedingungen erfüllen kann oder nicht.

4.1.4 Anwendung von Low-Power-Strategien

Wie schon in Kapitel 1 dargestellt, haben Optimierungsstrategien zur Steigerung der Energieeffizienz in der Regel ein größeres Potential, je höher die Abstraktionsebene ist, in der sie angewandt werden. Techniken auf niedriger Abstraktionsebene, wie z.B. Power- oder Clock-Gating, werden beim Entwurf der entsprechenden Hardware-Schaltungen eingesetzt und sind somit bereits in der zugrunde liegenden Hardware-Plattform implementiert. Da in dieser Arbeit ein Ansatz zur Optimierung auf Systemebene

adressiert wird, sollen hier ausschließlich *dynamische Mechanismen* zur Anwendung kommen, insbesondere die bereits im Abschnitt 2.8 grundlegend erklärten Techniken *Dynamic Voltage and Frequency Scaling (DVFS)* und *Dynamic Power Management (DPM)*.

Notwendige Voraussetzung für die Anwendung beider Techniken ist, dass eine Aufgabe vor Ende seiner maximalen Periode bzw. *Deadline* beendet werden kann. Sie stehen damit in direktem Bezug zu den Performanzeigenschaften, die für eingebettete Hardware/Software-Systeme jeweils gefordert werden –insbesondere für diejenigen, die in Echtzeitsystemen eingesetzt werden. Die dieser Arbeit zugrunde liegenden Performanzanforderungen werden in Abschnitt 4.1.3 definiert.

Die zentrale Herausforderung bei der Anwendung dieser Strategien zur Reduzierung der Leistungsaufnahme ist die situationsabhängige Entscheidung, ob innerhalb eines bestimmten Zeitfensters eine Technik angewandt werden sollte und welche vorteilhafter in Bezug auf die Energieeffizienz ist. Prinzipiell wird durch die Anwendung einer DVFS-Strategie die Leistungsaufnahme verringert, die für die Ausführung einer Aufgabe benötigte Zeit allerdings erhöht. Da die Energieeffizienz sowohl von der Leistungsaufnahme als auch von der Ausführungszeit abhängt, besteht hierbei ein bestimmter Zielkonflikt. Durch DPM werden einzelne Teile der Hardware-Plattform in einen Ruhezustand versetzt, in dem die Leistungsaufnahme reduziert ist. Der Wechsel zwischen Betriebszustand und Ruhezustand ist jedoch mit einem Mehraufwand an Zeit- und Leistungsverbrauch verbunden, wodurch eine Abwägung getroffen werden muss, ob ein Wechsel überhaupt sinnvoll ist. Eine grundsätzliche Herangehensweise bezüglich dieser Abwägung wurde in Abschnitt 2.7.1 vorgestellt.

In folgenden Abschnitt 4.1.5 werden Modelle diskutiert, um die Leistungsaufnahme eines Hardware/Software-Systems innerhalb eines Analyseprozesses bzw. einer Simulation abbilden zu können.

4.1.5 Abbildung der Leistungsaufnahme

Zur Integration der Leistungsaufnahme in modellbasierten Analyseprozessen müssen zwingend geeignete Modelle existieren, die die Leistungsaufnahme der zugrunde liegende Hardware-Plattform repräsentieren. Ein grundlegender Unterschied dieser Leistungsmodelle liegt in deren abgebildeter Abstraktionsebene bzw. wie sie zur Berechnung einer Gesamtleistungsaufnahme und eines Energieverbrauchs für die Verrichtung einer spezifizierten Arbeit während einer bestimmten Laufzeit verwendet werden.

Ein Leistungsmodell auf höherer Abstraktionsebene ist das *zustandsbasierte Leistungsmodell*, das in Abschnitt 2.7.1 vorgestellt wird. Darin wird auch ein geeignetes Modell zur Darstellung der Leistungsaufnahme beschrieben. Dieses zeichnet sich im Wesentlichen dadurch aus, dass die Leistungsaufnahme eines Hardware/Software-Systems sich zwar ändern kann, über eine bestimmte Laufzeit aber als gleichverteilt angenommen werden kann.

Ein weiteres Leistungsmodell, das auf niedrigerer Abstraktionsebene verwendet wird und deswegen detailliertere Informationen benötigt, ist das *instruktionsbasierte Leistungsmodell*. Im Gegensatz zum zustandsbasierten Leistungsmodell wird dabei die

Leistungsaufnahme nicht als zeitweise konstant angesehen, sondern basiert auf einer Charakterisierung der Leistungsaufnahme der auf der Zielarchitektur ausgeführten Instruktionen. Dieses Leistungsmodell wird in Abschnitt 2.7.2 näher beschrieben und Erweiterungen dazu vorgestellt.

Aufgrund der niedrigeren Abstraktionsebene kann ein instruktionsbasierte Leistungsmodell effektiv nur für einen simulationsbasierten Ansatz angewandt werden – natürlich unter der Voraussetzung, dass die ausgeführten Instruktionen bekannt sind. Zur Beherrschung der Komplexität wird jedoch in modellbasierten Analyse- und Optimierungsprozessen zumeist von der Funktionalität und damit den ausgeführten Instruktionen abstrahiert, weshalb sich hier die Anwendung eines zustandsbasierten Leistungsmodells anbietet. Dieses Leistungsmodell ist zur Betrachtung der Energie bzw. des Energieverbrauchs weitestgehend ausreichend, wohingegen sich akkurate Aussagen über die Temperaturverteilung und Zuverlässigkeit kaum treffen lassen.

Da sowohl die benötigte Zeit zur Ausführung einer bestimmten Funktionalität, als auch deren Leistungsaufnahme meist auf einem normierten Betriebszustand basiert, können diese nicht-funktionalen Eigenschaften (NFP) unter Verwendung der Formeln zur Abschätzung der Leistungsaufnahme und der Schaltgeschwindigkeit auf andere Betriebszustände übertragen werden.

Die relative Berechnung der Ausführungszeit und der Leistungsaufnahme basiert auf der vereinfachten Formel zur Berechnung der dynamischen Leistung (vgl. Gleichung 2.1 im Grundlagenabschnitt)

$$P_{dyn} = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f \tag{4.2}$$

und der Abschätzung für die Schaltgeschwindigkeit moderner integrierter Schaltungen durch Gleichung 2.2:

$$f = \frac{(V_{dd} - Vth)^2}{V_{dd}}. \tag{4.3}$$

Mit konstanten Werten für die Schaltaktivität α der integrierten Schaltungen während der Ausführung der gleichen Funktionalität, für die zu schaltende Kapazität C_L und für die durch die verwendete Technologie definierte Schwellspannung V_{th} können durch Anwendung dieser beiden Formeln sowohl die Leistungsaufnahme P als auch die Frequenz f für die Ausführungsgeschwindigkeit zweier Betriebsmodi bzw. Power-Modi PM_1 und PM_2 mit den jeweils geltenden Versorgungsspannungen V_1 und V_2 in Relation zueinander gebracht werden:

$$P_2 = \left(\frac{V_2}{V_1}\right)^2 \cdot P_1 \tag{4.4}$$

$$f_2 = \frac{\left(\frac{(V_2 - V_{th})^2}{V_2}\right)}{\left(\frac{(V_1 - V_{th})^2}{V_1}\right)} \cdot f_1 \tag{4.5}$$

Durch diese relative Betrachtung der Ausführungszeit und der Leistungsaufnahme können auch zustandsbasierte und instruktionsbasierte Leistungsmodelle zusammengeführt werden. Die Genauigkeit der Leistungsaufnahme auf Instruktionsebene wird dadurch mit der Möglichkeit kombiniert, die zugrunde liegende Hardware-Plattform bzw. deren Modell in verschiedene Betriebszustände zu versetzen.

4.2 Vorgeschlagener Lösungsansatz

Nach der Darstellung der Voraussetzungen und Herausforderungen der in dieser Arbeit adressierten Analyse und Optimierung der Energieeffizienz eingebetteter Hardware/ Software-Systeme soll in diesem Abschnitt der entwickelte Lösungsansatz beschrieben werden. Dazu werden zunächst jedoch einige Begriffe definiert, die bei der Modellbildung als Grundlage des Analyse- und Optimierungsprozesses verwendet werden. Danach erfolgt eine grobe Beschreibung des Konzepts bzw. des Lösungsansatzes, detailliertere Erklärungen zu den enthaltenen Schwerpunkten finden sich dann in den nachfolgenden Kapiteln 5 und 6.

4.2.1 Definitionen

Bei der Festlegung der Begrifflichkeiten und Abstraktionsebenen existieren in der Literatur zum Bereich digitaler eingebetteter Hardware/Software-Systeme immer wieder mehrdeutige Definitionen, z.B. bei der Abgrenzung von Applikationen und Tasks. Die dem vorgeschlagenen Lösungsansatz zugrunde liegenden Definitionen sollen hier an zentraler Stelle erfolgen, um Missverständnisse und Fehlinterpretationen in den nachfolgenden Abschnitten und Kapiteln zu verhindern.

Definition 4.3 (Applikation)
Unter einer Applikation wird eine Einheit zur Realisierung einer gewünschten Funktionalität verstanden, die in Hardware und/oder Software implementiert sein kann. Eine Applikation ist zumeist unabhängig von anderen Applikationen, prinzipiell können allerdings gegenseitige Abhängigkeiten existieren. Die Begriffe „Anwendung" und „Applikation" können dabei als Synonyme verwendet werden.

Definition 4.4 (Task)
Eine Applikation eines eingebetteten Systems kann in mehrere Tasks unterteilt sein, die einen Datenfluss repräsentieren und somit in Zusammenarbeit zur Funktionserfüllung der Applikation beitragen. In Abgrenzung zu Applikationen bestehen zwischen Tasks deswegen in der Regel Datenabhängigkeiten, sodass durch die Semantik der Applikation eine Reihenfolge bei der Ausführung der Tasks abgeleitet werden kann. Ein Task benötigt – sofern es sich nicht um die Quelle handelt – dementsprechend Daten seiner Vorgänger und gibt diese nach der Bearbeitung an seine Nachfolger weiter.

Definition 4.5 (Verarbeitungsressource)
Eine Verarbeitungsressource (kurz: Ressource) ist eine Ausführungseinheit, auf der Applikationen und somit auch Tasks ausgeführt werden. Sie besitzt neben den Zuständen idle *und*

sleep *noch potentiell mehrere* run*-Zustände (vgl. Abschnitt 2.7.1), denen jeweils diskrete Ausführungsgeschwindigkeiten zugeordnet werden können. Für den Wechsel zwischen aufeinanderfolgenden Zuständen existieren definierte Umschaltzeiten, in denen keine Berechnungen von Applikationen bzw. Tasks ausgeführt werden dürfen.*

Definition 4.6 (Device)
Ein Device (zum Teil auch als Device-Komponente bezeichnet) ist eine Einheit, die Daten produziert und konsumiert, wie z.B. Speicherelemente, Ein- und Ausgabegeräte, Komponenten der Sensorik und Aktorik. Im Gegensatz zu Ressourcen führen sie Applikationen und Tasks nicht direkt aus, sondern sind denjenigen Applikationen und Tasks zugeordnet, welche die durch das Device produzierten Daten benötigt bzw. welche die für das Device notwendigen Daten erzeugt. Ein Device besitzt lediglich die Betriebszustände on *und* off*, wobei ebenfalls definierte Umschaltzeiten existieren.*

4.2.2 Analyse- und Optimierungsansatz

Wie in Abschnitt 4.1.4 bereits beschrieben sollen in dieser Arbeit *dynamische Mechanismen* zur Optimierung der Energieeffizienz angewandt werden. Diese lassen sich applikationsspezifisch steuern und versprechen aufgrund der Anwendung auf hoher Abstraktionsebene eine signifikante Minimierung des Energieverbrauchs. Entscheidend für den applikationsspezifischen Einsatz dieser Mechanismen ist die Frage, wann eine Bewertung des Gesamtsystems hinsichtlich der entscheidenden Eigenschaften vorgenommen werden kann. Dies ist nicht zuletzt abhängig von den zu diesem Zeitpunkt feststehenden Informationen und der Komplexität, mit der diese Informationen in eine sinnvolle Strategie bezüglich des Optimierungsziels transformiert werden können.

Ein grundsätzliches Konzept zur Optimierung der Energieeffizienz in digitalen eingebetteten Hardware/Software-Systemen ist in Abbildung 4.1 dargestellt. Dieses Konzept ist in unterschiedliche Phasen der Optimierung aufgeteilt, welche die jeweils zur Optimierung verfügbaren Eigenschaften widerspiegeln und für den jeweiligen Optimierungsschritt genutzt werden können, um eine Erhöhung der Energieeffizienz des Gesamtsystems zu erreichen. Hauptsächlich muss bei den zur Verfügung stehenden Informationen unterschieden werden, ob diese bereits zur Entwurfszeit feststehen oder von den dynamischen Eigenschaften bei der Ausführung der Funktionalität abhängen. Auf beiden Optimierungsebenen wird davon ausgegangen, dass nicht nur eine entsprechend der Ebene angepasste Modellierung der Funktionalität in Software vorliegt, sondern auch die zugrunde liegenden Hardware-Plattform modelliert ist. Weiterhin müssen die an das Hardware/Software-System gestellten Anforderungen definiert sein, um daraus den Optimierungsrahmen und damit die möglichen Freiheitsgrade während der Optimierung ableiten zu können. Dies ist deshalb von essentieller Bedeutung, da die Leistungsfähigkeit eines System und der Energieverbrauch, der zur Erbringung dieser Leistungsfähigkeit benötigt wird, in der Regel eng miteinander verknüpft sind – eine hohe Leistungsfähigkeit ist meist nur durch einen erhöhten Energieverbrauch zu erreichen. Wird aber eine geringere Leistungsfähigkeit gefordert, kann der Energieverbrauch gesenkt werden. Dies führt wiederum zu einer Steigerung der Energieeffizienz, da die Anforderungen an das System trotz Reduzierung der Leistungsaufnahme erfüllt

werden.

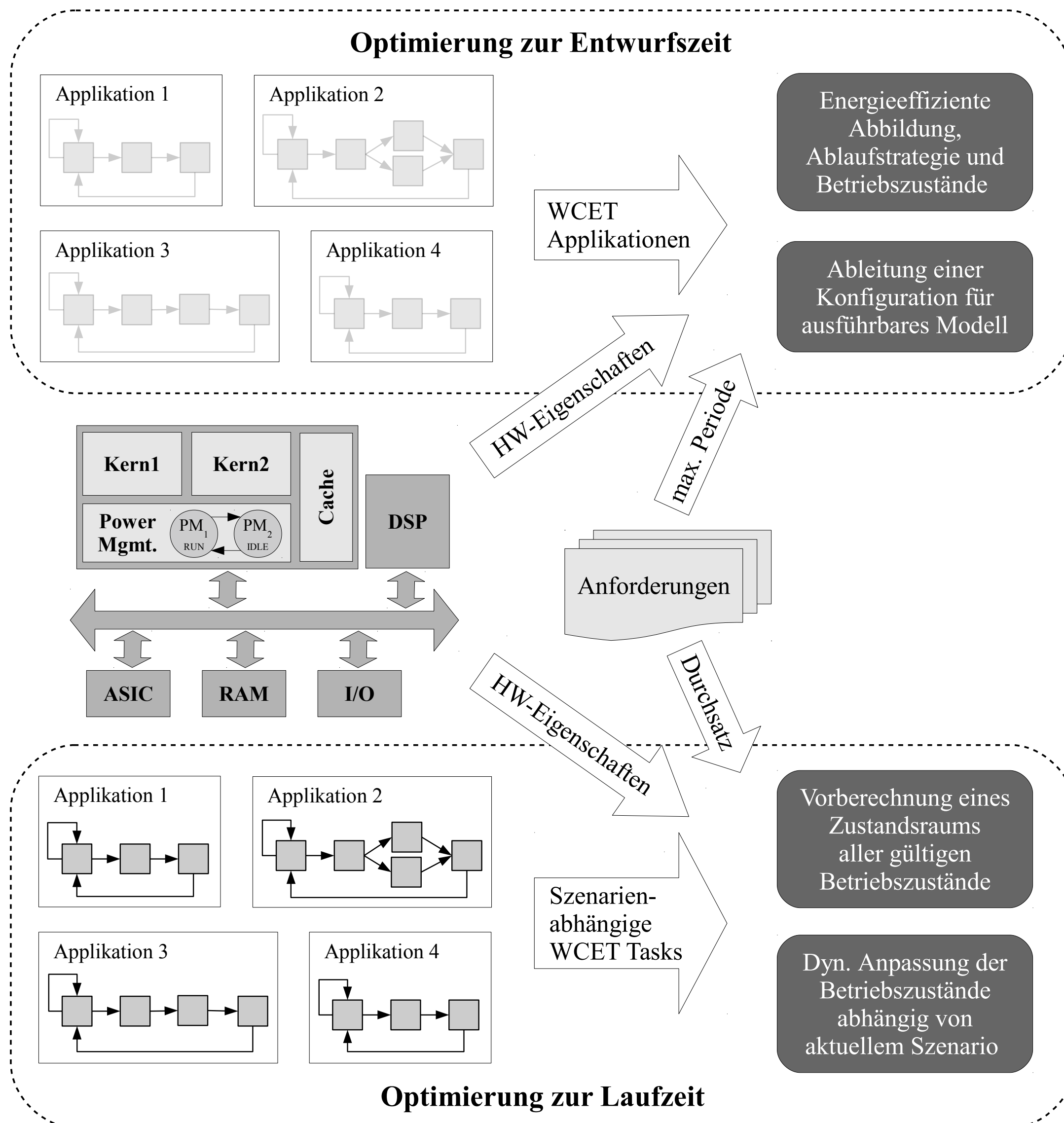


Abbildung 4.1: Konzept zur Optimierung der Energieeffizienz in unterschiedlichen Phasen

Funktionale, vor allem aber nicht-funktionale Eigenschaften, die bereits zur *Entwurfszeit* definiert sind, werden für eine einmalige oder iterative Optimierung während des Systementwurfs verwendet. Solche Eigenschaften gelten meist für einzelne Applikationen, die im Allgemeinen nur wenige gegenseitige Abhängigkeiten mit anderen Applikationen aufweisen. Weiterhin spielt der interne Aufbau einer Applikation eine untergeordnete Rolle, weshalb dieser in der Abbildung nur angedeutet ist. Die Laufzeit wird lediglich durch die Angabe einer festen unteren bzw. oberen Schranken, wie z.B. der *Worst-Case Execution Time (WCET)*, begrenzt. Bei dieser Optimierung auf Ebene einzelner Applikationen existieren im Allgemeinen viele Freiheitsgrade bezüglich der Ausführungsreihenfolge und der Abbildung der Applikationen auf Verarbeitungsressourcen, weshalb in höchstem Maße eine ganzheitliche Sicht auf das

Hardware/Software-System benötigt wird [84]. Das Ergebnis des Optimierungsprozesses ist eine bezüglich des Energieverbrauchs optimierte Abbildung der Applikationen auf Verarbeitungsressourcen, eine auf die Abbildung abgestimmte Ablaufstrategie und eine explizite Zuordnung von Betriebszuständen zu den jeweiligen Applikationen. Basierend auf diesem Ergebnis wird eine Konfiguration für ein ausführbares Systemmodell abgeleitet, welches das funktionale und nicht-funktionale Verhalten des Hardware/Software-Systems modelliert und simuliert.

Jedoch stellen Ausführungszeiten, die anhand ihres Worst-Case-Verhaltens definiert werden, lediglich obere Schranken der statisch bestimmbaren Laufzeit dar. Somit kann das dynamische Verhalten innerhalb von Anwendungen bzw. Applikationen, vornehmlich aufgrund unterschiedlicher Ausführungspfade der daran beteiligten Tasks, bei einer Optimierung zur Entwurfszeit nicht berücksichtigt werden. Liegen jedoch während der Nutzung des Systems detailliertere Angaben bezüglich der momentan geltenden Eigenschaften vor, kann der interne Aufbau der Applikationsfunktionalitäten durch Tasks und deren dynamische Eigenschaften ebenfalls zur Optimierung herangezogen werden. Für eine weitere Steigerung der Energieeffizienz wird deshalb ein Analysemodell entwickelt, das den Ablauf der einzelnen Tasks abhängig vom aktuellen Szenario repräsentiert. Dadurch erfolgt zur Entwurfszeit die Berechnung eines Zustandsraums, der alle gültigen Betriebszustände des modellierten Hardware/Software-System enthält, die von diesem während der Laufzeit angenommen werden können. Basierend auf dieser Vorberechnung erfolgt während der Laufzeit und unter Berücksichtigung des aktuellen Szenarios eine dynamische Anpassung der Betriebszustände, indem der Zustandsraum exploriert wird.

Aus dem eben beschriebenen Konzept zur Optimierung der Energieeffizienz ergibt sich der in Abbildung 4.2 abgebildete Entwurfsfluss, welcher gleichzeitig auch die Aufteilung der einzelnen Schritte in den nachfolgenden Kapiteln darstellt.

Ausgangspunkt des Entwurfsflusses, auf dem der Analyse- und Optimierungsansatz basiert, ist ein Systemmodell, welches die zu realisierenden Funktionalität, die zugrunde liegenden Hardware-Plattform und die an das Hardware/Software-System gestellten Anforderungen umfasst.

Ausgehend von diesem Systemmodell wird ein Analysemodell erstellt, das unter Verwendung mathematischer Formulierungen ein Optimierungsproblem löst und somit den Gesamtenergieverbrauch minimiert. Dieses Optimierungsproblem resultiert in einer optimierten Abbildung der Applikationen auf Verarbeitungsressourcen und einer Zuordnung dieser Applikationen zu Betriebszuständen, die auf den jeweiligen Ressourcen zur Verfügung stehen. Die Optimierung erfolgt dabei unter Berücksichtigung aller an das System gestellten Anforderungen. Aufgrund der verwendeten Abstraktionsebene kann diese Optimierung zur Entwurfszeit, die in Abschnitt 6.1 im Detail behandelt wird, mit dem in der Literatur verwendeten Begriff *Inter-Task-Optimierung* verglichen werden, da die Betriebszustände nur jeweils zwischen den Applikationen gewechselt werden können.

Die aus diesem ersten Optimierungsschritt gewonnenen Ergebnisse über die energieeffiziente Abbildung der Applikationen und die Ablaufstrategie werden in Kombination mit dem Systemmodell zur automatischen Generierung eines ausführbaren Modells bzw.

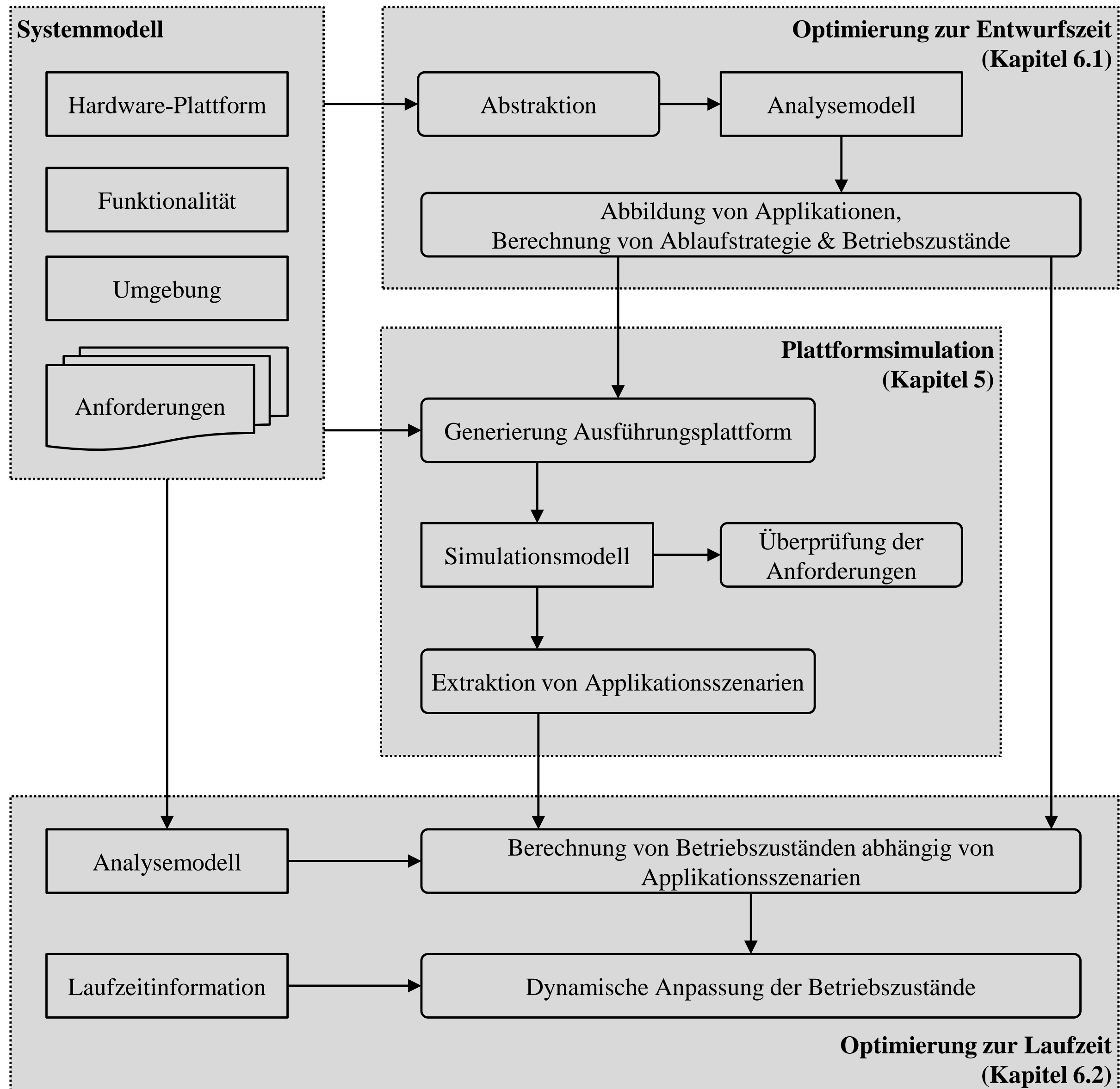


Abbildung 4.2: Aus dem Konzept abgeleiteter Entwurfsfluss

eines *virtuellen Prototypen* in SystemC verwendet. Da SystemC jedoch die Ausführung von eingebetteter Software nur unzureichend abdeckt, wird zusätzlich eine virtuelle Ausführungsplattform generiert, die die im Systemmodell enthaltene Funktionalität in Software-Komponenten kapselt und unter Berücksichtigung der Ablaufstrategie und der verfügbaren Verarbeitungsressourcen auf einer virtuellen Hardware-Plattform ausführbar macht, so dass deren Semantik erhalten bleibt. Dazu beinhaltet diese Ausführungsplattform ein dynamisches Powermanagement, das auf einem zustandsbasierten Modell der Leistungsaufnahme basiert (siehe dazu Abschnitt 4.1.5). Liegt die auszuführende Funktionalität oder zumindest Teile davon in C/C++-Quelltext vor, wird das zustandsbasierte Leistungsmodell und das damit verbundenen dynamische Powermanagement mit einem instruktionsbasierten Leistungsmodelle kombiniert, um die Genauigkeit der Simulation zu erhöhen. Mithilfe dieser gesamten virtuellen

Ausführungsplattform wird das Zeitverhalten der Funktionalität auf einer realen Zielarchitektur unter Berücksichtigung der spezifizierten Ablaufstrategie und der Integration eines dynamischen Powermanagements simuliert. Die Transformation des Systemmodells in ein Simulationsmodell erfolgt anhand definierter Transformationsregeln, was eine frühzeitige und schnelle Generierung des ausführbaren Modells erlaubt. Diese automatische Generierung der virtuellen Ausführungsplattform wird ausführlich in Kapitel 5 behandelt. Durch die ganzheitliche Simulation, die auch die spezifizierte Umgebung des Hardware/Software-Systems mit einschließt, wird zum einen eine frühzeitige Verifikation bzw. Validierung des Systemverhaltens und eine damit verbundenen Überprüfung der Anforderungen auch unter Berücksichtigung verschiedenster Parameter und dynamischer Eigenschaften ermöglicht. Zum anderen können durch eine genügend große Abdeckung von Testfällen typische Applikationsszenarien extrahiert werden, die als Eingabe für einen weiteren Optimierungsprozess dienen.

Dieser Optimierungsschritt bezieht das dynamische Verhalten des Systems aufgrund unterschiedlicher Ausführungspfade in die Optimierung der Energieeffizienz ein, indem zur Entwurfszeit alle gültigen Betriebszustände unter Berücksichtigung der möglichen Szenarien der Applikationen berechnet werden. Diese Vorberechnung ermöglicht die Anpassung der Betriebszustände zur Laufzeit in Abhängigkeit des aktuellen Applikationsszenarios. Das Ausnutzen des während der Ausführung festgestellten dynamischen Verhaltens zur Steigerung der Energieeffizienz wird in der Optimierung zur Laufzeit thematisiert, die wie in Abbildung 4.2 zu sehen eingehend in Abschnitt 6.2 beschrieben wird. Aufgrund der Abstraktionsebene und dem Anpassen der Betriebszustände innerhalb einer Applikation kann diese Optimierung auch als *Intra-Task-Optimierung* bezeichnet werden.

Kapitel 5

Modellierung und Simulation digitaler Hardware/Software-Systeme

Heutige Systeme zeichnen sich durch einen hohen Grad an Komplexität aus. Diese Form der Entwicklung dürfte in der Zukunft sogar noch stark zunehmen, was in gleichem Maße auch für digitale eingebettete Hardware/Software-Systeme gilt. Waren beispielsweise im Jahr 1989 drei Busteilnehmer in Automobilen der Standard, so sind es heute weit über siebzig Teilnehmer, die über mehrere Bussysteme kommunizieren [31]. Hauptgründe für diese Entwicklung sind vor allem die steigenden Anforderungen an die Funktionalität und die Verfügbarkeit moderner Hardware-Architekturen, aber auch die stärkere Vernetzung zwischen einzelnen Systemen und Subsystemen. Diese Vernetzung führt mitunter dazu, dass ein Mehrwert an Funktionalität nicht nur durch Hinzufügen einzelner Module und Komponenten, sondern ganz speziell durch die Verknüpfung bereits existierender Komponenten und Funktionalitäten realisiert werden kann. Dies führt im Umkehrschluss aber auch dazu, dass beim Entwurf und der Entwicklung dieser Systeme ein zunehmendes Betrachten von Gesamtsystemen anstatt von einzelnen Modulen gefordert ist.

Im Hinblick auf eine *ganzheitliche Simulation* ist eine der größten Herausforderungen, möglich schnell und automatisiert zu einem ausführbaren Modell des zu entwickelnden Hardware/Software-Systems zu gelangen. Weiterhin sollte sich der zusätzliche Aufwand, der in die Entwicklung des virtuellen Prototyps gesteckt werden muss, in Grenzen halten, was ebenfalls durch einen hohen Grad an Automatisierung erreicht werden kann. Zugleich sinkt durch Automatisierung die Wahrscheinlichkeit, dass durch eine manuelle Implementierung Fehler in der Entwicklung der Modelle entstehen. Besonders bei Systemen mit hoher Komplexität kann die Modellierung des Systems vereinfacht und durch eine automatisierte Generierung des Simulationsmodells vermeidbare Fehlerquellen ausgeschlossen werden [141].

In diesem Kapitel wird zunächst ein Ansatz zur Modellierung von eingebetteten Hardware/Software-Systemen vorgestellt. Anschließend wird beschrieben, wie das resultierende Modell ausgeführt werden kann und wie dieses ausführbare Systemmodell durch automatische Schritte generiert werden kann. Des Weiteren wird veranschaulicht, wie Mechanismen der Energieminimierung in das Systemmodell integriert und simuliert werden können. Zum Ende hin wird ein Verfahren diskutiert, wie eine simulative

Analyse und die Extraktion der Dynamik des Hardware/Software-Systems während der Simulation dazu benutzt werden kann, ein analytisches Modell zu verfeinern und die Energieeffizienz dadurch zu optimieren.

5.1 Modellierung

Eine wichtige Anforderung für die Abbildung von Software-Funktionalität auf Single- und Multicore-Architekturen ist die freie Abbildung und Verschiebung von Software-Komponenten, sowie deren entsprechende Konfiguration [138]. Dies ist auch die Grundlage für einen komponentenbasierten Entwurfsprozess, der im Hinblick auf die Ausführungssemantik noch näher in Abschnitt 5.2 erläutert wird.

Eine der größten Herausforderung für die praktische Anwendung eines komponentenbasierten Entwurfsprozesses ist die Verwendung einer verständlichen und domänenübergreifenden Modellierungsmethodik, die unterschiedliche Sichtweisen auf das zu modellierende Hardware/Software-System erlaubt. Diese Herausforderung bedingt, dass die Modellierungsmethodik auf einem wohl verstandenen und akzeptierten, sowie standardisierten Verfahren aufgesetzt werden sollte. Vor allem im Bereich der modellbasierten Softwareentwicklung hat sich in dieser Hinsicht die *Unified Modeling Language (UML)* zur am weitesten verwendeten Modellierungssprache entwickelt. Aus Gründen der Anwendbarkeit und Unterstützung bereits etablierter Entwicklungsprozesse bietet es sich also an, die Modellierungssprache UML auch für die ganzheitliche Modellierung eines Hardware/Software-Systems zu benutzen. Weiterhin existiert eine Vielzahl von Modellierungsumgebungen und -werkzeuge, die sich zumindest weitestgehend an den UML-Standard halten. Ausführlichere Details zum UML-Standard können dem Grundlagenabschnitt 2.2.1 entnommen werden. Um eine vielseitige Modellierung zu unterstützen, verfügt UML über ein Profil-Konzept. Das bedeutet, dass der Sprachstandard UML durch zusätzliche Elemente, Bedeutungen, Diagramme, und Eigenschaften erweitert werden kann, um die Sprache an die Anforderungen der anvisierten Modellierungskontexte, wie z.B. verschiedene Anwendungsdomänen oder Abstraktionsebenen, anpassen zu können.

Für die Beherrschung der zunehmenden Systemkomplexität, die insbesondere bei einer ganzheitlichen Modellierung eingebetteter Hardware/Software-Systeme gegeben ist, ist es außerdem unbedingt notwendig, Modelle auf verschiedenen Abstraktionsebenen integrieren und verwenden zu können. Für Software-intensive Systeme können die Abstraktionsebenen von der abstrakten Modellierung öffentlicher Schnittstellen bis zur exakten Modellierung der Funktionalität von Software und Hardware bzw. der Einbindung bereits existierender Implementierungen reichen.

Die entwickelte Modellierungsmethodik wird im Folgenden am Beispiel einer Cockpit-By-Wire-Anwendung – also einer Kombination aus Steer-By-Wire und Brake-By-Wire – aus dem Automobilbereich aufgezeigt. Darin werden aus Sensoren, die sich sowohl an der Lenkradeinheit als auch an der Pedalerie befinden, die notwendigen Daten zum Steuern eines Fahrzeugs ausgelesen und in einem Steuergerät aufbereitet, um danach die Aktuatorik anhand der empfangenen Daten ansteuern zu können.

Die relevanten Daten umfassen den aktuellen Lenkradwinkel, sowie die jeweiligen Stellungen des Brems- und Gaspedals. Als Modellierungsumgebung wird das Werkzeug *Papyrus* als Erweiterung für die *Eclipse*-Entwicklungsumgebung verwendet. Prinzipiell kann jedoch jede Modellierungsumgebung verwendet werden, die den UML-Standard ab Version 2.0 einhält.

5.1.1 Modellierung der Funktionalität

Um die hohe Komplexität während des Entwurfsprozesses Software-intensiver eingebetteter Hardware/Software-Systeme beherrschen zu können, werden diese oftmals auf sehr abstrakter Ebene modelliert, um dann in mehreren Verfeinerungsschritten zu einer Systemimplementierung zu gelangen. Um den Verfeinerungsprozess zu beschleunigen und um Fehler zu vermeiden, sollten diese Verfeinerungsschritte möglichst automatisiert durchgeführt werden. Weiterhin kann das System durch eine strikte Trennung der Systemarchitektur und der zugrunde liegenden Hardware-Plattform von der eigentlichen Funktionalität konfiguriert werden, ohne dass sich diese Bereiche gegenseitig beeinflussen, was einen Optimierungsprozess deutlich vereinfacht.

Als erster Schritt muss dementsprechend das funktionale Verhalten des Hardware/Software-Systems modelliert werden. In einem komponentenbasierten Entwicklungsprozess, der auf UML aufbaut, werden hierfür *UML-Komponenten* als Spezialisierung von *UML-Klassen* verwendet.

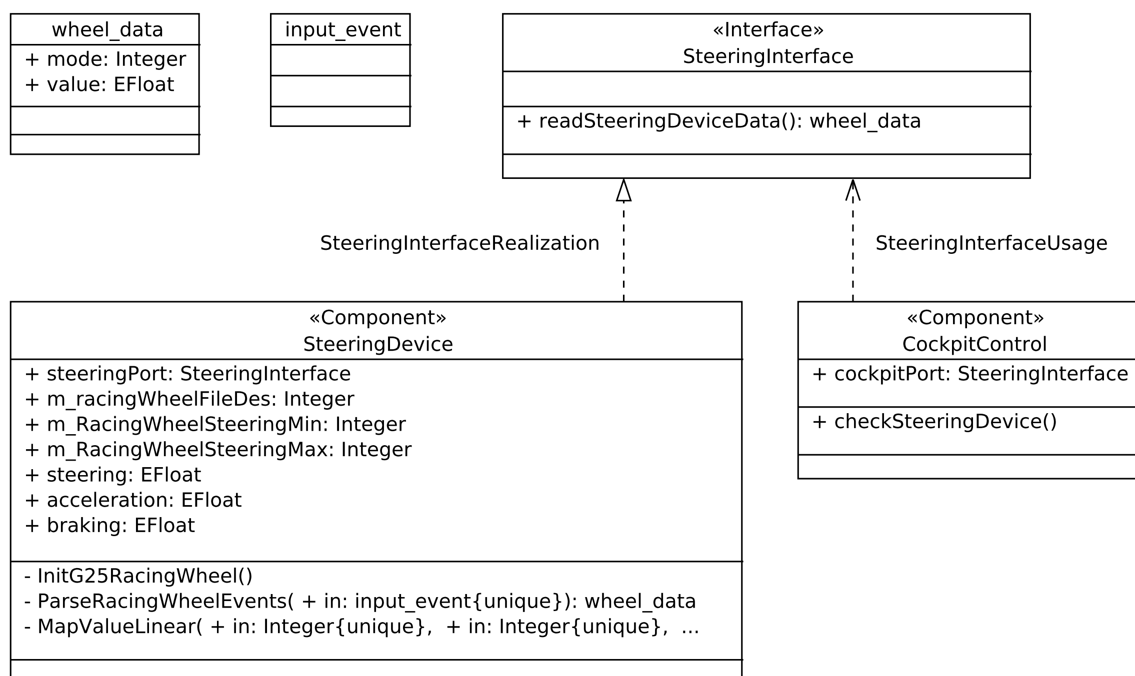


Abbildung 5.1: Modellierung der Funktionalität der Software-Komponenten

Abbildung 5.1 stellt die Funktionalität der Steuer-Komponente in einem *UML-Klassendiagramm* dar. Die Komponente `SteeringDevice` bietet die öffentliche Methode `readSteeringDevice()` durch eine *Angebotsschnittstelle* (engl.: *Provided Interface*)

`SteeringInterface` an. Diese Methode gibt einen komplexen Datentyp `wheel_data` in Form einer UML-Klasse, die den Modus bzw. die Kodierung des gerade ausgelesenen Sensors und den entsprechenden Wert enthält, an die aufrufende Instanz zurück. Zusätzlich enthält `SteeringDevice` weitere Variablen und private Methoden, die durch die Komponente gekapselt werden. Analog zur Komponente `SteeringDevice` wird auch die Funktionalität des Cockpits durch eine UML-Komponente `CockpitControl` modelliert. Diese Komponente benutzt die von `SteeringDevice` angebotene Schnittstelle `SteeringInterface`, diese bildet somit eine *Bedarfsschnittstelle* (engl.: *Required Interface*) für die Komponente `CockpitControl`. Auch diese Komponente enthält weitere Variablen und Methoden, die durch sie gekapselt werden. Innerhalb der Methode `checkSteeringDevice()` wird die Schnittstellenmethode `readSteeringDevice()` aufgerufen, die wie zuvor bereits dargestellt durch die Angebotsbeziehung von der Komponente `SteeringDevice` implementiert werden muss.

5.1.2 Modellierung der Hardware-Plattform

Neben der Modellierung der Funktionalität, wie sie in Abschnitt 5.1.1 beschrieben ist, spielt bei digitalen Hardware/Software-Systemen auch die Modellierung der zugrunde liegenden Hardware-Plattform eine entscheidende Rolle. Insbesondere gilt dies für den Einfluss der Hardware-Plattform auf die funktionalen und nicht-funktionalen Eigenschaften des Systems. Die Genauigkeit der nicht-funktionalen Eigenschaften während der Simulation hängt maßgeblich von der Abstraktionsebene des Modells der *virtuellen Hardware* ab. Transaktionsorientierte Kommunikationsarchitekturen ermöglichen zwar eine schnelle Simulation des Datentransports, abstrahieren aber von den wirklich zu übertragenden Daten und den komplexen Mechanismen des Medienzugriffs. Bei berechnenden Hardware-Elementen können ebenso Details der Mikroarchitektur abstrahiert werden, wie z.B. Cache-Zugriffe oder Pipeline-Verarbeitungen eines Prozessors.

Eine wichtige Anforderung an die Modellierung der Hardware-Plattform ist deshalb deren Unabhängigkeit von der zu modellierenden Abstraktionsebene. Zur eindeutigen Abgrenzung von Hardware-Komponenten und Software-Komponenten werden Hardware-Komponenten unter Verwendung der *Systems Modeling Language (SysML)* modelliert. SysML ist ein Profil, das den UML-Standard erweitert und speziell für die Modellierung von physikalischen Systemen entwickelt wurde. Die Hauptelemente von SysML bilden *SysML-Blöcke*, die meist physikalische Elemente modellieren, weshalb diese auch zur Modellierung der einzelnen Hardware-Komponenten verwendet werden.

In Abbildung 5.2 ist die Modellierung der Hardware-Plattform in SysML dargestellt. Im Gegensatz zum Komponentenmodell der Software, das auf Methodenaufrufen basiert, liegt der Fokus bei der Kommunikation der Hardware-Komponenten auf Datenstrom-orientierten Kommunikationsmechanismen. Zur Verwendung von Datenstrom-orientierten Schnittstellen für die jeweiligen Ein- bzw. Ausgangs-Ports dient eine Adapterbibliothek `UML2SystemCAdapter`, die Elemente der TLM-2.0-Bibliothek mit UML-Elementen semantisch verbindet und damit die Typisierung der Ports durch `simple_initiator_socket` als Sende-Port und `simple_target_socket` als Empfangs-

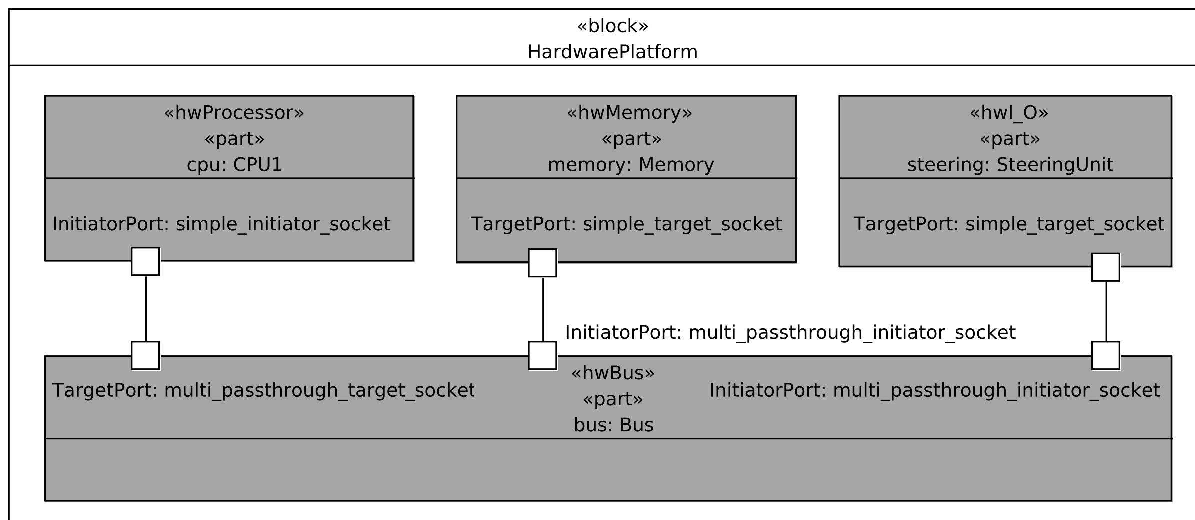


Abbildung 5.2: Modellierung der Hardware-Plattform

Port erlaubt. Zur Abbildung von hierarchischen Komponentenmodellen können Komponenten als *Part* in anderen Komponenten enthalten sein, was auch eine Verbindung von deren Ports durch UML-Konnektoren erlaubt. Zur Darstellung von Datenverbindungen auf niedrigeren Abstraktionsebenen können Ports auch durch primitive Datentypen, die in SystemC zur Verfügung stehen, typisiert werden, was einer direkten Übertragung von Daten dieses Typs entspricht.

Zur Interoperabilität mit existierenden Werkzeugketten, die auf dem vor allem in der Halbleiterindustrie eingesetzten IP-Austauschformat *IP-XACT* zur Spezifizierung von Hardware-Komponenten aufbauen, kann die Modellierung der Hardware-Plattform auch im standardisierten IP-XACT-Format erfolgen. Dazu existiert eine direkte Abbildung von Elemente des IP-XACT-Metamodells auf UML/SysML-Elemente, sodass diese mithilfe eines zusätzlichen Adapters problemlos in den hier vorgestellten modellbasierten Prozess integriert werden können.

5.1.3 Modellierung von Abhängigkeiten und Deployment

Die bisher modellierten Funktionalitäten – sowohl der aufrufenden wie auch der aufgerufenen Seite – definieren jedoch noch keine funktionale Abhängigkeit zwischen den beiden Komponenten `SteeringDevice` und `CockpitControl`. Insbesondere wird bisher zwar funktionales Verhalten modelliert, dieses wird jedoch noch keinen ausführenden Instanzen zugewiesen. Weiterhin können Komponenten Teil eines hierarchischen Systems sein, wodurch Beziehungen über Hierarchiegrenzen bzw. über weitere Komponenten hinweg modelliert werden müssen. Abbildung 5.3 zeigt die Modellierung der funktionalen Abhängigkeit bzw. der Aufrufbeziehung in einem *UML-Kompositionsstrukturdiagramm*. Dafür werden die modellierten Software-Komponenten innerhalb einer Systemkomponente auf höchster Hierarchieebene instanziiert. Im vorliegenden Fall sind dies die Instanzen `cockpit` und `steering`. Weiterhin werden die in diesem Diagrammtyp darstellbaren *UML-Ports* als Interaktionspunkte zwischen den

Komponenten modelliert, die durch die gemeinsame Schnittstelle `SteeringInterface` typisiert sind. Da diese Typisierung die Aufrufbeziehung der Ports und dadurch der jeweiligen umschließenden Komponenten definiert, können nur Ports desselben Typs verbunden werden. Die Verbindung selbst wird durch einen *UML-Konnektor* modelliert und wird als direkte Aufrufbeziehung zwischen den jeweiligen Komponenten interpretiert. Dementsprechend werden Methodenaufrufe weitergeleitet, wenn Ports über Hierarchiegrenzen hinweg verbunden sind, oder sich weitere Komponenten zwischen aufrufender und aufgerufener Komponente befinden.

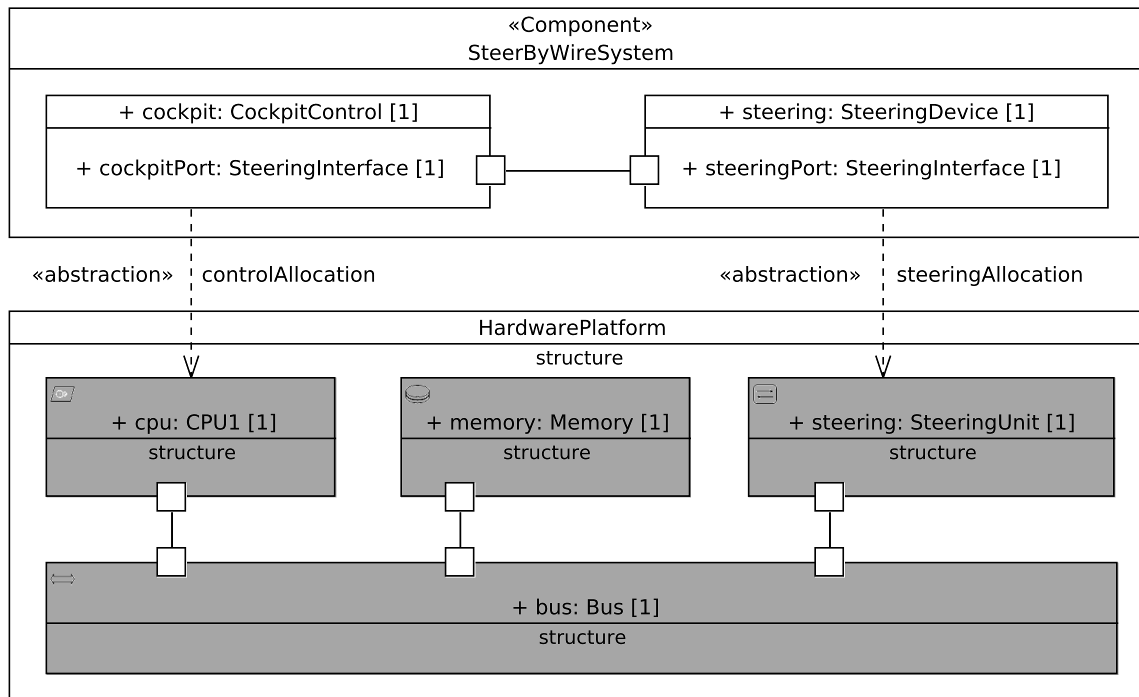


Abbildung 5.3: Modellierung des Hardware/Software-Abbildung

Zur ganzheitlichen Modellierung des Hardware/Software-Systems muss zusätzlich die Abbildung (engl: *Deployment*) der Funktionalitätselemente auf die Komponenten der Hardware-Plattform modelliert werden. Damit wird spezifiziert, welche Software-Komponenten auf welcher Hardware-Komponente ausgeführt werden. In einem UML-Kompositionsstrukturdiagramm werden sowohl Funktionalitäts- als auch Hardware-Sicht in einem Diagramm integriert. Funktionalitätselemente werden durch die Verwendung von *UML-Abstraktionen*, die eine Spezialisierung der *UML-Dependency*-Beziehung darstellen, auf Elemente der Hardware-Plattform abgebildet, was in Abbildung 5.3 unter Verwendung der Allokationen `controlAllocation` und `steeringAllocation` mit gestrichelten Pfeilen dargestellt ist. Die Instanz der `CockpitControl`-Komponente wird auf der Hardware-Komponente `cpu` vom Typ `CPU1` ausgeführt, die `SteeringDevice`-Komponente wird auf die `SteeringUnit`-Komponente der Hardware-Plattform abgebildet. Die `SteeringUnit`-Komponente ist mit dem Stereotyp *hwI_O* gekennzeichnet. Dieser Stereotyp ist wie die weiteren bei der Modellierung der Hardware-Plattform verwendeten Stereotypen, wie z.B. *hwProcessor*, *hwMemory*

und *hwBus*, Teil des UML-Profils *MARTE*[1][91], das damit eine Typisierung sowohl generischer, als auch spezieller Hardware-Komponenten erlaubt. Welche Eigenschaften der Hardware-Plattform innerhalb der Stereotypen des MARTE-Profils konfiguriert werden können, wird in Abschnitt 5.1.4 beschrieben.

Weiterhin muss die Semantik der eigentlichen Ausführung der spezifizierten Funktionalität auf der jeweils durch das Deployment zugeordneten Hardware-Komponente modelliert werden. Abbildung 5.4 stellt diese Zuordnung unter Verwendung des MARTE-Profils dar. Es steht mit *swSchedulableResource* ein Stereotyp zur Verfügung, der ein abstraktes Aktivierungsschema an eine Funktionalität in Form einer Methode – in UML-Syntax als *Operation* bezeichnet – binden kann. Dazu wird die betreffende Methode, in diesem Fall `checkSteeringDevice()`, der Liste *entryPoints* hinzugefügt. Gleichzeitig kann als ausführende Einheit in der Eigenschaft *host* eine Instanz einer *Scheduler*-Klasse spezifiziert werden. Diese Scheduler-Instanz kann wiederum einer bestimmten Hardware-Komponente zugewiesen werden, wodurch die UML-Operation eindeutig an die Hardware-Komponente `CPU1` gebunden wird. Innerhalb der Scheduler-Instanz wird das jeweils gültige Aktivierungsschema in der Eigenschaft *schedPolicy* definiert. Unter anderem stehen vordefinierte Strategien zur Ablaufplanung (engl.: *Scheduling*) wie Earliest Deadline First (EDF), Round Robin, Fixed Priority und ein TDMA-Verfahren zur Auswahl, es können allerdings auch eigene Ablaufstrategien spezifiziert werden. Diese werden mithilfe der *Value Specification Language (VSL)* formuliert und als Wert *schedule* des Typs *ScheduleSpecification* hinterlegt. Zusätzlich können weitere Charakteristika der Ablaufstrategie definiert werden, z.B. ob eine Preemption, also ein Verdrängung des aktuell ausgeführten Thread durch einen anderen, erlaubt ist oder nicht. Die Ausführung bzw. Transformation der modellierten Ablaufstrategien in ein Simulationsmodell wird detailliert in Abschnitt 5.4.1 thematisiert.

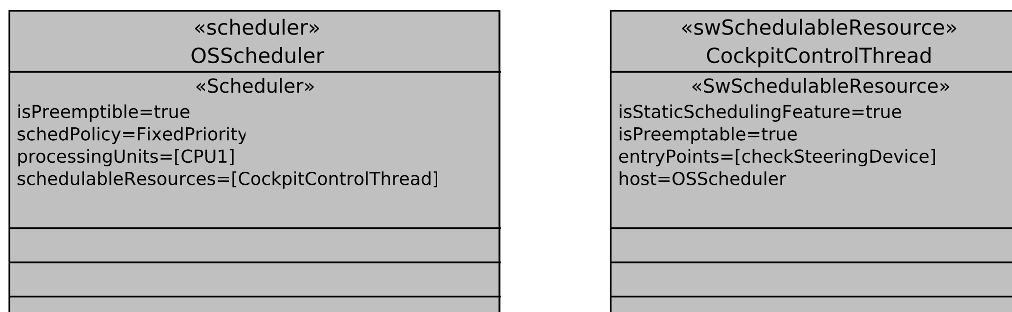


Abbildung 5.4: Modellierung der Ausführung von UML-Operationen

5.1.4 Spezifizierung nicht-funktionaler Eigenschaften

Im Begriff *nicht-funktionale Eigenschaften (NFP* (engl.: Non-Functional Properties) werden diejenigen Eigenschaften zusammengefasst, die keinen unmittelbaren Einfluss auf das rein funktionale Verhalten eines Systems haben, sondern vielmehr auf dessen zeitliches Verhalten, den Energieverbrauch und die Zuverlässigkeit. Aufgrund

[1]Modeling and Analysis of Real-Time Embedded Systems

der gegenseitigen Abhängigkeiten lassen sich funktionale und nicht-funktionale Eigenschaften nicht immer vollkommen getrennt betrachten. Wenn z.B. eine Verarbeitungseinheit zur Berechnung eines Ergebnisses auf die rechtzeitige Verfügbarkeit mehrerer Eingangsdaten angewiesen ist, kann eine unvorhersehbare Latenz dieser Daten auch Einfluss auf das funktionale Verhalten des Systems haben. Für die Modellierung nicht-funktionaler Eigenschaften von Hardware/Software-Systemen wird das bereits erwähnte UML-Profil *MARTE* verwendet. Dieses Profil enthält im Paket *MARTE_DesignModel::HRM::HwLogical* eine Vielzahl spezifischer Komponententypen einer Hardware-Plattform, u.a. Prozessoren, Caches, Speicherelemente und Busstrukturen. Abbildung 5.5 zeigt einige der modellierbaren Eigenschaften der im Beispiel verwendeten Hardware-Komponenten.

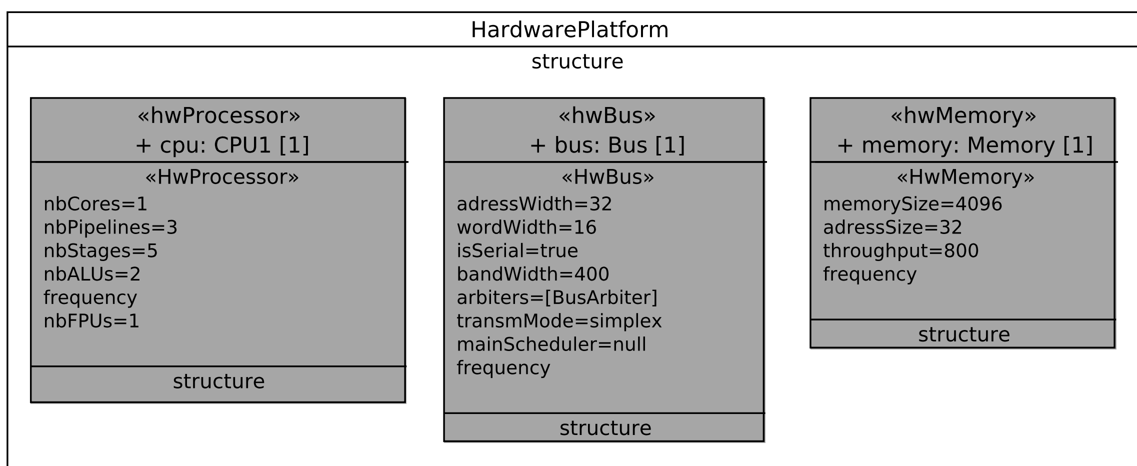


Abbildung 5.5: Nicht-funktionale Eigenschaften in der UML-Modellierung mit MARTE

Für Prozessoren können sowohl abstrakte Eigenschaften, wie z.B. die Anzahl der zur Verfügung stehenden Kerne und deren Frequenz, als auch fein-granulare Eigenschaften der Mikro-Architektur, wie z.B. die Anzahl der Pipelines, der algorithmisch-logischen Einheiten und der Einheiten zur Verarbeitung von Fließkommazahlen, angegeben werden. Für die verwendete Bus-basierte Kommunikationsstruktur kann ebenfalls die Bandbreite und die Übertragungsfrequenz, aber auch Zugriffseigenschaften wie die Adress- und Wortbreite spezifiziert werden. Zusätzlich werden Verknüpfungen zur Zugriffsverwaltung und Ablaufplanung hergestellt. Bei Speicherkomponenten werden Eigenschaften wie Größe und Performanz innerhalb der MARTE-Stereotypen definiert. Diese Eigenschaften können einen signifikanten Einfluss auf die nicht-funktionalen Eigenschaften des Gesamtsystems haben. Simulations- und Analysemodelle, die durch manuelle Überführung oder (semi-)automatische Transformationsschritte erzeugt werden, werden anhand der eben beschriebenen Modelleigenschaften parametrisiert, sodass ein direkter Bezug zum Systemverhalten hergestellt wird. Ein Ansatz zur automatischen Transformation in ein ausführbares Simulationsmodell wird in Abschnitt 5.2 beschrieben, wobei die Transformationsschritte während des Generierungsprozesses in Abschnitt 5.3 noch näher erläutert werden.

5.2 Aufbau einer virtuellen Ausführungsplattform

Grundlage für die Generierung einer virtuellen Ausführungsplattform bzw. eines ausführbaren Systemmodells ist die Modellierung des Hardware/Software-Systems in UML, die bereits in Abschnitt 5.1 dargestellt wurde. Der Schwerpunkt dieses Abschnitts liegt auf der Definition einer *Ausführungssemantik* des virtuellen Prototyps. Der hierfür vorgestellte Ansatz basiert auf einem *mehrstufigen Schichtenmodell*, das die Realisierung einer Plattform zur Ausführung von Software-Komponenten auf einer zugrunde liegenden Hardware-Plattform in SystemC ermöglicht [142]. Abhängig von der betrachteten Schicht können dabei unterschiedliche funktionale und nicht-funktionale Eigenschaften berücksichtigt werden, die in ihrer Gesamtheit eine möglichst realistische Ausführung des virtuellen Simulationsmodells bewirken. Es sollen dazu Modellierungs- und Ausführungskonzepte von SystemC auf die in dieser Arbeit adressierten Aufgabenstellung übertragen werden. Im Vordergrund steht dabei die Ausführung der Software-Komponenten anhand einer definierten Ablaufstrategie und ein damit verbundene Integration eines Energiemanagements. SystemC als Simulationswerkzeug stellt die Basis dar, verfügt jedoch über keine Systemverwaltung, die das Verhalten eines eingebetteten Hardware/Software-Systems akkurat repräsentiert, wie es z.B. bei der Verwendung geteilter Ressourcen bzw. der Ausführung von Software auf mehreren Recheneinheiten unbedingt notwendig ist. Diese Funktionalität muss durch eine separate Mittelschicht gewährleistet werden. Generell ist in SystemC die Ausführung von Software nur unzureichend abgedeckt. Weiterhin muss ein zusätzliches Modell miteinbezogen werden, das die Leistungsaufnahme der gesamten Hardware-Plattform abbildet. Zwei grundsätzliche Modelle zur Abbildung der Leistungsaufnahme wurden bereits in Abschnitt 4.1.5 diskutiert.

5.2.1 Software-Schicht

In der Regel besteht in einer klassischen objektorientierten Softwarearchitektur entweder eine direkte Aufrufbeziehungen zwischen aufrufender und aufgerufener Komponente oder indirekt über eine wohldefinierte Schnittstelle. Das bedeutet, dass eine Methode auf ein sichtbares Objekt bzw. auf die Instanz einer Klasse aufgerufen wird. Die Schnittstelle stellt also mögliche Interaktionspunkte zur Verfügung, wodurch die Semantik der Komponente ausgedrückt wird. Grundlegend existieren bei der Interaktion zwei zu betrachtende Seiten – die aufrufende und die aufzurufende. Die Schnittstellen werden dementsprechend als *Bedarfsschnittstelle* und *Angebotsschnittstelle* bezeichnet. Die Angebotsschnittstelle gibt an, welche Funktionalität und welche Dienste die jeweilige Komponente ihren Dienstnehmern anbietet. Die Bedarfsschnittstelle wiederum gibt an, was die Komponente von ihrer Umgebung erwartet.

Da es sich bei dem entwickelten Schichtenmodell um einen allgemeingültigen Ansatz handelt, soll dieser anhand eines abstrakten Beispiels eines Software-intensiven eingebetteten Hardware/Software-Systems vorgestellt werden. In Abbildung 5.6 ist eine Software-Funktionalität in Form von zwei Klassen `A` und `B` dargestellt. Beide Klassen verfügen über eine gemeinsame Schnittstelle `B_IF`, die wiederum zwei Methoden

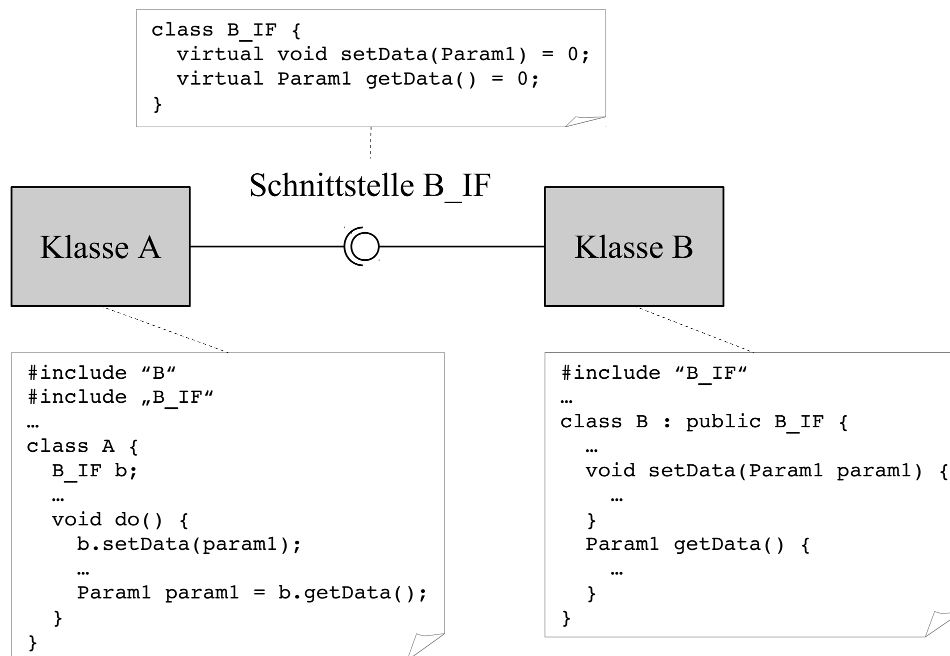


Abbildung 5.6: Struktur einer klassischen Softwarearchitektur

enthält – `getData()` mit Angabe eines Rückgabewerts und `setData()` mit Übergabe eines Parameters. Diese Methoden stellen auch exemplarisch die zwei grundlegend unterschiedlichen Charakteristika von Methoden dar. Zum einen existieren Methoden, die einen Rückgabewert besitzen und somit eine Daten-empfangende Semantik besitzen. Zum anderen Methoden, die eine Daten-versendende Semantik besitzen, indem der Methode ein oder mehrere Parameter übergeben werden. Auftretende Mischformen werden durch eine gleichzeitige Betrachtung der eben beschriebenen Methodenarten abgebildet und müssen deshalb an dieser Stelle nicht gesondert betrachtet werden. Klasse `A` kann auf die beiden Methoden zugreifen, indem sie ein Objekt der Klasse `B` instanziiert. Dieser Aufruf setzt allerdings sowohl die Sichtbarkeit als auch die Verfügbarkeit von `B` voraus, was dem ursprünglichen Gedanken eines reinen komponentenbasierten Ansatzes widerspricht. Für verteilte Hardware/Software-Systeme gilt dies in besonderem Maße, da zum Zeitpunkt der Softwareentwicklung noch nicht feststehen muss, wie die zugrunde liegende Hardware-Plattform aufgebaut ist, auf welcher Hardware-Ressource die Software-Funktionalität ausgeführt werden soll, über welche Kommunikationsstrukturen und Kommunikationsmechanismen kommuniziert werden soll oder wie überhaupt die Konfiguration des Systems sein wird.

Um dies zu verdeutlichen, existieren nach [120] drei grundlegende Anforderungen an Komponenten, die entsprechend auf Software-Komponenten übertragen werden können:

- Unabhängige Verteilung (engl.: *Deployment*)
- Komposition durch Dritte
- Kein öffentlich beobachtbarer Zustand

Diese Eigenschaften müssen demzufolge auf einer zusätzlichen Ebene zugesichert werden.

5.2.2 Integrations- und Verwaltungsschicht

Zur Verwirklichung eines komponentenbasierten Entwurfs von Hardware/Software-Systemen bedarf es der Möglichkeit, die modellierten Software-Komponenten frei und konfigurierbar auf die zur Verfügung stehenden Hardware-Ressourcen abbilden zu können. Diese Eigenschaft deckt sich z.B. mit den grundlegenden Anforderungen, die an den AUTOSAR[2]-Standard gestellt werden, in dem eine verwaltende Mittelschicht für die Interaktion der einzelnen Software-Komponenten zur Laufzeit verantwortlich ist. In [67] wird ein Ansatz zur Simulation AUTOSAR-konformer Software-Komponenten mit SystemC vorgestellt. Dabei wird allerdings angenommen, dass Software-Komponenten bereits als generierter Quellcode der Funktionalität zur Verfügung stehen und somit lediglich die verwaltende Mittelschicht erzeugt werden muss. In dieser Arbeit wird ein allgemeiner Ansatz vorgestellt, wie Software-Komponenten durch die Anwendung eines Schichtenmodells ausführbar gemacht werden, indem ein entsprechendes Modell in SystemC generiert wird. Der strukturelle Aufbau dieses Schichtenmodells ist in Abbildung 5.7 dargestellt.

Vor der Erklärung des mehrstufigen Schichtenmodells sei an dieser Stelle erwähnt, dass es bei den beiden Klassen `A` und `B` um Softwareklassen handelt, die entweder eine Angebots- oder eine Bedarfsschnittstelle implementieren. Der Ansatz des mehrstufigen Schichtenmodells ist aber davon unabhängig, ob die Klasse einen einzigen Schnittstellentyp oder eine ebenfalls mögliche Mischform implementiert. Die anzuwendenden Mechanismen bei der Transformation beziehen sich jeweils auf den Schnittstellentyp und können deshalb kombiniert und mehrfach durchgeführt werden.

Im Schichtenmodell werden beide Klassen `A` und `B` jeweils in einem eigenen SystemC-Modul gekapselt, wobei die Schnittstellensemantik an entsprechende SystemC-Ports delegiert wird (in Abbildung 5.7 durch einen Pfeil angedeutet). Zur Realisierung einer Bedarfsschnittstelle wird ein `sc_port` verwendet, für eine Angebotsschnittstelle dementsprechend ein `sc_export`. Um die Schnittstellensemantik einzuhalten und eine Typisierung der Ports vornehmen zu können, muss eine Modulschnittstelle `SC_B_IF` erzeugt werden, die sowohl die Anforderungen einer SystemC-Schnittstelle (engl.: SystemC-Interface) erfüllt, als auch von der Softwareschnittstelle `B_IF` ableitet. Innerhalb des jeweiligen SystemC-Moduls wird die gekapselte Klasse instanziiert. Durch den Verweis auf die Instanz werden Methodenaufrufe von der gekapselten Klasse an die Ports bzw. von den Ports an die Methoden der gekapselten Klasse weitergeleitet.

Innerhalb der Klasse `A` setzt das SystemC-Modul sich selbst als Schnittstelle nach außen. Das ist deshalb möglich, weil das SystemC-Modul die Modulschnittstelle implementiert, die wiederum von der Softwareschnittstelle abgeleitet ist. Der Methodenaufruf in Klasse `A` kann vollkommen unverändert bleiben, somit ist die Weiterleitung über das SystemC-Modul und dessen Ports transparent. Dies ist beim Entwurf eine wichtige Anforderung, da ansonsten die Implementierung von Klasse `A` angepasst werden

[2]AUTomotive Open System ARchitecture, http://www.autosar.org

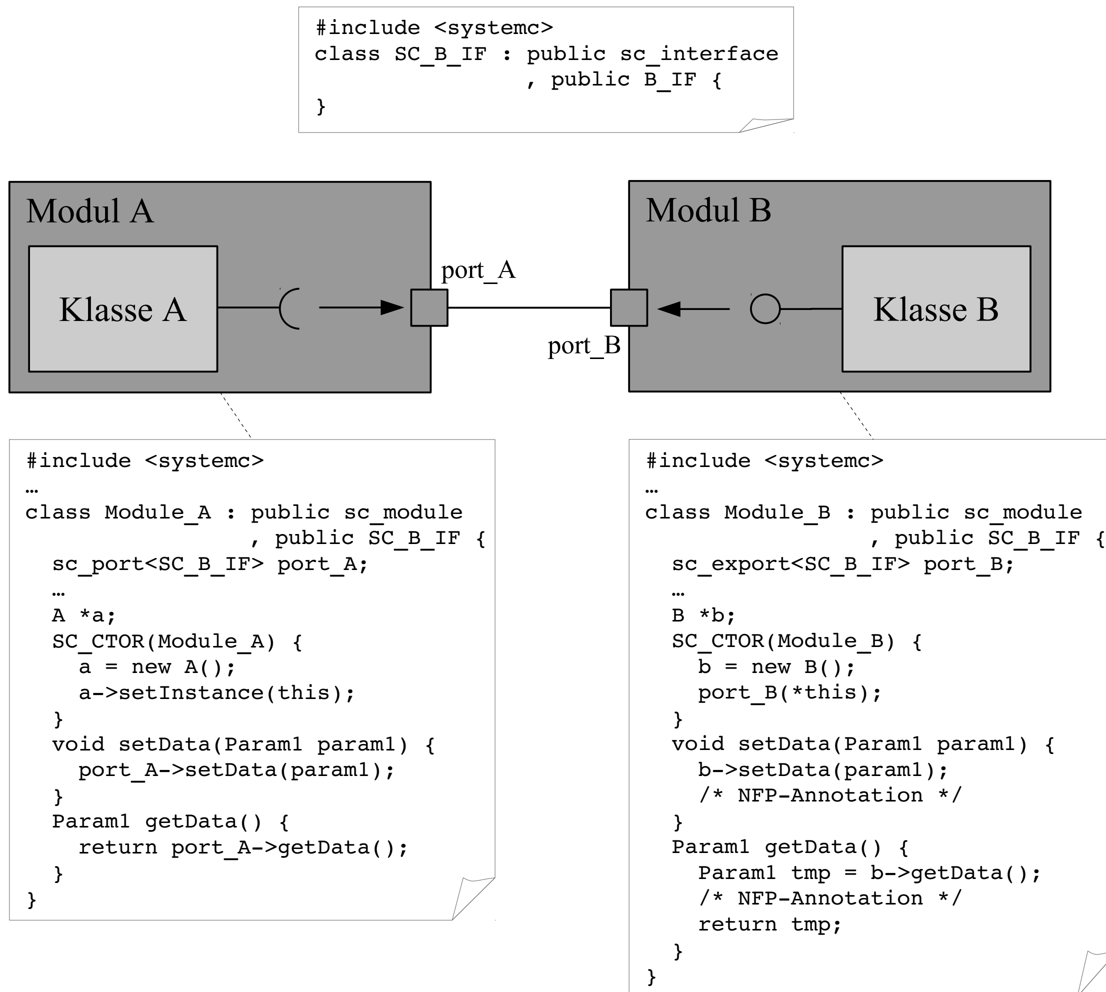


Abbildung 5.7: Schichtenmodell für die Integration von Softwarekomponenten

müsste und es somit der zuvor geforderten Bedingung der freien Komponierbarkeit von Komponenten widersprechen würde.

Analog zur aufrufenden werden die Methodenaufrufe der aufgerufenen Software-Komponente durch die Ports an die Methoden der gekapselten Klasse `B` weitergeleitet. Hier bietet das SystemC-Modul die Implementierung der Angebotsschnittstelle über `port_B` an. Innerhalb der Methoden des SystemC-Moduls können Annotationen nicht-funktionaler Eigenschaften vorgenommen werden, z.B. Zeitannotationen mittels der SystemC `wait()`-Methode. In aufgerufenen Software-Komponenten kann damit die Zeit ausgedrückt werden, die von Klasse `B` benötigt wird, um die angebotene Funktionalität zu erfüllen. Erst nach Ablauf dieser Zeit kehrt die Methode zurück und die aufrufende Komponente kann im Kontrollfluss fortfahren.

Bei den bisher vorgestellten Aufrufbeziehungen handelt es sich um synchrone Aufrufe. Wird ein asynchroner Aufruf verlangt, muss die Methodenweiterleitung innerhalb eines SystemC-Threads implementiert sein. Dieser SystemC-Thread ist so lange blockiert, bis der entsprechende Methodenaufruf über `port_B` erfolgt und dies durch ein SystemC-Event angezeigt wird. Hier sei auf die SystemC-Grundlagen in Abschnitt 2.3.1 und insbesondere auf die Notifizierungsmechanismen hingewiesen.

Durch die zusätzliche Verwendung von Semaphoren kann der Zugriff auf die angebotenen Methoden geschützt werden, wodurch ein ungewollter gleichzeitiger Zugriff verhindert werden kann.

Durch eine Instanziierung der SystemC-Module `Module_A` und `Module_B`, sowie die Bindung der Ports `port_A` und `port_B` kann bereits auf dieser Ebene eine Simulation erfolgen, vorausgesetzt die aufrufende Komponente enthält einen SystemC-Thread. Zu beachten ist hier jedoch, dass die Komponenten direkt miteinander kommunizieren und deshalb die Simulation ohne den Einfluss der zugrunde liegenden Hardware-Plattform durchgeführt wird. Sie erfolgt also auf der Abstraktionsebene kommunizierender Prozesse (engl.: *Communicating Processes (CP)*) oder falls ein geschätzter Zeitverbrauch annotiert ist auf der Ebene kommunizierender Prozessen mit Zeitverhalten (engl.: *Communicating Processes with Timing (CPT)*). Die Simulation auf dieser Ebene eignet sich deshalb hauptsächlich für die Überprüfung der funktionalen Korrektheit, sowie für die grobe Analyse nicht-funktionaler Eigenschaft, wie z.B. die Einhaltung von Budgets für Zeit- und Energieverbrauch.

5.2.3 Hardware-Schicht

Soll die Genauigkeit der Simulation erhöht werden, muss der Einfluss der zugrunde liegenden Hardware-Plattform in die Simulation integriert werden. Oftmals impliziert eine Erhöhung der Genauigkeit automatisch eine detailliertere Modellierung und damit eine niedrigere Abstraktionsebene des Systems, was zu einer langsameren Simulationsgeschwindigkeit führt. Um dies weitestgehend zu verhindern, sollten die nicht-funktionalen Eigenschaften der Hardware-Plattform möglichst abstrakt in die Simulation eingebunden werden können. SystemC bietet mit der TLM-2.0-Bibliothek einen standardisierten und abstrakten Mechanismus, Elemente einer Hardware-Plattform zu modellieren und zu simulieren. Für Details der TLM-Bibliothek sei an dieser Stelle auf Abschnitt 2.3.1.4 verwiesen. Für eine virtuelle Ausführungsplattform bedeutet das, dass für die Modellierung der Hardware-Plattform ein zusätzliches Schichtenmodell eingebunden werden muss. Dieses dient der integrierten Repräsentation des Hardware/Software-Systems und ist in Abbildung 5.8 dargestellt.

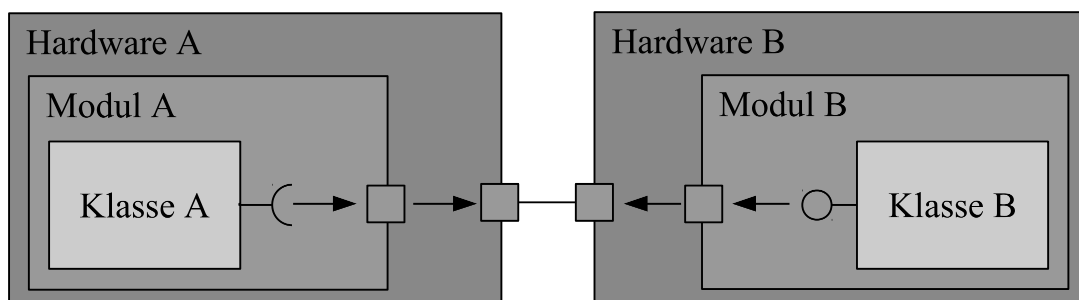


Abbildung 5.8: Schichtenmodell für Hardware/Software-System

Der SystemC-Standard TLM-2.0 basiert auf dem Konzept, dass zu kommunizierende Daten nicht wie auf *RT-Ebene* in einzelnen Bit- bzw. Bytefeldern übertragen werden,

sondern als komplexer Datentyp in Form einer Klasse bzw. einer *Transaktion* übertragen werden. Die einzelnen Variablen werden in einer generischen Klasse gekapselt und definieren somit die unterschiedlichen Eigenschaften und abstrakten Bedeutungen der Kommunikation. So existieren unter anderem eine Variable für die Adressierung bei der Kommunikation über einen gemeinsamen Speicherbereich bzw. einem geteilten Medium, eine Variable zur Definition der Art des Zugriffs und ein Zeiger auf die eigentlich zu übertragenden Daten. Da die Kommunikation über TLM-Schnittstellen in einer Simulation und damit in einem Adressbereich stattfindet, kann der Transport dieser generischen Klasse von der Quelle (engl.: *TLM-Initiator*) zur Senke (engl.: *TLM-Target*) über eine Zeigerzuweisung realisiert werden. Im Gegensatz zu einer ansonsten notwendigen Kopieroperation für alle Variablen führt der Transport eines einzigen Zeigers in der Regel abhängig von der Komplexität des modellierten Systems zu einer deutlichen Beschleunigungen der Simulation [34].

Für das Hardware/Software-Schichtenmodell ist entscheidend, auf welches Hardware-Element welche Software-Komponente abgebildet wird. Es müssen hierbei neben der Modellierung der Funktionalität und der Hardware-Plattform also auch Informationen und Eigenschaften aus der Modellierung des Deployments (siehe Abschnitt 5.1.3) berücksichtigt werden. Quellcode 5.1 stellt die TLM-Struktur eines Hardware-Element `Hardware_A` vor, auf das eine aufrufende Komponente `Module_A` abgebildet wird. Da die Initiative zur Kommunikation von demjenigen Hardware-Element ausgeht, auf das eine aufrufende Software-Komponente abgebildet wird, ist ein *TLM-Initiator-Port* zur Verbindung mit demjenigen Hardware-Element notwendig, auf das die aufgerufene Komponente abgebildet wird. Weiterhin muss die eben angesprochene Software-Komponente `Module_A` instanziiert und deren Port `port_A` zur Implementierung des Methodenaufrufs der Bedarfsschnittstelle von Klasse `A` an das Hardware-Modul weitergeleitet werden. Durch die Vererbungsbeziehung von `Hardware_A` zur Modulschnittstelle `SC_B_IF` ist auch diese Weiterleitung transparent.

Die größte Herausforderung bei der Erstellung des Schichtenmodells zur Repräsentation des Hardware/Software-Systems ist jedoch die Transformation eines am Aufruf von Methoden orientierten Kommunikationsmechanismus in einen Mechanismus, der an einer Kommunikation von Hardware-Komponenten orientiert und deswegen auf die Übertragung eines Datenstroms ausgerichtet ist. Dies hat zur Folge, dass jeder Methodenaufruf und vor allem dessen Signatur zur Übertragung über TLM-Schnittstellen serialisiert werden müssen. Genau wie bei der Auswahl des entsprechenden TLM-Ports hängt die Transformation allerdings davon ab, ob durch die Methode eine Bedarfs- oder eine Angebotsschnittstelle implementiert wird. Da im Fall von Klasse `A` eine Bedarfsschnittstelle implementiert wird, muss in beiden Methoden `setData()` und `getData()` eine TLM-Transaktion initiiert werden. Dazu wird zunächst im *TLM-Initiator* ein generisches TLM-Datenpaket `tlm_generic_payload` instanziiert (siehe Zeile 14 in Quellcode 5.1). Verfügt die Methode wie im Fall von `setData()` über keinen Rückgabewert, wird der TLM-Befehlsanzeiger auf einen Schreibbefehl `TLM_WRITE_COMMAND` gesetzt, mit Rückgabewert – wie im Fall der Methode `getData()` – auf einen Lesebefehl `TLM_READ_COMMAND`. Als TLM-Adresse wird im Transformationsschritt eine eindeutige Identifizierung der Zielmethode gewählt. Durch diese Identifizierung erfolgt bei der

```
#include <tlm>
...
class Hardware_A : public sc_module, public SC_B_IF {
  tlm_utils::simple_initiator_socket<Hardware_A> initSocket;
  ...
  Module_A *module_A;
  SC_CTOR(Hardware_A) {
    module_A = new Module_A(...);
    module_A->port_A(*this);
  }
  ...
  void setData(Param1 param1) {
    unsigned char* tmpPayload = new unsigned char(sizeof(Param1));
    memcpy(tmpPayload, param1, sizeof(Param1));
    tlm::tlm_generic_payload payload;
    payload.set_command(TLM_WRITE_COMMAND);
    payload.set_address(1);
    payload.set_data_ptr(tmpPayload);
    ...
    initSocket->b_transport(payload, delay);
    if (delay.value() > 0)
      wait(delay);
    ...
  }
  Param1 getData() {
    unsigned char tmpPayload[1];
    tlm::tlm_generic_payload payload;
    payload.set_command(TLM_READ_COMMAND);
    payload.set_address(2);
    payload.set_data_ptr(tmpPayload);
    ...
    initSocket->b_transport(payload, delay);
    if (delay.value() > 0)
      wait(delay);
    Param1* returnPtr = reinterpret_cast<Param1*>(payload.get_data_ptr());
    ...
    return Param1(*returnPtr);
  }
}
```

Quellcode 5.1: Hardware-Schicht der aufrufenden Software-Komponente

Dekodierung im TLM-Target eine eindeutige Zuordnung des Ports zur Weiterleitung des Methodenaufrufs an die aufgerufene Software-Komponente.

Die durch die TLM-Transaktion übertragenen Nutzdaten werden im Datenzeiger referenziert. Da durch die TLM-Adressierung bereits die aufzurufende Methode eindeutig identifiziert werden kann, müssen die Daten serialisiert werden, die während der Ausführung der Methode gebraucht werden. Handelt es sich um eine Lesemethode, muss zuerst ein Speicherbereich für den Rückgabewert reserviert und ein Zeiger auf diesen Speicherbereich übergeben werden. Bei einem Schreibbefehl entfällt diese

Notwendigkeit. Anschließend wird die vollständige Liste der Methodenparameter per Referenzierung übergeben (siehe Zeilen 13 bzw. 17). Für die Berücksichtigung und Berechnung eines anfallenden Zeitverbrauchs für die Kommunikation über die TLM-Verbindung kann auch noch mittels der Methode `set_data_length()` die Gesamtgröße der zu übertragenden Parameterdaten für die Transaktion gesetzt werden. Die TLM-Transaktion wird durch den Aufruf der Methode `b_transport()` und der Übergabe des generischen TLM-Datenpakets initiiert. Nach der Rückkehr dieser Methode kann die als `delay` übergebene Zeit gewartet werden, bevor der Kontrollfluss weiter abgearbeitet werden kann. Besitzt die aufrufende Methode der Software-Komponente einen Rückgabewert, ist dieser nach Rückkehr der `b_transport()`-Methode im dafür reservierten Speicherbereich enthalten und kann deshalb nach einer entsprechenden Typkonvertierung (siehe Zeilen 34 bis 36) zurückgegeben werden.

```
1  #include <tlm>
2  ...
3  class Hardware_B : public sc_module, public SC_B_IF {
4    tlm_utils::simple_target_socket <Hardware_B> targetSocket;
5    sc_port<SC_B_IF> port_MB;
6    ...
7    Module_B *module_B;
8    SC_CTOR(Hardware_B) {
9      module_B = new Module_B(...);
10     port_MB(module_B->port_B);
11     targetSocket.register_b_transport(this, &Hardware_B::b_transport);
12   }
13   ...
14   void b_transport(tlm::tlm_generic_payload& transaction,
        sc_core::sc_time& delay) {
15     switch(transaction.get_address()) {
16     case 1:
17       unsigned char* ptrs = transaction.get_data_ptr();
18       Param1* param1 = reinterpret_cast<Param1*>(ptrs[0]);
19       port_MB->setData(*param1);
20       transaction.set_response_status(tlm::TLM_OK_RESPONSE);
21       break;
22     case 2:
23       unsigned char* ptrs = transaction.get_data_ptr();
24       Param1* returnValue = reinterpret_cast<Param1*>(ptrs[0]);
25       returnValue = new Param1(port_MB->getData());
26       transaction.set_data_ptr(reinterpret_cast<unsigned
            char*>(returnValue));
27       transaction.set_response_status(tlm::TLM_OK_RESPONSE);
28       break;
29     }
30   }
31 }
```

Quellcode 5.2: Hardware-Schicht der aufgerufenen Software-Komponente

Im Gegensatz zum *TLM-Initiator* muss das Hardware-Element, auf das die auf-

gerufene Komponente abgebildet wird, einen *TLM-Target-Port* besitzen. Durch die Verbindung von TLM-Initiator-Port und TLM-Target-Port werden direkt kommunizierende Hardware-Elemente modelliert. TLM-Modelle werden jedoch auch häufig zur Kapselung von IP-Hardware-Komponenten verwendet, da die Schnittstelle im TLM-2.0-Standard eindeutig spezifiziert ist. Die TLM-Ports der Hardware-Elemente des Schichtenmodells können deshalb auch jeweils mit existierenden TLM-Kommunikationsbibliotheken oder Hardware-Komponenten verbunden werden, um sowohl komplexe Kommunikationsmechanismen zu implementieren, als auch Peripherie- und Speicherbausteine einbinden zu können. Zur Weiterleitung der aufgerufenen Methoden an die gekapselte Software-Komponente muss ein mit der Modulschnittstelle typisierter Port `port_MB` existieren (siehe Zeile 12 in Quellcode 5.2). Der Aufruf der Methode `b_transport()` eines durch TLM-Ports verbunden TLM-Initiators wird nach der Registrierung der `b_transport()` als *Rückruf*-Methode (engl.: *Callback Method*) innerhalb des TLM-Target ausgeführt (siehe Zeile 11). Dabei wird zuerst die Dekodierung der TLM-Adresse durchgeführt, um die aufzurufende Methode der abgebildeten Software-Komponente identifizieren zu können (siehe Zeile 15). Im Falle einer aufgerufenen Methode ohne Rückgabewert werden die Übergabeparameter der Methode aus dem TLM-Datenfeld extrahiert und der Methodenaufruf inklusive der extrahierten Parameter an den zugeordneten Port der Software-Komponente weitergeleitet (siehe Zeilen 18 und 19). Handelt es sich um eine Methode mit Rückgabewert, wird zuerst der für den Rückgabewert reservierte Speicherbereich extrahiert, damit dieser zum Speichern des Rückgabewerts der an die Software-Komponente weitergeleiteten Methode benutzt werden kann (siehe Zeilen 24 und 25). Der nach Rückkehr des Methodenaufrufs mit gültigen Daten gefüllte Speicherbereich wird der TLM-Transaktion vor der Rückkehr zum TLM-Initiator übergeben (siehe Zeile 26), sodass dieser den Rückgabewert extrahieren und ihn dann wiederum als Rückgabewert der gekapselten aufrufenden Software-Komponente übergeben kann.

Mithilfe des eben dargestellten mehrstufigen Schichtenmodells ist ein transparentes Aufrufen von Schnittstellenmethoden sowohl unter Berücksichtigung von Anforderungen der Komponenten anhand funktionaler Gesichtspunkte als auch Einflüssen der Hardware auf nicht-funktionale Eigenschaften, wie z.B. Zeitverhalten und Energieverbrauch, gewährleistet. Dazu wurde eine Methodik entwickelt, um Modellierungskonzepte und Techniken von SystemC dazu zu benutzen, Software-Komponenten kapseln und eine Systemverwaltung integrieren zu können, die sowohl einen definierten Ablaufplan für die Ausführung der Funktionalität, als auch ein Energiemanagement innerhalb der Simulation realisiert.

5.3 Generierung eines ausführbaren Systemmodells

Bestehende Ansätze zur simulativen Analyse, Verifikation und Evaluierung eingebetteter Hardware/Software-Systeme basieren in der Regel auf einer möglichst frühzeitigen Verfügbarkeit der zugrunde liegenden Simulationsmodelle. Nicht zuletzt deshalb spielen unterstützende Ansätze zu einer weitestgehend *automatischen Generierung* von

Simulationsmodellen bzw. *virtuellen Prototypen* eine entscheidende Rolle während des Entwurfsprozesses. Zum einen sollen automatisierte Generierungsprozesse als Alternative zu manuellen Implementierungen die Integration von zufälligen oder systematischen Fehlern verhindern – zumindest unter der Voraussetzung, dass der Generierungsprozess selbst als fehlerfrei gilt. Zum anderen soll die Komplexität minimiert und die Skalierbarkeit dadurch garantiert werden, dass sich die Instrumente der Generierung, z.B. die Regeln einer transformationsbasierten Generierung, theoretisch auf beliebig komplexe Systeme und beliebig oft anwenden lassen. Somit werden die regelmäßig vorkommenden Elemente und Strukturen innerhalb eines modellierten Systems für wiederkehrende und mitunter verifizierte Generierungsprozesse ausgenutzt. Allgemein betrachtet können sich Simulationsmodelle dabei unter anderem anhand der folgenden Eigenschaften unterscheiden:

- vorkommende Abstraktionsebenen
- verwendete Modellierungssprache
- Detaillierungsgrad einzelner Modelle
- modellierte Aspekte und Sichtweisen auf das Gesamtsystem

Im Falle der in dieser Arbeit betrachteten eingebetteten Hardware/Software-Systeme können hauptsächlich zwei Entwurfsabläufe zur Generierung eines *ganzheitlichen Simulationsmodells* [140] identifiziert werden, die in mehrere der eben genannten Kategorien eingeordnet sind. Das Hauptaugenmerk während aller Generierungsschritte liegt auf der Bewahrung der Modellsemantik.

Dem Entwurfsablauf, der auf hoher Abstraktionsebene bzw. Systemebene beginnt und in Abbildung 5.9 mit dem Begriff *Top-Down-Entwurf* bezeichnet ist, liegt ein UML-Modell zugrunde, das aus Teilmodellen für die im System vorhandenen Software-Komponenten, der ausführenden Hardware-Plattform und der Abbildung von Funktionalität auf die Plattform besteht. Für die Modellierung eines ganzheitlichen Systemmodells wird die in Abschnitt 5.1 beschriebene Modellierungsmethodik verwendet. Stehen dabei Informationen über die tatsächliche Implementierung zur Verfügung, kann zusätzlich noch ein *Timing-Modell* integriert werden, welches das Zeitverhalten der Software-Komponenten auf der Zielarchitektur repräsentiert. Weiterhin kann die Hardware-Plattform durch Angaben der Leistungsaufnahme in verschieden Betriebsmodi und ein damit verknüpfte Ablaufplanung von geteilten Ressourcen angereichert werden. Dieses ganzheitliche Systemmodell wird im weiteren Verlauf des Generierungsprozesses durch mehrere Transformationsschritte auf ein ausführbares Simulationsmodell abgebildet. Die dabei enthaltenen Verfeinerungsschritte bis hin zu einem Code-nahen Modell, das die Struktur von Simulations-Code in Form eines *abstrakten Syntaxbaums* (engl.: *Abstract Syntax Tree (AST)*) widerspiegelt, sind hauptsächlich Gegenstand der beiden Abschnitte 5.3.1 und 5.3.2. Die Generierung des ausführbaren Simulations-Code aus einer abstrakten Repräsentation der Quelltextstruktur erfolgt durch ein Verfahren, das auf Generierungsmustern (engl.: *Template*) basiert und in Abschnitt 5.3.3 betrachtet wird.

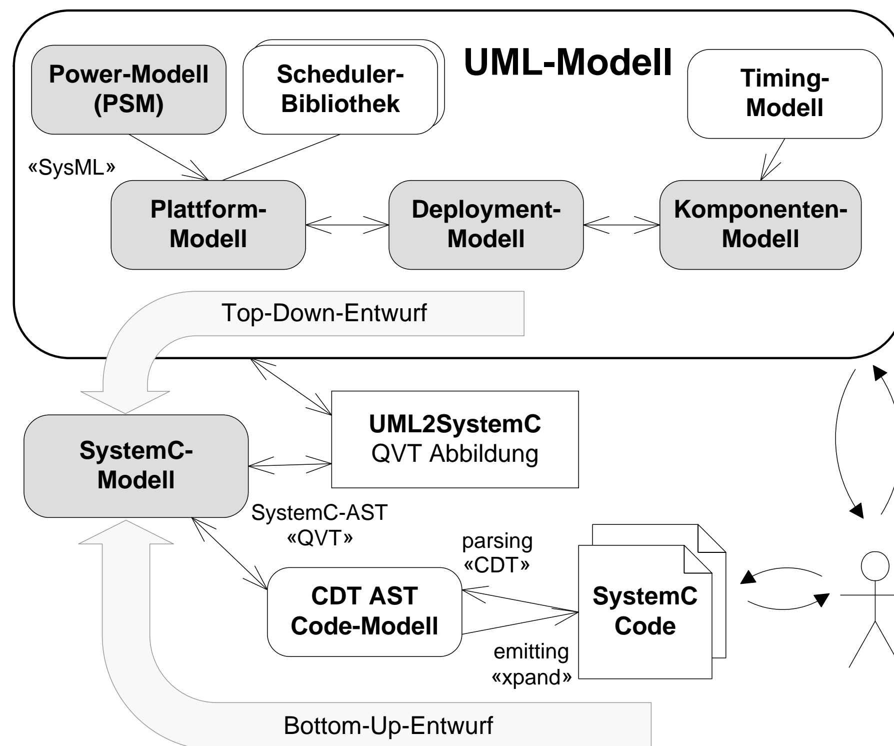


Abbildung 5.9: Übersicht der angewandten Modell-Transformationen

Die wichtigste Grundlage für Transformationen zwischen unterschiedlichen Modellen ist die Definition der zugrunde liegenden Metamodelle für die verwendeten Quell- und Zielmodelle [71]. Durch die Formulierung von *Abbildungsregeln* werden ein oder mehrere Elemente des Quellmodells in ein oder mehrere Elemente des Zielmodells überführt. Die konkrete Umsetzung der Abbildungsregeln erfolgt mittels dafür geeigneter Transformationswerkzeuge, die auf konkrete Modelle, also Instanzen der Metamodelle angewandt werden. Einige dieser Werkzeuge basieren auf der Methodik der *Query-View-Transformations (QVT)*, die eine Sprache für Modell-zu-Modell-Transformationen darstellt und im Grundlagen-Abschnitt 2.2.4 näher beschrieben ist.

Wegen der weniger komplexen Programmierung und der fehlenden Notwendigkeit von bidirektionalen Abbildungen wird in dieser Arbeit ausschließlich QVT-O verwendet. Die Entwicklungsumgebung *Eclipse* verfügt über ein eigenes Modell-zu-Modell-Transformationswerkzeug, das auf Basis von QVT-O arbeitet. Im Anhang A.1 sind einige Beispieltransformationen in QVT-O enthalten.

5.3.1 Abbildung des Systemmodells auf ein Komponentenmodell

Die *komponentenbasierte Entwicklung* stellt ein Paradigma dar, das weitestgehend an der eindeutigen Spezifikation von Schnittstellen orientiert ist und deswegen eine klare Trennung bei der Entwicklung und Konfiguration der Hardware und der darauf auszuführenden Software erlaubt. Da SystemC zur Ausführung des Systemmodells benutzt wird und dieser Sprache ein Komponentenmodell zugrunde liegt, wird das in UML modellierte ganzheitliche Systemmodell durch Transformationsschritte auf ein an

SystemC angelehntes Komponentenmodell abgebildet. Die Struktur des daraus resultierenden Komponentenmodells wird durch das in Abbildung 5.10 gezeigte Metamodell definiert, das eine Spezialisierung des allgemeinen SystemC-Komponentenmodell darstellt. Dieses Metamodell ist im *Eclipse Modeling Framework (EMF)* der Entwicklungsumgebung *Eclipse* unter Verwendung des EMF-internen *Ecore*-Metamodells modelliert. Zur eindeutigen Identifizierung wird die SystemC-spezifische Namensgebung der Modellelemente beibehalten, soweit dies möglich ist.

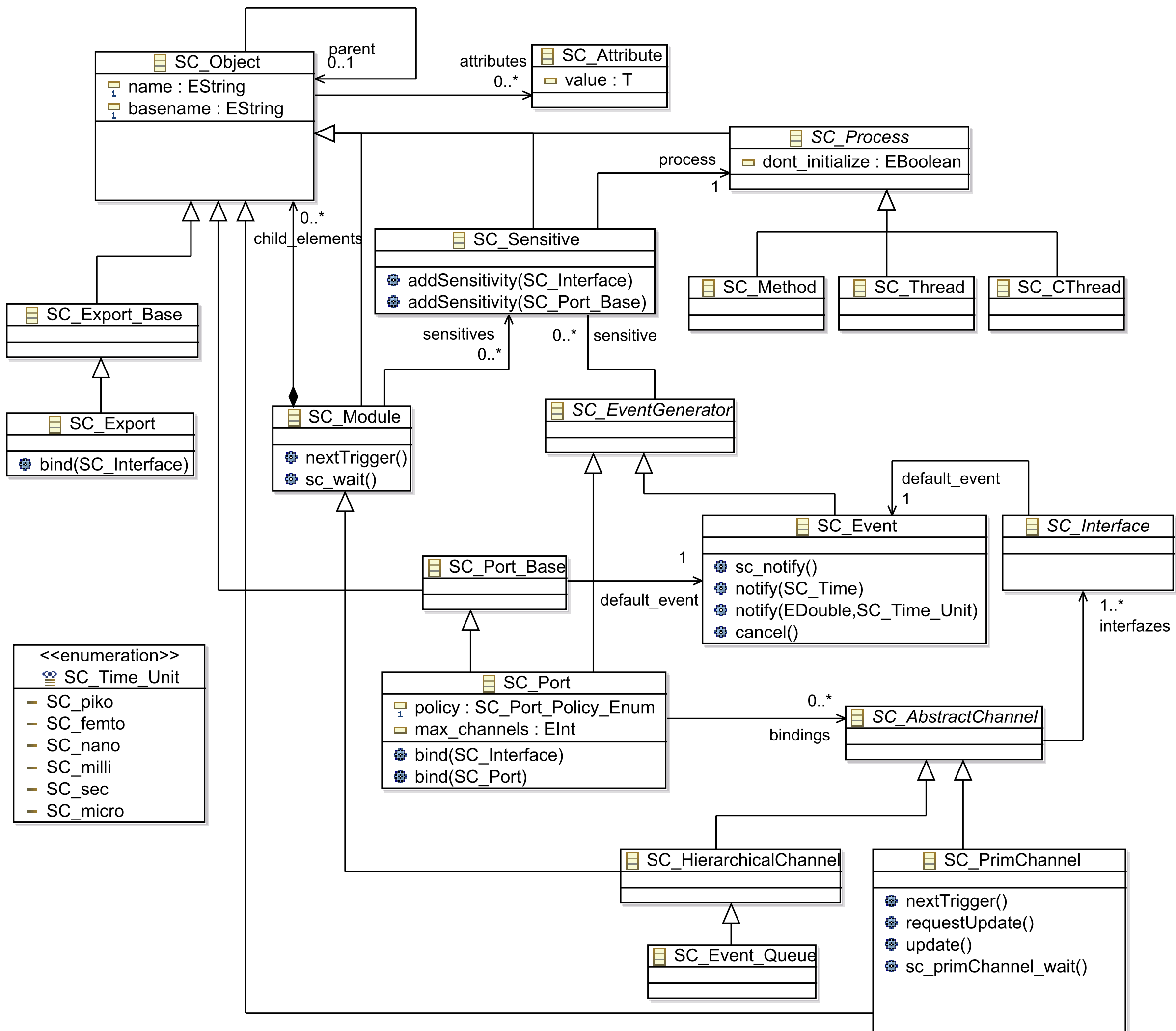


Abbildung 5.10: Metamodell des Komponentenmodells basierend auf SystemC

Wie im SystemC-Komponentenmodell bildet die Klasse *SC_Object* das Hauptelement des Metamodells. Von dieser Klasse sind sowohl die Klassen *SC_Module*, die die Komponenten im Modell repräsentiert, als auch die Klassen *SC_Port_Base* und *SC_Export_Base* bzw. *SC_Port* und *SC_Export* zur Darstellung der Kommunikationspunkte der Komponenten abgeleitet. Instanzen der Klasse *SC_Object* können in anderen Klassen des Typs *SC_Object* enthalten sein, was einen hierarchischen Komponentenentwurf erlaubt. Das in einer Komponente enthaltene bzw. implementierte Verhalten wird in Form

von SystemC-Methods oder SystemC-Threads mithilfe einer Vererbungsbeziehung als Klasse des Typs *SC_Process* dargestellt. Die Klasse *SC_Process* ist dabei ebenfalls von `SC_Object` abgleitet. Anhand dieser wenigen Metaklassen lässt sich bereits die statische Struktur des UML-Systemmodells in ein Komponentenmodell transformieren.

Für die Abbildung auf eine komponentenbasierte virtuelle Ausführungsplattform, wie sie in Abschnitt 5.2 beschrieben ist, muss zusätzlich eindeutig zwischen der Typenebene und der Instanzenebene unterschieden werden. Im Gegensatz zur Typenebene, die lediglich die statische Struktur des Komponentenmodells modelliert, werden auf der Instanzenebene

- die Anzahl der Instanzen eines jeden Typs,
- der hierarchische Aufbau aller Komponenten und somit des Gesamtsystems und
- die Verbindungen der Komponenten untereinander

definiert. Dabei existieren in SystemC aufgrund der Implementierung als C++-Bibliothek mehrere Varianten der Instanziierung und Verbindung von Komponenten bzw. SystemC-Module. Das Metamodell der Instanzenebene des Komponentenmodells ist in Abbildung 5.11 visualisiert. Instanziierungen und Verbindungen innerhalb einer Komponente modellieren eine hierarchische Komponente und können somit innerhalb des Konstruktors der jeweiligen Modulklasse vorgenommen werden. Dies führt zu einer Abbildung auf das Element *InnerAssembly*. Instanziierungen und Verbindungen, die auf oberster Ebene modelliert werden, z.B. in der SystemC-Startmethode `sc_main()`, werden auf Elemente des Typs *Assembly* abgebildet. Die Verbindungen werden im Metamodell anhand ihrer spezifischen Verbindungsart unterschieden. Für die Verbindung von SystemC-Ports über Hierarchiegrenzen hinweg, also z.B. eines Ports mit dem Port eines Sub-Moduls, wird der Typ *DelegationBinding* verwendet. Elemente des Typs *PortBinding* verbinden zwei SystemC-Ports unter Verwendung eines abstrakten SystemC-Kanals miteinander, Elemente des Typs *ExportBinding* einen SystemC-Port mit einem SystemC-Export. Die logische Verknüpfung zwischen den Instanzen und deren Verbindungen ist in Elementen des Typs *AssemblyContext* enthalten, die jeweils auf das Modul verweisen, das den bestimmten Port enthält.

Die Methodik des *Transaction-Level Modeling (TLM)* abstrahiert von einer Datenstrom-orientierten Kommunikation zwischen Modulen, indem die zu übertragenden Daten als komplette Transaktion gekapselt werden. Im SystemC-Standard stellt TLM eine Spezialisierung der allgemeinen Schnittstellensemantik dar, indem alle TLM-Ports von den SystemC-Ports *SC_Port* und *SC_Export* abgeleitet sind. Das Metamodell für eine transaktionsorientierte Kommunikation im Komponentenmodell ist in Abbildung 5.12 dargestellt. Im Vordergrund stehen vor allem die Elemente des Typs *simple_initiator_socket* und *simple_target_socket*, da sie sendende bzw. empfangende Ports mit Eins-zu-eins-Verbindungen zwischen Elemente der Hardware-Plattform modellieren (vgl. Modellierung in UML und SysML in Abschnitt 5.1.2). Ports mit Eins-zu-viele-Verbindungen werden analog zur TLM-Methodik in SystemC auf Elemente des Typs *multi_passthrough_initiator_socket* bzw. *multi_passthrough_target_socket* abgebildet.

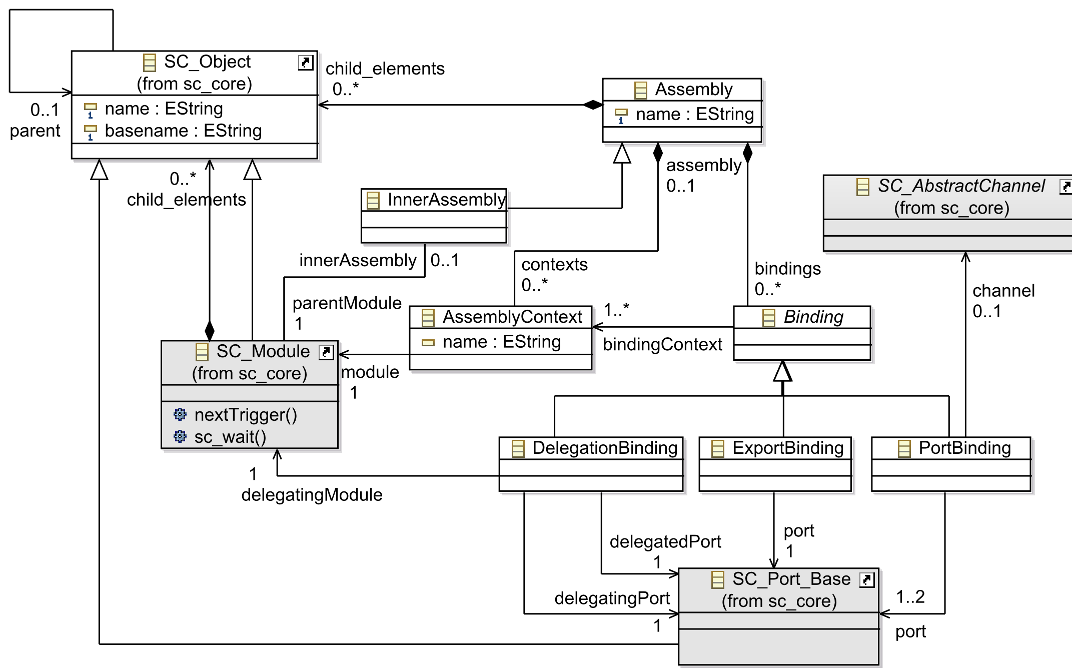


Abbildung 5.11: Metamodell zur Instanziierung und Verbindung von Komponenten

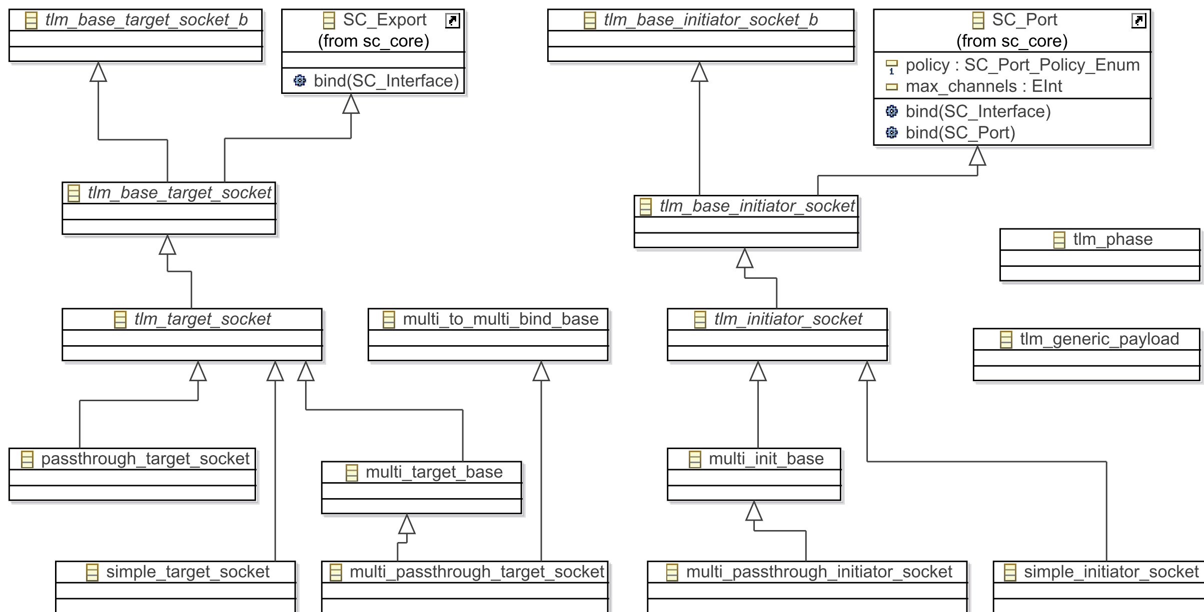


Abbildung 5.12: Metamodell für transaktionsorientierte Kommunikation basierend auf SystemC-TLM

Abbildung 5.13 zeigt das konzeptionelle Vergehen der Modell-zu-Modell-Transformation, die auch auf die Transformation des Systemmodells in das Komponentenmodell angewandt wird. Darin werden Transformationsregeln definiert, die ausgehend von einer Hauptregel als Wurzelelement rekursiv ausgeführt werden, wenn sie auf die in einer Regel spezifizierten Elemente anwendbar sind. Innerhalb einer Regel können entsprechend weitere Regeln angewandt werden, sowie Elemente der Ausgabemodelle

erzeugt werden.

Die Transformation des Systemmodells verfügt neben dem Eingangsmodell in UML über zwei Ausgabemodelle. Das erste Ausgabemodell ergibt das SystemC-orientierte Komponentenmodell, das zweite ein Modell, das eine Verknüpfung zwischen dem Komponentenmodell und dem ursprünglichen UML-Modell herstellt und damit in weiteren Transformationsschritten zur Verfügung steht. Dieses *Link-Modell* wird in Abschnitt 5.3.2 detailliert erklärt, da es für die spätere Generierung des Simulations-Codes eine eindeutige Verbindung zwischen den Elementen des Eingabemodells und des Komponentenmodells herstellt.

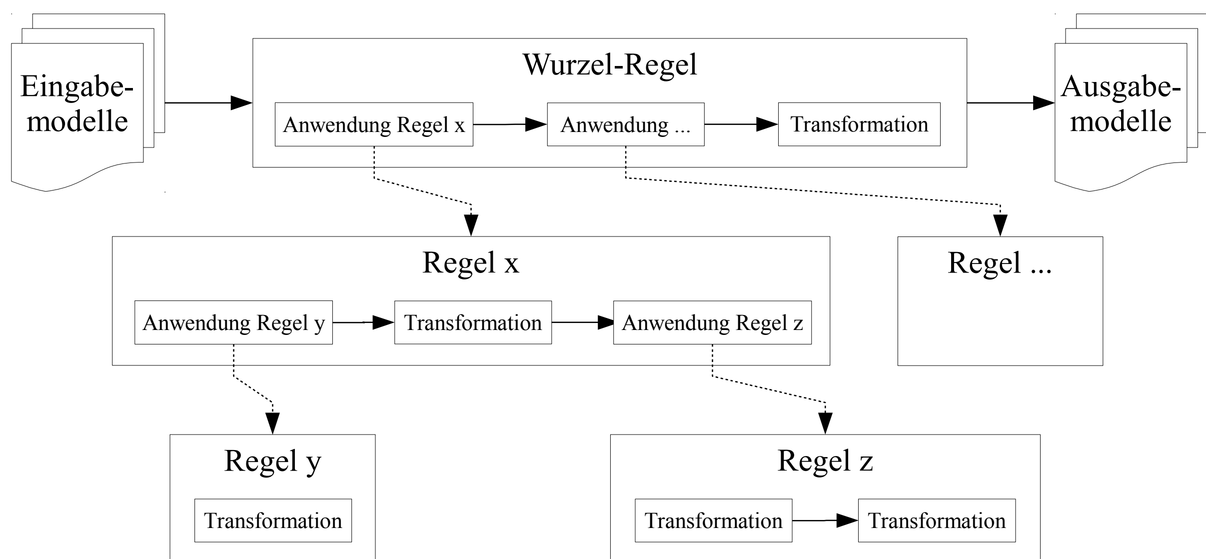


Abbildung 5.13: Konzept von regelbasierten Modell-zu-Modell-Transformationen

Zur Vervollständigung listet Tabelle 5.1 die in diesem Abschnitt beschriebenen Regeln für die Transformation des in UML und SysML modellierten Systemmodells in das SystemC-orientierte Komponentenmodell noch einmal in kompakter Weise auf. Der besseren Übersichtlichkeit wegen wird die genaue Paket-Beschreibung der jeweils verwendeten Stereotypen weggelassen und nur das entsprechende UML-Profil und der verwendete Stereotyp angegeben. Ist kein Kontext angegeben, wird diese Transformation nur dann auf das Element angewandt, wenn keiner der anderen ausgelisteten Kontexte gültig ist.

5.3.2 Transformation des Komponentenmodells in ein AST-Modell

Das Ziel der mehrstufigen Abbildung des Systemmodells ist eine möglichst automatisierte Generierung eines Simulationsmodells in SystemC, das das Systemmodell repräsentiert. Zumeist befinden sich Systemmodellierung und Implementierung des entsprechenden Simulationsmodells auf unterschiedlichen Abstraktionsebenen. Das ist zum einen durch die Anforderung begründet, dass das Systemmodell möglichst einfach und schnell zu modellieren sein soll, sodass die modellbasierte Entwicklung des Systems eine *Aufwandsreduzierung* gegenüber der manuellen Implementierung darstellt. Zum

Tabelle 5.1: Transformationsregeln zur Erstellung des Komponentenmodells

Systemmodell	Stereotypisierung bzw. Kontext	Komponentenmodell
UML-Modell	-	SystemC-Modell
UML-Interface	-	SystemC-Interface
UML-Komponente	-	SystemC-Modul (SW)
UML-Komponente	SysML::Block	SystemC-Modul (HW)
UML-Komponente	MARTE::GaResourcesPlatform	SystemC-Modul (Topebene)
UML-Klasse	-	C++-Klasse
UML-Klasse	MARTE::Scheduler	VEP-Scheduler
UML-Klasse	MARTE::EntryPoint	VEP-Thread
UML-Port	Required Interface	SystemC-Port
UML-Port	Provided Interface	SystemC-Export
UML-Port	TLM::initiator_socket	SystemC-TLM Initiator-Socket
UML-Port	TLM::target_socket	SystemC-TLM Target-Socket
UML-Property	Komponenteninstanz in Komponente	SystemC-Module-Instanz
UML-Konnektor	Verbindung zwischen zwei Komponenteninstanzen	SystemC-Port-Port-Binding
UML-Konnektor	Verbindung zwischen Komponenteninstanz und Komponente	SystemC-Port-Export-Binding
UML-Operation	-	C++-Methode
UML-Operation	MARTE::SwSchedulableResource	SystemC-Thread

anderen sollen auf Systemebene absichtlich noch viele Entwurfsentscheidungen ohne konkrete Kontextinformation der tatsächlichen Implementierung getroffen werden, sodass die Freiheitsgrade einer Parametrisierung des ausführenden Systems weiterhin bestehen bleiben und damit Gegenstand eines anschließenden *Explorationsprozesses* sein können.

Eine geeignete Abstraktionsebene zwischen dem Komponentenmodell aus Abschnitt 5.3.1 und des Simulationsmodells in SystemC bildet die Ebene eines *AST-Modells*. Das AST-Modell repräsentiert ein Code-nahes Modell, indem es die Struktur des Simulationsmodells widerspiegelt, abstrahiert jedoch von der zugrunde liegenden Implementierungssprache. Dies ist auch der Grund, warum viele Generierungsprozesse auf AST-Ebene als Eingabemodell ansetzen und ein oder mehrere Implementierungssprachen als Ausgabemodell unterstützen. Das Metamodell eines AST umfasst dabei in der Regel generische Metaklassen wie *Node* als Basis für die Modellelemente, *Declaration* zur Definition von Klassen, Variablen, Arrays, u.a., *Expression* zur Darstellung von unären oder binären Ausdrücken oder *Statement* zur Repräsentation von Kontrollstrukturen wie Schleifen und Verzweigungen [4]. Ein generisches AST-Metamodell kann zusätzlich mit spezifischen Elementen für die Zielsprache ergänzt werden, um so

eine enge Verzahnung mit der Implementierung zu erreichen, falls dies beabsichtigt ist. Im Fall von C/C++ kann dies z.B. *BaseSpecifier* für Vererbungsbeziehungen oder *TemplateDeclaration* zur Definition von Template-Strukturen und Template-Parametern sein.

In einem AST-Modell, das zur automatischen Generierung von Quelltext verwendet werden soll, müssen allerdings auch Konstrukte repräsentiert werden, die in einem reinen Komponentenmodell abstrahiert werden, um dessen Semantik nicht unnötig komplex werden zu lassen. Komponentenmodelle fokussieren im Allgemeinen auf den Aufbau und die hierarchischen Beziehungen zwischen Komponenten, sowie deren Schnittstellen und damit verbundenen Interaktionspunkte. Typische Konstrukte, wie sie vor allem in objektorientierten Sprachen vorkommen, stehen dabei nicht im Vordergrund, müssen aber bei der Generierung des Simulationsmodells berücksichtigt werden. Zu diesen Konstrukten gehören u.a. Vererbungsbeziehungen, sowie Deklarationen von Variablen, Datenstrukturen und Methoden, die intern in einer Komponente bzw. Klasse definiert sind. Da UML als Modellierungssprache für objektorientierte Implementierungssprachen diese Konstrukte unterstützt, wird ein Modell benötigt, das diese Modellinformationen bei der Transformation in ein Komponentenmodell konserviert, um im Transformationsschritt der AST-Modellgenerierung darauf zurückgreifen zu können. Abbildung 5.14 stellt das Metamodell zur Verknüpfung, das *Link-Modell*, grafisch dar.

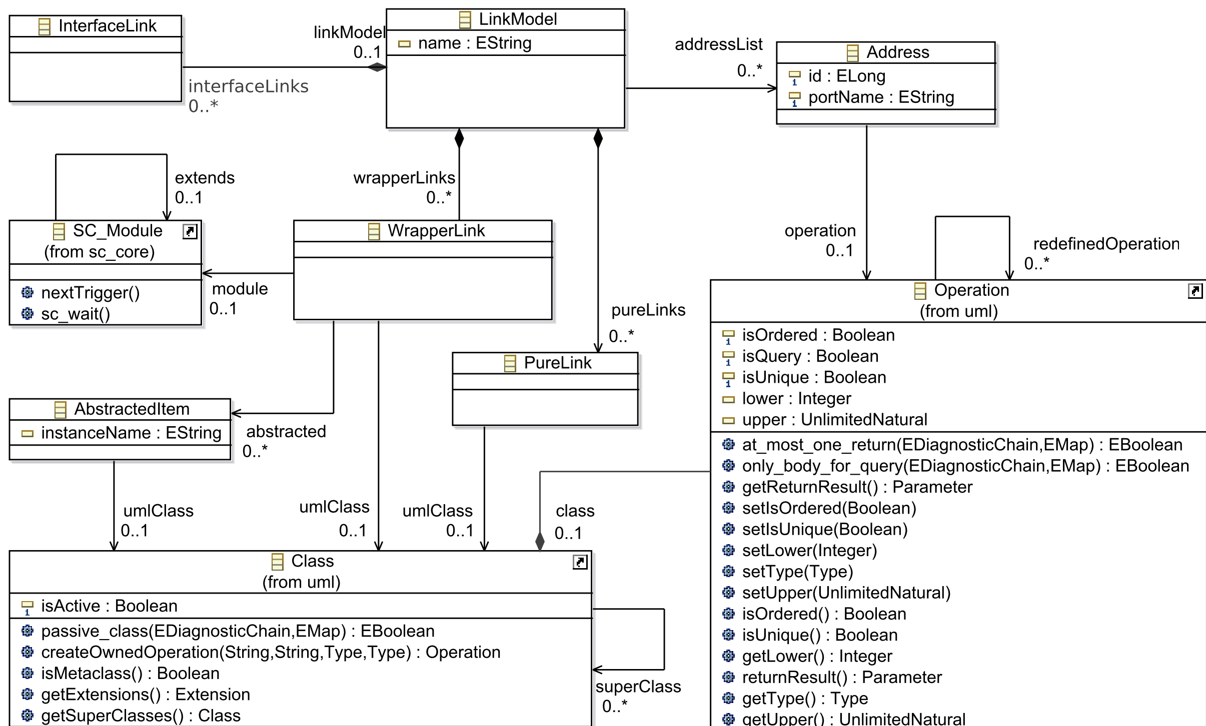


Abbildung 5.14: Verknüpfung zwischen Systemmodell und Komponentenmodell

Hauptsächlich werden durch dieses Link-Modell die Elemente des Systemmodells in Verbindung zum daraus transformierten Komponentenmodell gesetzt. Dazu wird sowohl die Metaklasse *Class* aus dem UML-Paket und die Metaklasse *SC_Module*

des Komponentenmodells referenziert. Die Metaklasse *WrapperLink* verbindet beide Elemente, um festzuhalten, welches Modul aus welcher Klasse des Systemmodells transformiert wird. Weiterhin enthält das Link-Modell eine nummerierte Liste aller in Komponenten vorkommenden Methoden, die in Bedarfs- oder Angebotsschnittstellen enthalten sind. Anhand dieser Liste kann eine eindeutige Adressierung der Methoden vorgenommen werden, was die Weiterleitung von ankommenden und abgehenden Methodenaufrufen an die jeweils zuständigen Ports erlaubt. Dieses Weiterleitungskonzept der Integrationsschicht der virtuellen Ausführungsplattform wurde bereits in Abschnitt 5.2.2 erläutert. Die Verknüpfung von Systemmodell und Komponentenmodell ermöglicht die Transformation von internen Elementen unabhängig von deren Darstellung im Kontext des komponentenbasierten Entwurfs.

Tabelle 5.2 listet die in diesem Abschnitt beschriebenen Regeln für die Transformation des an SystemC-orientierten Komponentenmodells in ein Code-nahes AST-Modell auf oberster Ebene in Abhängigkeit des Link-Modells auf. Bei zusammengesetzten Typen (engl.: *Composite Types*) werden die darin enthaltenen Inhalte in einer separaten Spalte aufgeführt. Elemente des Typs *SystemC-Unit* haben lediglich eine strukturelle Bedeutung innerhalb des Eingabemodells und kennzeichnen Einheiten, die zur besseren Übersichtlichkeit in eigene Abschnitte bzw. Dateien überführt werden können. Dadurch soll die Lesbarkeit und Verständlichkeit des generierten Simulationsmodells entscheidend erhöht werden, um einen iterativen Entwicklungsprozess bzw. Round-Trip-Engineering zu ermöglichen. Der Typ *SystemC-Wrapper* bildet die Integrationsschicht für Software-Komponenten (vgl. dazu Abschnitt 5.2.2). Typen können dabei auch kombiniert werden bzw. ineinander enthalten sein, z.B. ist jedes SystemC-Modul per vorheriger Definition auch eine SystemC-Unit, da im Prozess der Code-Generierung jedes Modul in eine separate Datei transformiert werden soll.

5.3.3 Verbindung zwischen Modellen und Quelltext

Für die Vervollständigung des *Top-Down-Entwurfs* um die Generierung des Quelltexts des Simulationsmodells aus einem gültigen AST-Modell wird ein Template-basierter Ansatz angewandt. Darin werden Muster von Elementen und Strukturen im AST-Modell dazu benutzt, Quelltext in der Zielsprache zu generieren. Das Auftreten dieser Muster wird durch den Verarbeitungsprozess Template-basierter Werkzeuge erkannt und abhängig von damit verknüpften Regeln agiert. Das Modell-zu-Text-Transformationswerkzeug *Xpand* lässt sich in die *Eclipse*-Entwicklungsumgebung integrieren, sodass die gesamte Werkzeugkette auf diese Entwicklungsumgebung und dem darin enthaltenen Rahmenwerk zur Modellierung aufgesetzt werden kann. Xpand liest in einem ersten Schritt das zu transformierende AST-Modell ein und wendet die definierten Transformationsregeln auf diejenigen Elemente an, die für das vorgegebene Muster gültig sind. Schlüsselwörter bzw. Schlüsselkonstrukte der Xpand-Sprache werden durch Anführungszeichen, wie sie in romanischen Sprachen verwendet werden, *Guillemets* genannt, gekennzeichnet. Xpand verfügt über Kontrollstrukturen wie Schleifen, bedingte Anweisungen und Verzweigungen, deren Semantik vergleichbar mit denen höherer Programmiersprachen wie C/C++, Java, u.a. ist. Grundsätzlich wird

Tabelle 5.2: Transformationsregeln der Elemente des Komponentenmodells

Komponentenmodell	Link-Modell	AST-Modell	CompositeTypeSpecifier
SystemC-Projekt	-	AST-Projekt	Include-Anweisungen, Instanziierungen, sc_main()
SystemC-Unit	-	Translation Unit	Name, Include-Anweisungen, Präprozessor-Anweisungen, Instanziierungen
SystemC-Modul	-	Simple Declaration	Name, Vererbungen, Instanziierungen, lokale Variablen/Methoden, Interface-Methoden
SystemC-Wrapper	-	Simple Declaration	Vererbungen, Instanziierung Softwarekomponente, Konstruktor, Destruktor, Interface-Methoden, Ports
SystemC-Modul	SysML::Block	Simple Declaration	Name, Vererbungen, Instanziierung SystemC-Wrapper, Konstruktor, Destruktor, TLM-Ports, TLM-Methoden
SystemC-Interface	-	Simple Declaration	Name, Vererbungen
-	UML-Klasse	Simple Declaration	Name, Vererbungen, Instanziierungen, lokale Variablen/Methoden
SystemC-Thread	-	Simple Declaration	Name, Funktionsrumpf
SystemC-Port	-	Simple Declaration	-
SystemC-Export	-	Simple Declaration	-
-	UML-Operation	Simple Declaration	Name, Rückgabewert, Parameter, Funktionsrumpf

in Xpand ein bestimmtes Template bzw. Vorgehen für ein bestimmtes Element des AST-Modells definiert, indem das Schlüsselwort *DEFINE* mit einem Namen und einem Element versehen wird, auf das das Template angewandt werden kann. Innerhalb eines Templates kann sowohl statischer Quelltext generiert, als auch auf Eigenschaften der AST-Elemente und dessen Unterelemente zugegriffen werden. Weiterhin können Templates hierarchisch angeordnet sein. Ein Template „expandiert“ also, indem es auf andere Templates mittels des Aufrufs *EXPAND* verweist, die dann zuerst abgearbeitet werden. Durch diesen Mechanismus wird der Quelltext rekursiv aufgebaut. Die Quelltexte A.6 und A.7, die sich im Anhang befinden, zeigen beispielhaft die Generierung von Quelltext-Artefakten basierend auf Elementen des AST-Modells.

Da beim Entwurf digitaler Hardware/Software-Systeme oftmals auch schon *bestehende Quelltext-Artefakte* eingebunden werden müssen, sollten diese Artefakte nahtlos in den Entwurfsablauf integriert werden können. Weiterhin wird durch das Einbinden von Quelltext-Artefakten ein *Round-Trip-Engineering* ermöglicht, indem durch einen automatisierten Prozess die ständige Konsistenz zwischen Quelltext und dessen Repräsentation im Modell garantiert wird. Die programmiertechnische Aufbereitung von bereits existierendem Quelltext zur Überführung in eine strukturierte Form ist in die zwei Hauptabschnitte *Lexen* und *Parsen* getrennt. Beim Lexen wird der eingelesene Quelltext so in Symbole aufgeteilt, dass die einzelnen *Token* verschiedenen Kategorien zugeteilt werden können, z.B. Bezeichner, Schlüsselwörter, Operatoren. Durch das Parsen werden die atomaren Token auf die Grammatik der Sprache abgebildet, sodass der Quelltext in Elementen und Konstrukten der Sprache interpretiert werden kann. Das Resultat einer vollständigen Interpretation des Quelltexts ist die Erstellung eines *konkreten Syntaxbaums*, der durch Abstraktion auf die entscheidenden Elemente in einen *abstrakten Syntaxbaum* und somit in ein definiertes AST-Modell überführt werden kann. Da durch diesen Vorgang die Struktur des Quelltexts dargestellt und auf eine höhere Abstraktionsebene überführt wird, ist dieser Vorgang ein wichtiger Teil beim *Bottom-Up-Entwurf*, wie er in Abbildung 5.9 abgebildet ist.

Die Umkehrung der Transformation von Abschnitt 5.3.2 und die damit verbundene weitere Abstraktion des strukturorientierten AST-Modells in das Komponentenmodell komplettiert den Bottom-Up-Entwurf und macht den evaluierten Quelltext im Komponentenmodell verfügbar. Zur Definition der Komponentengrenzen müssen in diesem Schritt Annahmen getroffen werden. Bei der Verarbeitung eines bestehenden SystemC-Designs ist es sinnvoll, SystemC-Module in Komponenten des Komponentenmodells zu überführen. Wird reiner C/C++-Quelltext verwendet, kann die Zuordnung zu Komponenten auf der Granularität von Klassen erfolgen. Komplexere Ansätze analysieren die Interaktion zwischen verschiedenen Quelltext-Artefakten und definieren über eine Heuristik die Komponentengrenzen anhand der Intensität der gegenseitigen Interaktion [22]. Existiert also eine enge Interaktion bzw. über viele unterschiedliche Interaktionspunkte, sollten die Artefakte nicht in verschiedenen Komponenten implementiert sein.

5.4 Optimierung der Energieeffizienz in der Simulation

In diesem Abschnitt soll gezeigt werden, wie basierend auf einem Simulationsmodell mit annotierten nicht-funktionalen Eigenschaften für Zeitverbrauch bzw. Leistungsaufnahme und einer verwaltenden Schicht eine Optimierung der Energieeffizienz durchgeführt werden kann, indem das System auf sich ändernde Eigenschaften und Zustände der Auslastung (engl.: *Workload*) reagiert.

5.4.1 Ausführung von Simulationsmodellen mit NFP-Annotationen

Im Grundlagenabschnitt 2.3.2 wird ein existierender Ansatz zur Instrumentierung von Simulationsmodellen in SystemC vorgestellt. Dafür wird zusätzlich zum ursprünglichen Simulationsmodell ein Pfadsimulationsmodell generiert, die den Kontrollfluss des auszuführenden Programms auf der Zielarchitektur abbildet. Die Kombination von instrumentiertem Simulations-Code und Pfadsimulation ist jedoch auf die Ausführung einzelner Programme beschränkt, berücksichtigt also keine Ausführungsplattform von Hardware/Software-Systemen, die die gleichzeitige Benutzung geteilter Ressourcen verwaltet, sowie die Ausführung mehrerer Programme bzw. Applikationen und ein entsprechendes Energiemanagement erlaubt. Hierzu sind abstrakte Erweiterungen zum mehrstufigen Schichtenmodell aus Abschnitt 5.2 notwendig, die die Mechanismen einer *Systemverwaltungsschicht* in einer virtuellen Ausführungsplattform (engl.: *Virtual Execution Platform (VEP)*) modellieren und im Simulationsmodell ausführbar machen [142].

Wie in Abbildung 5.15 dargestellt, müssen auf der Modulschicht der Ausführungsplattform die Aufrufe der durch den Instrumentierungsprozess eingefügten NFP-Annotationen gekapselt werden, sodass die Annotationen für die benötigte Ausführungszeit nicht direkt vom SystemC-Simulationskern ausgeführt werden. Damit wird durch die VEP eine virtuelle Zeitebene parallel zur globalen Simulationszeit t_S eingeführt, die für jeden Simulations-Thread i dessen lokale Zeit t_i betrachtet. Die Trennung von realer und virtueller Simulationsebene ist in Abbildung 5.16 schematisch anhand der Verdrängung eines Threads t_1 durch einen Thread t_2 dargestellt. Da t_2 vor dem eigentlichen Ende von t_1 aktiv wird, darf die globale Zeit t_S nur die durch t_1^{sync} markierte Zeitspanne voranschreiten und auch nur diese wird für Thread t_1 berücksichtigt.

Die annotierten nicht-funktionalen Eigenschaften eines jeden Basisblocks werden durch das Aufrufen der Methode `consume()` während der Simulation in einer speziellen *Delay*-Datenstruktur gespeichert. Eine Synchronisation zwischen realer und virtueller Zeitebene erfolgt, wenn eine der folgenden Eigenschaften gilt:

- Die lokale Zeit t_i des betrachteten Threads i ist größer als eine global spezifizierte Zeitdauer d für die Abweichung zwischen virtueller und realer Simulationsebene
- Der Thread wird durch den Aufruf der SystemC-Methode `wait()` unterbrochen, z.B. durch das aktive Warten auf ein externes Ereignis e_j
- Der Thread kommuniziert nach außen und muss zur Synchronisation mit anderen Threads durch den Methodenaufruf `sleep()` unterbrochen werden

Die Datenstruktur L_i, die jedem Thread i zur Verfügung steht, stellt eine verkettete Liste dar, in der in jedem Element sowohl der zugehörige Zeitverbrauch als auch die Leistungsaufnahme der verwendeten Zielarchitektur pro Instruktion abgelegt wird, dargestellt in Abbildung 5.17. Zusätzlich wird in jedem Element die Summe des Zeitverbrauchs der vorigen Elemente und des aktuellen Elements abgelegt. Um bei der späteren Entnahme das Auffinden eines bestimmten Zeitwerts zu beschleunigen, wird die Liste in Bereiche (engl.: *Chunks*) einer bestimmten Länge n unterteilt. Somit

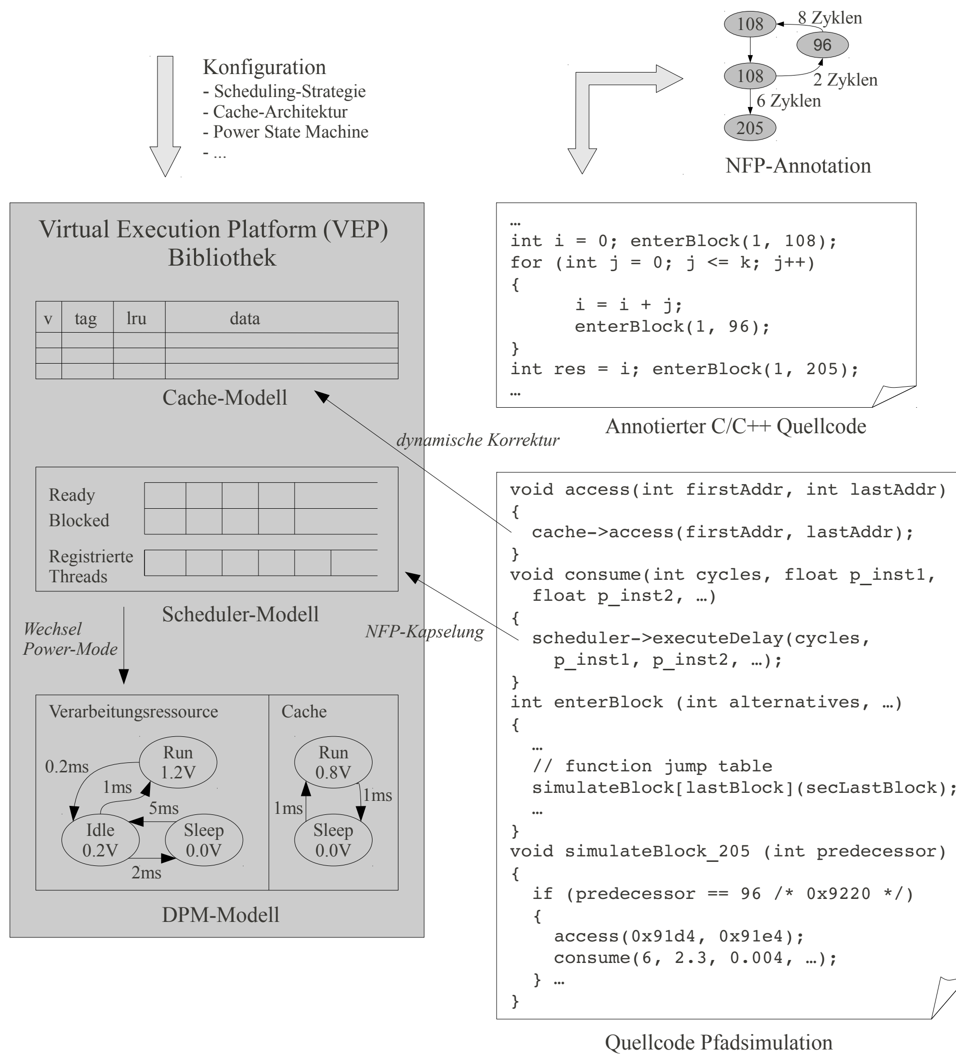


Abbildung 5.15: Ausführung von NFP-annotiertem Simulationsmodell auf der virtuellen Ausführungsplattform (VEP)

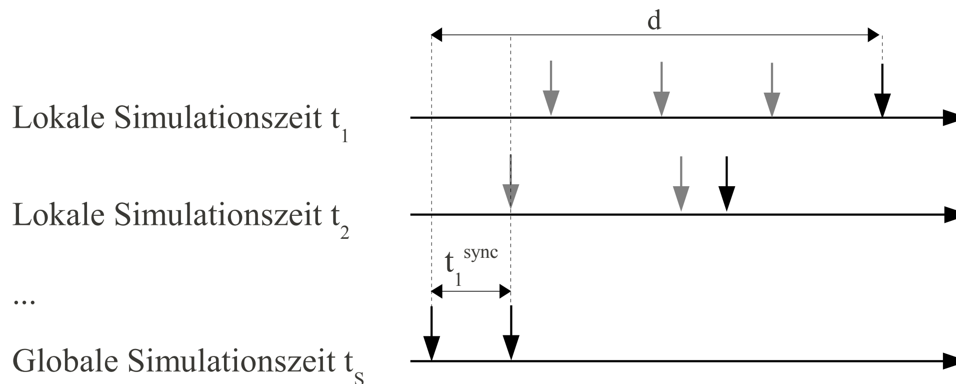


Abbildung 5.16: Trennung zwischen realer und virtueller Simulationsebene

kann zuerst der relevante *Chunk* ausgewählt und darin eine binäre Suche durchgeführt werden. Die Länge der Chunks richtet sich nach der Anzahl der Binärblöcke und soll

ein lineares Durchsuchen aller Elemente nach einem bestimmten Zeitwert verhindern. Dadurch reduziert sich der Suchaufwand von $O(n)$ auf $O(\log n)$.

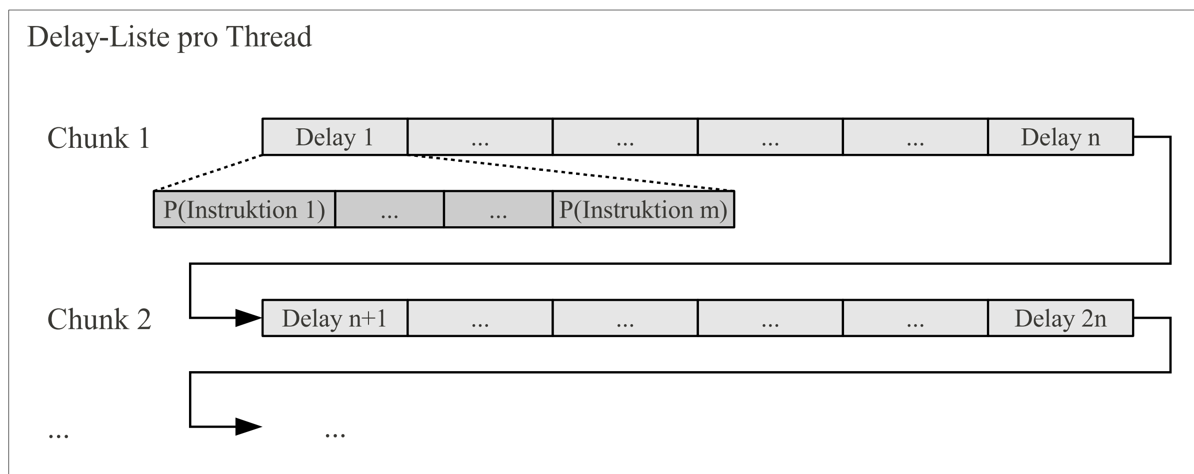


Abbildung 5.17: VEP-Datenstruktur für NFP-Annotationen

In der VEP-Bibliothek sind *Scheduler*-Komponenten mit verschiedenen Ablaufstrategien, z.B. TDMA oder Fixed Priority, implementiert, die den geteilten Zugriff auf geteilte Verarbeitungsressourcen verwalten. Existieren mehrere ausführende Einheiten innerhalb einer geteilten Ressource, z.B. bei einer Mehrprozessor-Architektur, wird jeder Einheit ein *Dispatcher* zugeteilt, der die Aktivierung der einzelnen Threads durchführt. Vor Beginn der Simulation registriert sich jeder Thread beim zuständigen Scheduler zur Verwaltung der möglicherweise auftretenden Zugriffskonflikte mit anderen Threads. Wenn während der Ausführung des instrumentierten Simulations-Quellcodes und des Pfadsimulations-Codes ein Synchronisierungsereignis eintritt, z.B. durch einen initiierten Wechsel des aktiven Threads oder den Aufruf einer Kommunikationsmethode, überprüft der Scheduler, ob der Thread, der gerade als aktiv markiert ist, die gesamte Zeit bis zum Eintreten des Synchronisierungsereignisses t_i^{sync} aktiv sein durfte, d.h. die globale Simulationszeit t_S um die Summe aller Zeitwerte seiner Liste L_i erhöht und diese Werte aus der Liste gelöscht werden dürfen. Ist $t_i^{sync} < t_i$, z.B. aufgrund eines durch das Betriebssystem des Hardware/Software-Systems initiierten Wechsels des auszuführenden Threads, wird jedoch nur die Differenz $t_i - t_i^{sync}$ aus L_i entnommen und die globale Simulationszeit t_S um diesen Wert erhöht. Das Fortschreiten der globalen Simulationszeit t_S wird dabei durch den Aufruf der Methode `wait()` im SystemC-Simulationskern und der Übergabe des Differenzwerts realisiert. Danach kann der Thread-Wechsel erfolgen, da die lokale Zeit und die Simulationszeit des Thread-Wechsels synchron sind.

Der `wait()`-Methode können neben dem entsprechenden Zeitwert noch endlich viele Ereignisse als SystemC-Events (siehe Abschnitt 2.3.1.3 des Grundlagenkapitels) übergeben werden, in dem sie disjunktiv mit dem Zeitwert verknüpft werden. Da ein Zeitwert für den ereignisbasierte Simulationskern ebenfalls ein Ereignis darstellt, bewirkt der Aufruf, dass die Simulationszeit t_S lediglich bis zum ersten Eintreffen eines dieser Ereignisse erhöht wird. Mithilfe der Disjunktion des Differenzwerts mit möglichen

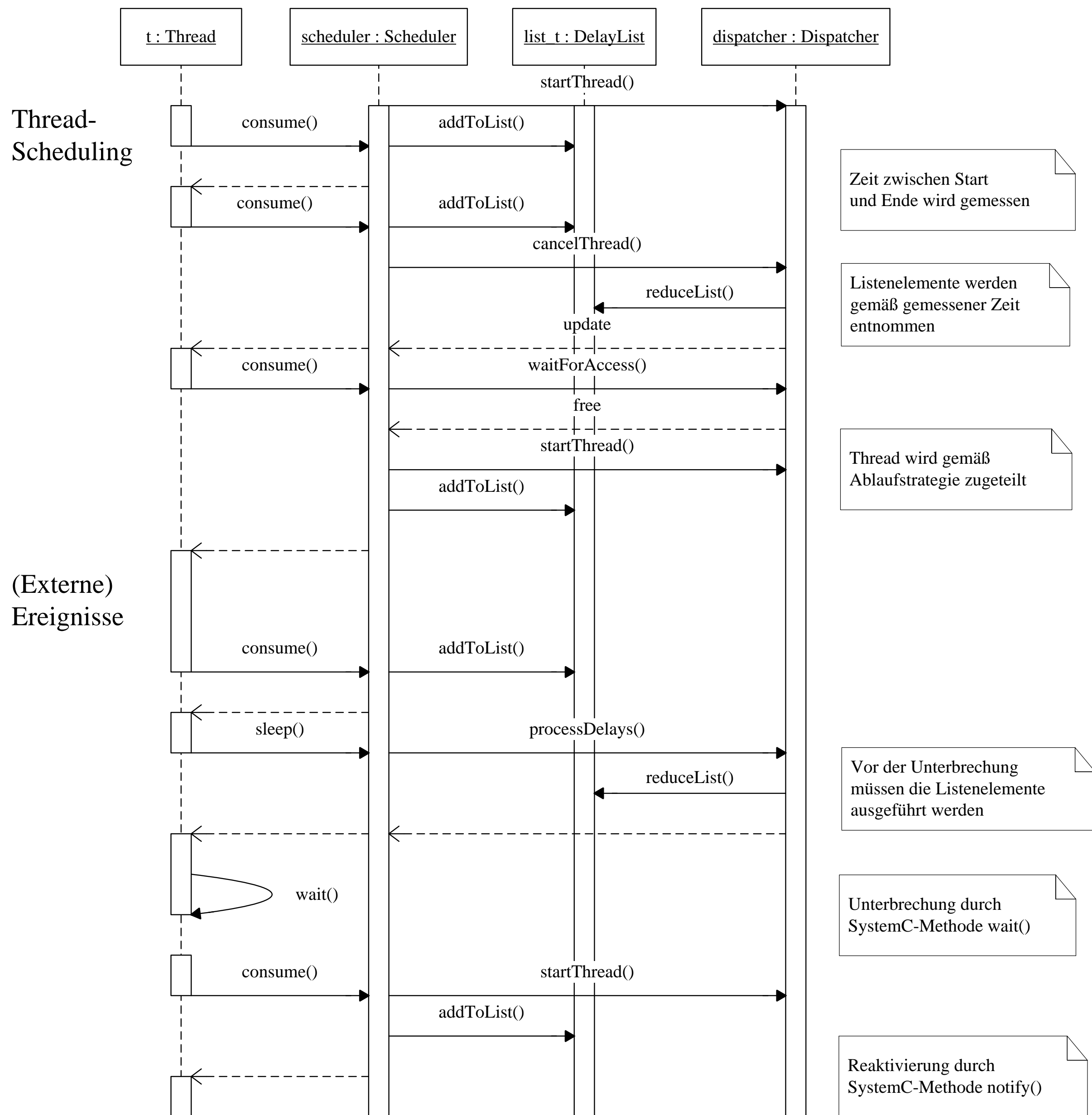


Abbildung 5.18: Thread-Scheduling und Einbeziehung von (externen) Ereignissen

Ereignissen können alle Arten von externen Ereignissen, wie z.B. potentiell auftretende Hardware-Interrupts, in die VEP integriert werden. Da die globale Synchronisation des gerade noch aktiven Threads erst durch das Aufrufen der `wait()`-Methode im SystemC-Simulationskern durchgeführt wird, müssen die Zeitstempel vor und nach dem Aufruf verglichen werden. Durch Bildung der Differenz wird berechnet, wie groß der Wert t_i^{sync} auch unter Einbeziehung externer Ereignisse letztendlich war, und die Liste L_i dementsprechend um diesen Wert verkürzt. Die verbleibenden Zeit- bzw. *Delay*-Elemente werden erst bei einer erneuten Aktivierung des Threads ausgeführt. Dies ist im oberen Abschnitt der in Abbildung 5.18 dargestellten Abfolge der in der VEP aufgerufenen Methoden ersichtlich. Im unteren Abschnitt wird der Ablauf beschrieben, wenn der Thread auf externe Ereignisse warten bzw. mit anderen Threads

kommunizieren soll. Dies ist dann der Fall, wenn im ursprünglichen Simulations-Quellcode bereits eine SystemC-Methode `wait()` enthalten ist, die unter Umständen auch in einem komplexen Kommunikationsmechanismus gekapselt sein kann. Vor der `wait()`-Methode wird durch das Ausführen der Methode `sleep()` mitgeteilt, dass der Thread danach blockiert sein wird, bis das entsprechende Ereignis zur Reaktivierung des Threads ausgelöst wird. Die `sleep()`-Methode wird immer dann aufgerufen, wenn ein Synchronisationspunkt erreicht wird. Vor der Unterbrechung des Threads werden daraufhin zuerst alle noch in der Liste befindlichen Elemente ausgeführt, sodass die lokale Zeit des Threads synchron zur globalen Simulationszeit ist. Nachdem die Synchronisation hergestellt ist, wird der Thread durch die nativen SystemC-Methoden `wait()` und `notify()` unterbrochen bzw. wieder reaktiviert, sodass die VEP in die Thread-Unterbrechung an sich nicht involviert ist.

Abbildung 5.19 stellt die Methode `consume()`, die in den Komponenten bzw. den darin enthaltenen Threads aufgrund der NFP-Annotationen ausgeführt wird, als Flussdiagramm dar. Dieser Methode werden der Zeitverbrauch und die Leistungsaufnahme des jeweiligen Basisblocks als Parameter *Dauer* und *Leistung* übergeben. In ihrem Verlauf kann weiterhin überprüft werden, ob durch den Parameter `sync` eine Synchronisierung explizit angefordert wird, z.B. wenn der Thread unmittelbar vor einer Kommunikation mit anderen Threads durch den Aufruf der Methode `sleep()` solange warten soll, bis die Kommunikation abgeschlossen wurde. Zuerst wird jedoch überprüft, ob der Thread aktiv und aufgrund geteilter Ressourcen, z.B. Ausführungseinheiten, überhaupt ausgeführt werden darf, ansonsten wird darauf gewartet. Danach wird unterschieden, ob eine Synchronisation explizit angefordert wird oder nicht. Ist dies nicht der Fall und bleibt die Abweichung der lokalen Zeit t_i zur globalen Simulationszeit t_S trotz Hinzunahme des übergebenen Zeitverbrauchs kleiner als die maximal erlaubte Abweichung d, werden sowohl der übergebene Zeitwert als auch die dazugehörige Leistungsaufnahme in der Delay-Liste L_i abgespeichert und die Ausführung des instrumentierten Simulations-Quellcodes fortgesetzt. Wird die Abweichung durch den übergebenen Zeitwert zu groß, wird durch Ausführung der Methode `processDelayList()` zunächst die Liste reduziert, was gleichbedeutend ist mit der Synchronisation zwischen lokaler und globaler Simulationszeit, wobei dies unter Berücksichtigung aller Threads und etwaiger Thread-Wechsel durchgeführt wird. Anschließend wird die `consume()`-Methode erneut aufgerufen, um die aktuellen NFP-Annotationen verwalten zu können. Die Delay-Liste wird auch dann reduziert, wenn eine explizite Synchronisation angefordert wird und $t_i > t_S$ gilt. Sind lokale und globale Simulationszeit bereits synchron, werden die NFP-Annotationen sofort bearbeitet, indem die Methode `executeDelay()` ausgeführt wird. Wurde der Thread während der Ausführung zum Zeitpunkt t_i^{sync} unterbrochen, wird die globale Simulationszeit nur bis zu diesem Zeitpunkt erhöht und die `consume()`-Methode mit der Differenz zwischen der ursprünglich geplanten Dauer und der realen Dauer erneut ausgeführt. In diesem Fall wird die Leistungsaufnahme linear interpoliert übergeben, da die Leistungsaufnahme innerhalb eines Basisblocks aufgrund des zugrunde liegenden instruktionsbasierten Leistungsmodells (siehe dazu auch Abschnitt 2.7.2) als gleichverteilt angenommen werden muss.

Die eben erwähnten Methoden `processDelayList()` und `executeDelay()` sind

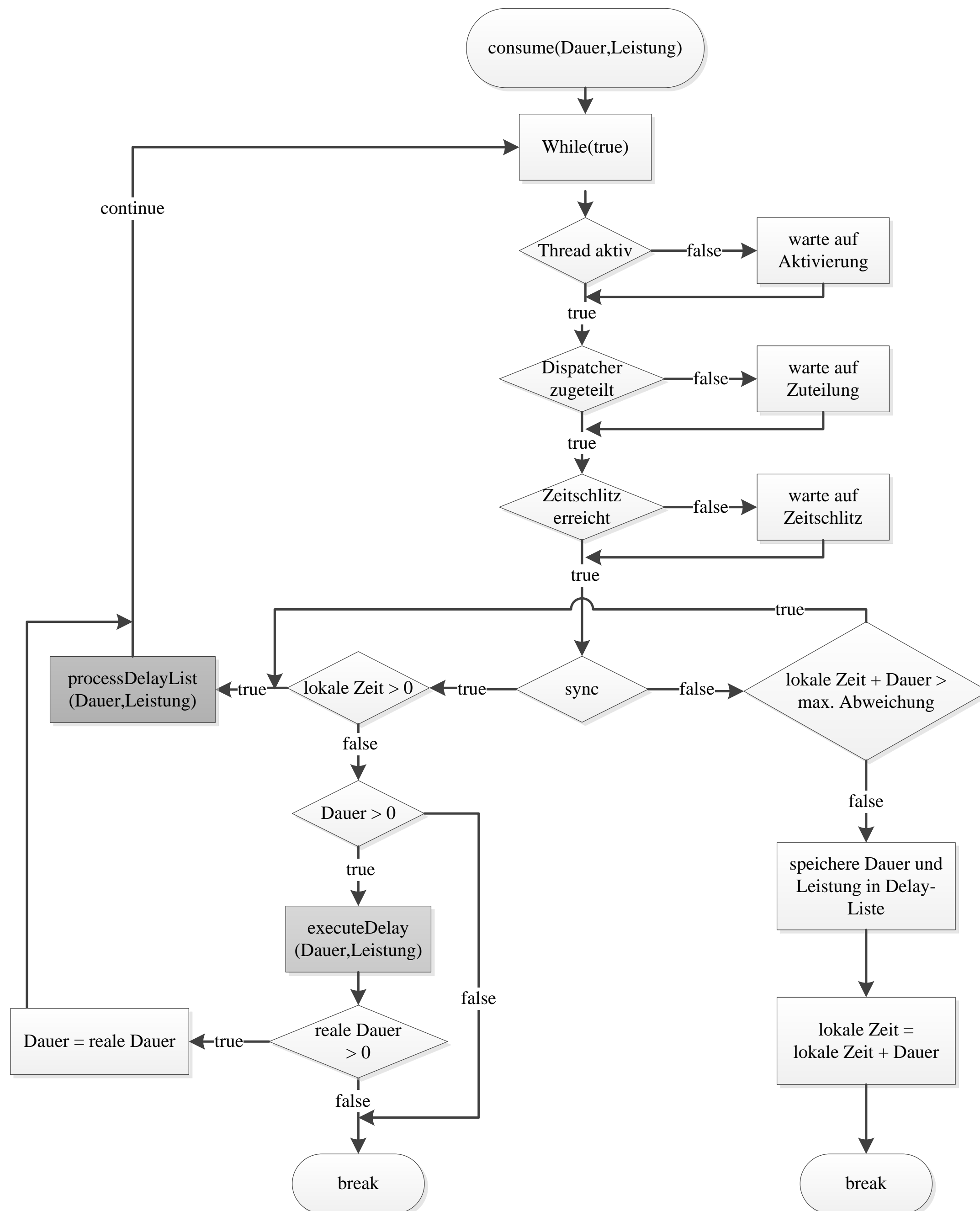


Abbildung 5.19: Flussdiagramm für `consume()`-Methode zur Verarbeitung der NFP-Annotationen im Simulationsmodell [73]

als Flussdiagramme in Abbildung 5.20 dargestellt und sollen hier nur kurz erläutert werden, da ihre Funktionalität bereits in vorigen Erklärungen implizit enthalten war. Ein Hauptbestandteil dieser Methoden ist der darin enthaltene Aufruf der `centralWait()`-Methode, die den akkumulierten Aufruf der Methode `wait()` des SystemC-Simulationskerns über die in der Liste enthaltenen Delay-Elemente kapselt. Hierbei wird die Differenz der Zeitstempel vor und nach dem Aufruf mit dem ursprünglich geplanten Zeitwert verglichen. Anhand dieses Vergleichs kann überprüft werden, ob der aktuelle Thread bei seiner Ausführung unterbrochen wurde, sei es

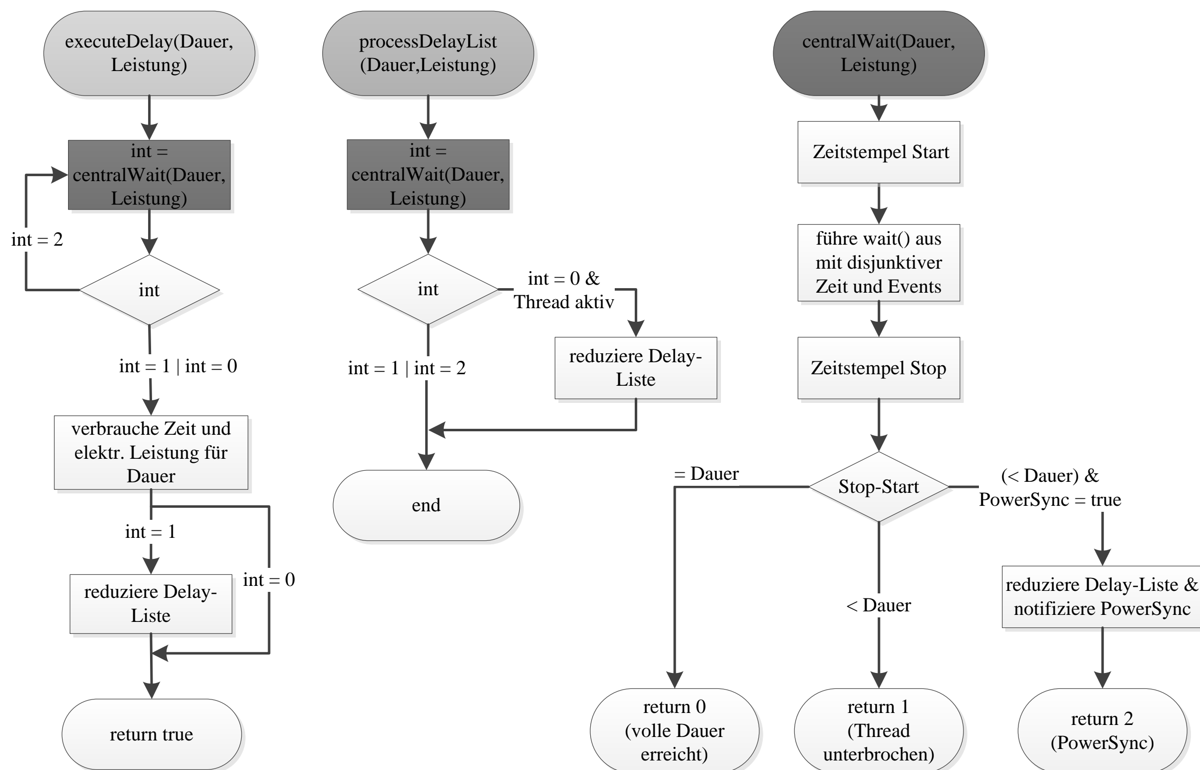


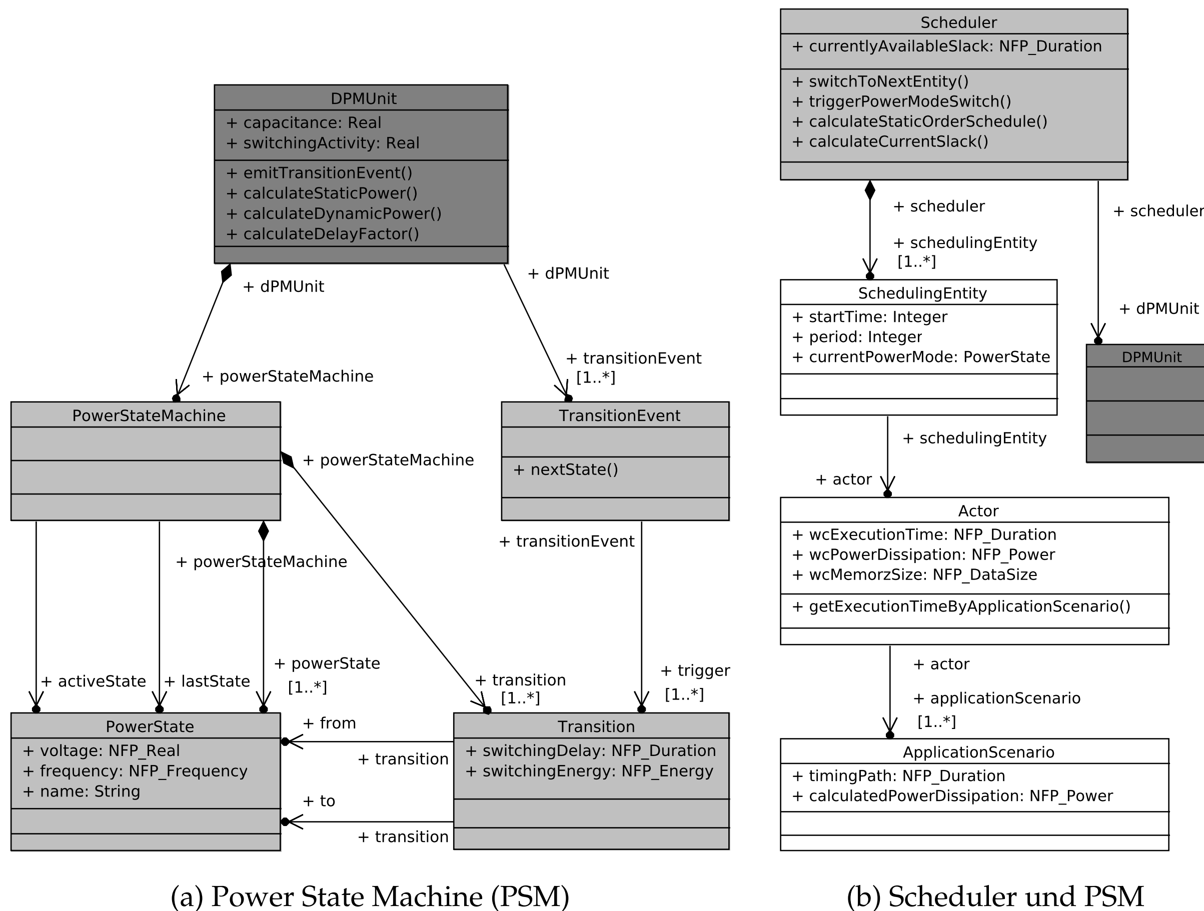
Abbildung 5.20: Flussdiagramme für weitere zentrale VEP-Methoden [73]

aufgrund einer Verdrängung durch den Scheduler oder durch das Eintreten bestimmter Ereignisse. Als weitere Möglichkeit ist hier mit der Ausgabe 2 dargestellt, dass ein periodisches Ereignis eintritt, um die bis zu diesem Zeitpunkt in der Liste enthaltenen NFP-Elemente nach jeweils einer bestimmten Zeit auszuführen. In diesem Fall handelt es sich um ein periodisches Synchronisationsereignis, um die Leistungsaufnahme in einem externen Werkzeug [135] zur Berechnung der Temperaturentwicklung auf der zugrunde liegenden Hardware-Architektur verwenden zu können. Die Stimulierung einer virtuellen Hardware-Plattform durch die umgebende Systeme und die Auswirkung der Leistungsaufnahme auf die Temperaturentwicklung wird in Abschnitt 7.2.2 des Anwendungs- und Ergebniskapitels näher betrachtet.

5.4.2 Reaktives Energiemanagement in der Simulation

Das Wissen der VEP-Verwaltungsschicht über die abzuarbeitenden Threads und den entsprechenden zeitlichen Ablaufplan wird zur Durchführung eines Energiemanagement während der Simulation verwendet. Da die Simulation jedoch nicht auf einem analytischen Modell basiert, können nur eingeschränkt Vorhersagen über zukünftige Systemzustände getroffen und deshalb lediglich der momentane Zustand in der Simulation für das Energiemanagement herangezogen werden. Da das Energiemanagement auf das aktuelle Systemverhalten reagiert, kann diese Form des Management auch als *reaktives Energiemanagement* bezeichnet werden.

Die dem Modell des Energiemanagements auf der Ausführungsplattform zugrunde



(a) Power State Machine (PSM) (b) Scheduler und PSM

Abbildung 5.21: Metamodellierung für das Energiemanagement auf der virtuellen Ausführungsplattform (VEP)

liegenden Metamodelle sind in Abbildung 5.21 dargestellt. Abbildung 5.21a zeigt die Metamodellierung einer *Power State Machine (PSM)*, Abbildung 5.21b deren Interaktion mit dem verwaltenden Scheduler. Eine PSM besteht demzufolge aus mehreren *Power States*, die über *Transitionen* und den zugehörigen Rollen *from* für den Ausgangszustand und *to* für den Zielzustand miteinander verbunden sind. Einer *DPMUnit*, also einer Einheit, die über ein dynamisches Powermanagement verfügt, ist eine PSM zugeordnet, wobei sie durch Auslösen eines Ereignisses *TransitionEvent* eine Transition eines PSM-Zustands in einen anderen veranlassen kann. Weiterhin kann die DPMUnit die statische und dynamische Leistungsaufnahme anhand der im aktiven Zustand verwendeten Frequenz und Versorgungsspannung berechnen. Eine DPMUnit ist wiederum einem Scheduler zugeteilt, der für die Verwaltung mehrerer *SchedulingEntities* verantwortlich ist. Die bei der Ablaufplanung zu berücksichtigenden Elemente sind Objekten des Typs *Actor* zugeordnet, die z.B. über eine Worst-Case Execution Time und eine durchschnittliche Leistungsaufnahme relativ zu einem festgelegten Referenz-Betriebsmodus bzw. Power-Modus verfügen. Die maximale Ausführungszeit kann abhängig von dem jeweils ausgeführten Applikationsszenario (engl.: *ApplicationScenario*) und dem damit ausgeführten Ausführungspfad innerhalb der Anwendung sein.

In der virtuellen Ausführungsplattform wird die Aufgabenverteilung zwischen der verwaltenden Rolle des Scheduler und der ausführenden Rolle der Dispatcher so geregelt, dass der Scheduler informiert wird, welche Dispatcher ihre momentane Ausführung beendet haben. Ist zu diesem Zeitpunkt kein Thread zur Abarbeitung bereit, also z.B. noch blockiert, oder bei einem TDMA-Verfahren liegt der nächste Zeitschlitz zur Zuteilung soweit in der Zukunft, dass ein Wechsel des Power-Modus trotz dessen Zusatzaufwands vorteilhaft für den Energieverbrauch des Hardware/Software-Systems ist, initiiert der Scheduler den Wechsel in einen energieeffizienteren Power-Modus. Benötigt der Dispatcher nach einer gewissen Zeit die volle Leistungsfähigkeit bzw. Ausführungsgeschwindigkeit, resultiert dies in einem erneut durchgeführten Modus-Wechsel, der die Performanzanforderungen erfüllt. Beim Voranschreiten der globalen Simulationszeit bzw. bei der Entnahme der Delay-Elemente aus den entsprechenden Listen wird der momentan gewählte Power-Modus berücksichtigt. Da die im Simulationsmodell annotierten NFP für Zeitverbrauch und Leistungsaufnahme auf einem bestimmten normierten Power-Modus basieren, müssen die tatsächlich aktuell anzuwendenden Werte durch Skalierung berechnet werden. Diese Skalierung wurde bereits in Abschnitt 4.1.5 vorgestellt. Die Berechnung der momentan geltenden Werte für Leistungsaufnahme und Zeitverbrauch kann also relativ zum während des Prozesses der NFP-Instrumentierung verwendeten Power-Modus und vor jeder Synchronisation erfolgen.

Abbildung 5.22 soll das grundsätzliche Vorgehen des reaktiven Energiemanagements anhand zweier Software-Komponenten verdeutlichen, die durch einen gemeinsamen Dispatcher ausgeführt werden. Die Komponenten werden durch den Scheduler einer einzelnen Verarbeitungsressource gemäß einer Fixed-Priority-Strategie zugeteilt, wobei *Komponente B* eine höhere Priorität besitzt als *Komponente A*. Die zur Verfügung stehende Verarbeitungsressource kann in zwei unterschiedlichen Power-Modi ausgeführt werden – hier vereinfacht mit *Nominaler Modus* und *Low-Power-Modus* bezeichnet. Der Low-Power-Modus wird genau dann aktiviert, wenn keine der beiden Komponenten ausgeführt wird. Die Zeiten, in denen die Komponenten ausgeführt werden können, sind mit einem umrandeten Rechteck gekennzeichnet. Wird eine Komponente tatsächlich ausgeführt, ist dies mit einem grauen Verlauf innerhalb des Rechtecks markiert. Die Zeiten, in denen die Komponente blockiert ist, z.B. durch das Warten auf ein bestimmtes Ereignis, sind durch einen schwarzen Verlauf gekennzeichnet. Eine ähnliche Markierung gilt für den oberen Bereich in Abbildung 5.22, der die zugrunde liegende Hardware-Plattform und insbesondere die Wahl des Power-Modus darstellt. Ein schwarzer Balken bedeutet, dass die ausführende Ressource wegen eines Wechsels des Power-Modus zeitweise blockiert ist, der graue Verlauf markiert die Ausführung im jeweiligen Power-Modus.

Zu Beginn der Simulation befindet sich die Verarbeitungsressource im Low-Power-Modus, was einen Wechsel in den nominalen Modus vor der Ausführung einer Komponente bedingt. Nach Vollendung des Wechsels können alle ablaufbereiten Komponenten bzw. Threads abgearbeitet werden. Da Komponente *B* eine höhere Priorität als *A* besitzt, wird zuerst *B* ausgeführt. Dies führt dazu, dass Komponente *A* zwar als ablaufbereit markiert ist, jedoch vorerst nicht ausgeführt werden kann, was

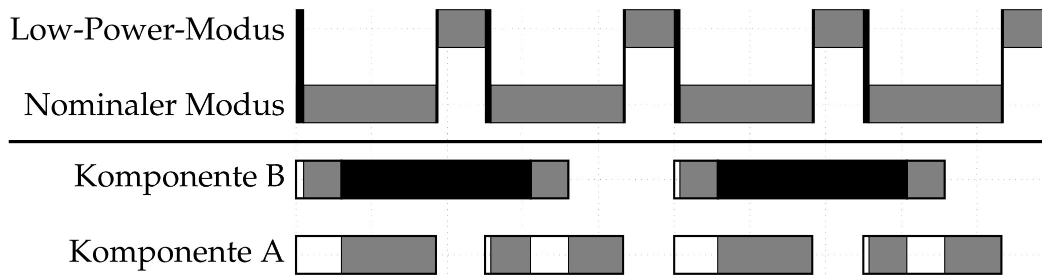


Abbildung 5.22: Zeitlicher Verlauf der Anwendung eines reaktiven Energiemanagements bei 2 Beispielkomponenten (grau: aktiv, schwarz: blockiert, weiß: wartend)

als weiße Fläche innerhalb eines vorhandenen Rechtecks dargestellt ist. Komponente *B* blockiert im weiteren Verlauf, was durch eine schwarze Fläche im Rechteck dargestellt ist. Da *A* immer noch ablaufbereit ist, weißt der Scheduler dem zur Verfügung stehenden Dispatcher die Komponente *A* zur Ausführung zu bis diese komplett abgearbeitet ist. Während dieser Zeit befindet sich die Verarbeitungsressource weiterhin im nominalen Power-Modus. Da Komponente *A* nach deren Ende nicht sofort wieder ausgeführt werden soll, Komponente *B* aber immer noch blockiert ist und somit kein ablaufbereiter Thread für den Dispatcher existiert, veranlasst der Scheduler einen Wechsel in den Low-Power-Modus. Dargestellt ist dies durch einen schwarzen Balken für den Modus-Wechsel und ein nachfolgendes graues Rechteck, das den zeitlichen Verlauf im Low-Power-Modus anzeigt. Wird Komponente *A* beim Eintritt ihrer nächsten Periode wieder aktiviert, wechselt der Betriebsmodus wieder in den nominalen Modus, was zunächst wieder zu einem kurzen Blockieren der Verarbeitungsressource führt. Ändert Komponente *B* während der Ausführung von *A* seinen Zustand von *blockiert* auf *ablaufbereit*, wird Komponente *A* vom Scheduler unterbrochen und dem Dispatcher Komponente *B* zur Ausführung zugeteilt. Erst nach Beendigung von *B* kann *A* seine Ausführung fortsetzen bis auch *A* beendet ist, was zu einem erneuten Wechsel in den Low-Power-Modus führt. Dieser eben erklärte und in Abbildung 5.22 dargestellte zeitliche Verlauf wiederholt sich danach periodisch. Kritische Systemzustände, wie z.B. die Verletzung der Periodengrenzen von Komponenten, lassen sich durch die Analyse des simulativen Verhaltens erkennen und ermöglichen damit sowohl gezielte Modifikationen in der Systemverwaltungsschicht, z.B. beim Energiemanagement oder bei der Ablaufplanung, als auch die frühzeitige Identifizierung von Entwurfsfehlern – sowohl was die Softwarearchitektur, als auch die Hardware-Plattform betrifft.

Unter anderem kann durch die Berücksichtigung der Leistungsaufnahme während der Simulation und einer festgelegten Strategie für das Energiemanagement auch ein *Budget* für die Leistungsaufnahme einzelner Software- bzw. Hardware-Komponenten berechnet und festgelegt werden. Dieses Budget ist offensichtlich abhängig von den Stimulierungsdaten, die für das Hardware/Software-System in der Simulation verwendet werden. Eine detaillierte Betrachtung und Einbeziehung sowohl der Testfälle als auch der Testdaten in die Berechnung der zu definierenden Budgets ist somit unumgänglich, soll aber in dieser Arbeit nicht näher betrachtet werden. Mithilfe einer geeigneten Stimulierung das ausführbaren Modells lassen sich typische Szenarien der jeweiligen Anwendung identifizieren. Die Extraktion der Applikationsszenarien und die Nutzung

dieser für ein gezieltes Energiemanagement stellt den *Brückenschlag* zwischen dem Simulationsmodell und einem analytischen Modell dar und wird im nachfolgenden Abschnitt 5.4.3 thematisiert.

5.4.3 Analyse des dynamischen Verhaltens in Applikationen

Simulationsmodelle, wie sie bisher Gegenstand dieses Kapitels waren, können nicht nur dafür benutzt werden, Systemmodelle bestehend aus Hardware und Software innerhalb einer Simulation ausführbar zu machen. Vielmehr können sie bei zielgerichtetem Einsatz und Interpretation einen essentiellen Beitrag zur Verfeinerung abgeleiteter analytischer Modelle darstellen. Analytische Modelle benutzen in der Regel spezifizierte Charakteristika und Eigenschaften eines Modells zur Ableitung bestimmter Analyseziele. Im Bereich eingebetteter Hardware/Software-Systeme spielen dabei insbesondere Analysen zum Nachweis von Echtzeitanforderungen basierend auf den maximalen Ausführungszeiten (engl.: *Worst-Case Execution Time*) bestimmter System- und Programmteile eine tragende Rolle [130]. Diese Ansätze haben oftmals die Gemeinsamkeit, dass sie ohne Berücksichtigung der jeweils geltenden Eingabedaten und Ausführungskontexte sind, um garantiert alle möglichen zeitlichen Randbedingungen abdecken zu können. Sie abstrahieren damit von unterschiedlichen Ausführungspfaden bzw. *Timing-Pfaden*, die innerhalb einer bzw. mehrerer Applikationen vorkommen können und die besonders in Form von datenabhängigen Kontroll- bzw. Programmstrukturen eine maßgebliche Rolle am Zeitverhalten und dem Energieverbrauch der implementierten Funktionalität auf der Zielarchitektur spielen können. Unterschiedliche Ausführungspfade innerhalb von Applikationen können meist nur in ausführbaren Systemmodellen detailliert berücksichtigt werden, da der dynamische Kontext erst während der Ausführung feststeht.

Anhand geeigneter Testfälle lassen sich NFP-annotierte Simulationsmodelle stimulieren und die daraus resultierenden Ausführungszeiten der ausgeführten Applikationen analysieren. Die Stimulierung kann dabei auch unter Verwendung einer vorgegebenen Testumgebung oder einer Infrastruktur zur automatischen Generierung von Testfällen durchgeführt werden. Verschiedene Ausführungszeiten bilden dadurch die unterschiedlichen Ausführungspfade ab. Die Abdeckung der möglichen Ausführungspfade durch die simulierten Testfälle kann durch Verwendung einer an das Systemmodell angepassten *Coverage-Analyse* gesteuert werden. Durch eine zielgerichtete Steuerung der Generierung von Test-Stimuli, z.B. durch Anpassung entsprechender Parameter nach einer bestimmten Verteilung, soll dabei eine möglichst vollständige Abdeckung der relevanten Ausführungspfade erreicht werden. Hierfür existiert eine Vielzahl von Ansätzen, die meist auf einem permanenten Vergleich zwischen erreichter Abdeckung und entsprechender Parameterbelegung basiert. Eine abgestimmte Coverage-Analyse stehen an dieser Stelle jedoch nicht im Fokus, sodass die Abdeckung der Ausführungspfade hier also als ausreichend angenommen wird.

Durch anschließende Anwendung einer *Cluster-Analyse* bzw. eines *Clustering-Algorithmus* auf die Menge der unterschiedlichen Ausführungspfade in mehreren Simulationen werden schließlich Gruppen (engl.: *Cluster*) gebildet, die ein vergleichba-

res Zeitverhalten aufweisen. Im Allgemeinen werden Cluster-Analysen zur Erkennung von Strukturen eingesetzt, die eine Ähnlichkeit bezüglich einer oder mehrerer Eigenschaften aufweisen. Hierfür wird ein *k-means-Algorithmus* verwendet, der für die Menge an Elementen eine Einteilung in *k* Gruppen abhängig von der Zielfunktion bzw. der jeweiligen Entfernung zum Gruppenmittelpunkt vornimmt. In einem iterativen Verfahren wird dabei die Methode der kleinsten Quadrate angewandt und einzelne Elemente so anderen Gruppen zugeordnet, dass die Zielfunktion minimiert wird [80]. Dieser Algorithmus wird auch als *Lloyd-Algorithmus* oder *Voronoi-Iteration* bezeichnet.

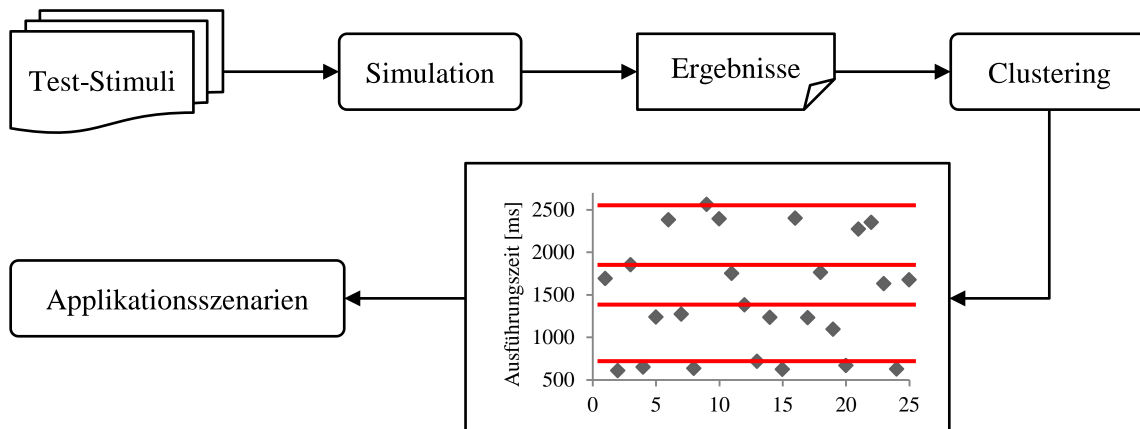


Abbildung 5.23: Ableitung von Applikationsszenarien durch Simulation

Als Entscheidungskriterium, an der die Gruppenzugehörigkeit der Ausführungspfade definiert wird, dient die Ausführungszeit der simulierten Applikation auf der virtuellen Ausführungsplattform, die der nächst-größeren Gruppengrenze entspricht. Dadurch bildet diese eine obere Schranke für die Ausführungszeit aller in der Gruppe befindlichen Ausführungspfade auf der Zielplattform. Da jede Gruppe ein bestimmtes Szenario der Applikationsausführung darstellt, kann dieses auch als *Applikationsszenario* bezeichnet werden. Den Applikationsszenarien wird jeweils eine charakteristische Ausführungszeit zugewiesen. Dieser Vorgang ist in Abbildung 5.23 dargestellt. Die Menge an Gruppen bzw. Applikationsszenarien ist vom zugrunde liegenden Systemmodell abhängig und kann deswegen nicht pauschal vorgegeben werden. Sie kann aber durch die Durchführung mehrerer Iterationen der Cluster-Analyse und unterschiedlicher Gruppenanzahl sinnvoll bestimmt werden.

Das Simulationsmodell wird durch das beschriebene Verfahren zur Analyse des dynamischen Verhaltens in den simulierten Applikationen benutzt. Die als Resultat gewonnenen Applikationsszenarien können anschließend in dafür vorgesehene Analysemodelle integriert werden, um die Ergebnisse einer zur Ausführungszeit durchgeführten Energieoptimierung unter Berücksichtigung der Eingabe-abhängigen Beeinflussung des nicht-funktionalen Verhaltens zu verbessern. Ein solches analytisches Modell wird inklusive der damit verbundenen Optimierung in Abschnitt 6.2 detailliert vorgestellt und in Abschnitt 7.3.2 anhand einer Beispiel-Applikation zur Anwendung gebracht.

Kapitel 6

Optimierung der Energieeffizienz digitaler Hardware/Software-Systeme

Wie in vorigen Kapiteln bereits erläutert, hat sich die Minimierung der Leistungsaufnahme und damit der Energieverbrauch, die für die Leistungserbringung aufgebracht werden muss, zu einem der wichtigsten Ziele beim Entwurf eingebetteter Hardware/Software-Systeme entwickelt. Dies ist vor allem die Folge des Trends zu mobilen Endgeräten wie Smartphones und Tablet-Computern, aber auch einer stark steigenden Anzahl an Steuergeräten im Automobilbereich und der stetig präsenten Notwendigkeit der Energieeinsparung über alle Anwendungsdomänen hinweg [87].

In diesem Kapitel wird beschrieben, wie in unterschiedlichen Phasen des Entwurfs sowie während des eigentlichen Betriebs des Systems automatisch eine optimierte Konfiguration des Energiemanagements abgeleitet werden kann. Dabei müssen grundlegende Performanzanforderungen eingebetteter Hardware/Software-Systeme beim Einsatz in Echtzeitsystemen beachtet werden. Weiterhin müssen Entscheidungen getroffen werden, welche Mechanismen zur Reduzierung der Leistungsaufnahme und deren zugrunde liegenden Powermodelle verwendet werden sollen, indem deren Einsatz während des Systementwurfs bzw. während der Laufzeit anhand der in den unterschiedlichen Phasen zur Verfügung stehenden Systemeigenschaften untersucht wird. Die entwickelten Verfahren zur Optimierung der Leistungsaufnahme werden im Detail vorgestellt. Hierbei wird ein Optimierungsverfahren zur Entwurfszeit des Systems mit einem Verfahren, das während der Laufzeit des Systems durchgeführt wird, kombiniert. Letzteres nutzt zusätzlich zur Charakteristik und zum Worst-Case-Zeitverhalten der Applikationen auch die Dynamik innerhalb der Applikationen zur Minimierung der Leistungsaufnahme. Die durch die Optimierung zu erzielende Steigerung der Energieeffizienz hängt entsprechend sowohl von den Aufgaben selbst, als auch von den an das Hardware/Software-System gestellten Anforderungen bezüglich zu geltender Performanzeigenschaften ab.

6.1 Optimierung zur Entwurfszeit

Eine Optimierung der Energieeffizienz von Hardware/Software-Systemen ist gleichbedeutend mit der Minimierung der Gesamtenergiekosten, die dieses System während der Ausführung spezifizierter Aufgaben verursacht. Da bei diesem Optimierungsschritt eine globale Sicht auf das System notwendig ist und es meist nur geringe Einschränkungen bezüglich der Realisierung dieser Aufgaben gibt, sollte er auf einer möglichst hohen Abstraktionsebene durchgeführt werden. Konkreter ausgedrückt soll also die Energie minimiert werden, die notwendig ist, um eine spezifizierte Menge an Applikationen auf einer oder mehreren Verarbeitungsressourcen auszuführen. Dabei können die Verarbeitungsressourcen in verschiedenen Betriebsmodi ausgeführt werden, die jeweils einen gewissen Einfluss auf den Energieverbrauch, aber auch auf die Ausführungsgeschwindigkeit besitzen. Das hauptsächliche Ziel ist eine optimierte Zuordnung von Applikationen auf Verarbeitungsressourcen und Betriebsmodi, in denen die jeweilige Applikation auf der Ressource ausgeführt wird. Da diese Zuordnung entweder als positive oder negative Entscheidung zu jeder Kombination von Applikation, Verarbeitungsressource und Betriebsmodus interpretiert werden kann, kann sie jeweils durch eine binäre Variable angezeigt werden. Aus dieser Betrachtung resultiert ein Zuordnungsproblem, das sich als ganzzahliges Optimierungsproblem formulieren lässt.

Die hier beschriebene Optimierung auf Applikationsebene basiert auf der Worst-Case Execution Time der an der Ausführung beteiligten Applikationen (zur Definition einer Applikation siehe Abschnitt 4.2). Da bei einer Optimierung auf Ebene einzelner Applikationen in der Regel wenige Datenabhängigkeiten zwischen diesen existieren, gibt es viele Freiheitsgrade bezüglich der Ausführungsreihenfolge und der Abbildung der Anwendungen auf Verarbeitungsressourcen.

6.1.1 Formulierung als mathematisches Optimierungsproblem

In diesem Abschnitt soll ein entsprechendes Optimierungsproblem formuliert und durch einen entsprechenden Lösungsalgorithmus gelöst werden. Die Vorteile dieses Vorgehens liegen zum einen an der Tatsache, dass das vorliegende Problem durch eine hohe Komplexität geprägt ist, sodass es durch ein konstruktives Optimierungsverfahren nur durch die Anwendung von abstrahierenden Heuristiken innerhalb einer akzeptablen Lösungszeit gelöst werden kann. Das Teilproblem der Zuordnung von Applikationen zu Ressourcen gehört zur Klasse der ganzzahligen Optimierungsprobleme, welche wiederum in die Komplexitätsklasse der NP-schweren Probleme fällt. Die zusätzliche Kombination aus DVFS und DPM stellt selbst für kleine Problemgrößen ein komplexes Problem dar. Zum anderen liegt der Vorteil in der Verfügbarkeit guter Optimierungssoftware, die für diese Anwendungsfälle entwickelt wird. Mathematische Werkzeuge wie *MathWorks® MATLAB®* oder *GNU Octave* bieten hierfür vielversprechende Ansätze.

Im Gegensatz zu einem konstruktiven Optimierungsverfahren verlagert die Anwendung einer mathematischen Optimierung die Problemkomplexität in die Formulierung einer *Zielfunktion* (teilweise auch *Kostenfunktion* genannt) und der zugehörigen *Nebenbe-*

dingungen oder *Randbedingungen*. Des Weiteren muss die Problemklasse identifiziert werden, um aus den verschiedenen bereits existierenden Lösungsverfahren und zugehörigen Lösungsalgorithmen die passende Methode zum Lösen des Problems verwenden zu können. Die Lösungsverfahren sind an die jeweilige Problemklasse angepasst und basieren in der Regel auf analytischen Ansätzen, die mithilfe von algebraischen Umformulierungen *Optima* – je nach Verfahren auch *Extremwerte* genannt – der Zielfunktion errechnen und bewerten. Insbesondere sollte auch bei der Problemformulierung selbst die jeweils verfolgte Lösungsmethode berücksichtigt werden.

Generell werden in einem analytischen Lösungsverfahren die enthaltenen Variablen so bestimmt, dass sie eine aufgestellte mathematische *Kosten- bzw. Zielfunktion* über einer definierten Menge an prinzipiell zulässigen Variablenwerten minimieren bzw. maximieren. Dies kann bei gegebener Zielfunktion z.B. durch Gradienten- oder Liniensuchverfahren erfolgen. Die Menge der zulässigen Variablenbelegungen kann weiterhin durch Nebenbedingungen, die in der Regel in Form von Gleichungen bzw. Ungleichungen formuliert sind, eingeschränkt werden. Eine detaillierte Beschreibung kann [30] entnommen werden.

6.1.1.1 Zielfunktion

Wie zuvor bereits erwähnt, muss die Zuordnung von Applikationen zu Ressourcen, Betriebsmodi und Device-Komponenten in Form von Binärvariablen realisiert werden, damit diese eindeutig und somit als Entscheidungsvariablen interpretiert werden können. Eine Applikation kann dabei nur auf einer Ressource und in einem Betriebsmodus ausgeführt werden, dabei aber mehrere Device-Komponenten nutzen. Die Zuordnung einer Applikation zu einem bestimmten Betriebsmodus impliziert, dass der Betriebszustand während der Ausführung einer Applikation nicht gewechselt werden kann. Diese Einschränkung ist durch die dadurch erzielte Verringerung der Problemkomplexität begründet, da sonst innerhalb der Applikationen zusätzlich noch über die möglichen Betriebsmodi iteriert werden müsste. Durch die Optimierung zur Laufzeit, die in Abschnitt 6.2 diskutiert wird, wird diese Einschränkung zumindest teilweise kompensiert, da die Ausführung von Tasks innerhalb der Applikationen betrachtet wird. Die Ausführungszeiten für jede Applikation in jedem möglichen Modus und die Wechselzeiten zwischen aufeinanderfolgenden Modi sind durch die technische Spezifikation gegeben – die Break-Even-Zeiten, also die akkumulierte Zeitspanne, nachdem sich die Aufwände für das Umschalten der Modi bzw. Ein- und Ausschalten der jeweiligen Ressourcen amortisieren würde, können daraus trivial berechnet werden. Um von einer geteilten Device-Nutzung profitieren zu können, kann es zur Energieeinsparung sinnvoll sein, die Ausführung einer Applikation zu verzögern. Dazu ist eine weitere Zeitinformation nötig, die ebenfalls optimiert wird. Ohne Beschränkung der Allgemeinheit wird hier die Startzeit einer Applikation gewählt. Die korrespondierende Endzeit der Applikation kann implizit durch die Summe der Startzeit und der Ausführungszeit bestimmt werden. Diese Ausführungszeiten werden im Folgenden in Form von Zeitwerten verwendet und werden direkt anhand der benötigten Zyklen zur Ausführung der Applikation und der vom jeweiligen Betriebsmodus abhängigen

Zykluszeit berechnet. Da sich alle Zeitpunkte ebenfalls auf Zyklen beziehen, ist der Endzeitpunkt einer Applikation immer ungleich des Startzeitpunkts der folgenden Applikationen auf derselben Ressource.

Der Lösungsvektor setzt sich somit aus den Startzeiten der Applikationen, im Folgenden mit s bezeichnet, und der Applikation-Mode-Ressourcen-Zuordnung, im Folgenden mit x bezeichnet, zusammen. Es erfolgt implizit also eine Ablaufplanung der auszuführenden Applikationen. In der Zielfunktion werden sowohl DVFS-, als auch DPM-Mechanismen berücksichtigt. Erstere beeinflussen hauptsächlich die Energie aufgrund dynamischer Leistungsaufnahme, letztere die Energie aufgrund statischer Leistungsaufnahme. Da Device-Komponenten in der Regel nur einen einzigen Betriebsmodus besitzen, geht hier nur die Energie in die Bewertung mit ein, die von der statischen Leistungsaufnahme stammt. Jedoch muss auf die mögliche geteilte Nutzung von Devices geachtet werden. Außerdem werden die Kosten für einen Moduswechsel in der Zielfunktion repräsentiert.

Die Gesamtkosten für die Abarbeitung einer gegebenen Menge an Applikationen innerhalb eines definierten Zeitraums auf einer begrenzten Menge zur Verfügung stehender Verarbeitungsressourcen und unter etwaiger Zuhilfenahme einer ebenso begrenzten Menge an Device-Komponenten setzen sich zusammen aus der Energie aufgrund dynamischer Leistungsaufnahme E^{dyn}, die bei der Ausführung der Applikationen auf den Verarbeitungsressourcen verbraucht wird, der Energie E^{sw} für mögliche Moduswechsel, der Energie aufgrund statischer Leistungsaufnahme E^{stat}, die immer dann anfällt, wenn Verarbeitungsressourcen angeschaltet sind, und der Device-Energie E^{dev}, die verbraucht wird, wenn eine oder mehrere zusätzliche Komponenten während einer Applikationsausführung aktiv sein müssen.

$$\min E^{gesamt} = E^{dyn} + E^{sw} + E^{stat} + E^{dev} \tag{6.1}$$

Bei der Optimierung der Energieeffizienz sollen diese Energiegesamtkosten E^{gesamt} minimiert werden, was durch die Gleichung 6.1 ausgedrückt wird. Wie sich die einzelnen Bestandteile dieser Energiegesamtkosten zusammensetzen, wird in den nachfolgenden Paragraphen beschrieben. Zunächst soll jedoch der Begriff einer *Hyper-Periode* definiert werden, die im weiteren Verlauf als Periodengrenze verwendet wird.

Definition 6.1 (Hyper-Periode)
Eine Hyper-Periode beschreibt das kleinste gemeinsame Vielfache der Perioden aller Applikationen bzw. Tasks. Die Anzahl der notwendigen Ausführungen einer Applikation bzw. eines Tasks vervielfacht sich entsprechend des Anteils der Periode der Applikation bzw. des Tasks an der Hyper-Periode.

Energie aufgrund dynamischer Leistungsaufnahme Der Anteil der dynamischen Energie an den Energiegesamtkosten ist abhängig von der Entscheidung, ob eine Applikation $i \in A$ für ihre Ausführungszeit auf einer Ressource $r \in R$ ausgeführt wird. Weiterhin ist entscheidend, in welchem Betriebs- bzw. Power-Modus $m \in PM$ diese Applikation i ausgeführt wird. Insgesamt ergibt sich die Berechnung des Energieverbrauchs aufgrund dynamischer Leistungsaufnahme durch Gleichung 6.2.

Im Zusammenhang mit den Konstanten $t_{i,m,r} \in \mathbb{R}_0^+$ für die Ausführungszeit von Applikation i und der damit verbundenen dynamischen Leistung $P_{i,m,r}^{dyn}$ jeweils in Abhängigkeit von der ausführenden Ressource r und dem angewandten Power-Modus m sorgt die Entscheidungsvariable $x_{i,m,r} \in \{0, 1\}$ dafür, dass bei einer Belegung $x_{i,m,r} \neq 0$ ein positiver Beitrag zur dynamischen Energie erfolgt. Ist jedoch $x_{i,m,r} = 0$, entfällt der Beitrag für diese Zuordnung von Applikation auf Ressource und Power-Modus.

$$E^{dyn} = \sum_m \sum_r \sum_i x_{i,m,r} \cdot t_{i,m,r} \cdot P_{i,m,r}^{dyn} \tag{6.2}$$

Die Einträge in $t_{i,m,r}$ und $P_{i,m,r}^{dyn}$ können entweder komplett gemessen bzw. analytisch ermittelt, oder teilweise relativ zu einem Basis-Betriebszustand berechnet werden (vgl. Abschnitt 4.1.5). Für eine korrekte Anwendung von Gleichung 6.2 ist entscheidend, dass es für jede Applikation i genau eine Zuordnung zu einer Ressource und einem Modus gibt. Dieses wird durch die entsprechende Formulierung der Nebenbedingungen in Abschnitt 6.1.1.2 garantiert.

Energie für Moduswechsel Falls auf einer Ressource r zwei Applikationen i und j mit unterschiedlichen Ausführungsgeschwindigkeiten und somit unterschiedlichen Power-Modi m und n direkt nacheinander ausgeführt werden, so muss die Ressource zwischen diesen beiden Ausführungen einen Moduswechsel durchführen. Dieser verbraucht sowohl Zeit als auch Energie, wobei nur letzteres in der Kostenfunktion der Gesamtenergiekosten erfasst werden muss. Implizit ist die benötigte Zeit für den Moduswechsel im Energieverbrauch für einen Moduswechsel $E_{(m,r)\times(n,r)}^{sw}$ enthalten, der von der ausführenden Ressource r und den beiden Power-Modi $m, n \in PM$ abhängt. $E_{(m,r)\times(n,r)}^{sw}$ kann anhand der Spezifikation für jede Ressource r berechnet werden.

$$E^{sw} = \sum_r \sum_i x_{i,m,r} \cdot E_{(m,r)\times(n,r)}^{sw} \cdot x_{j,n,r} \quad \forall j : j \text{ ist direkter Nachfolger von } i \tag{6.3}$$

Weiterhin wird der Zeitverbrauch in den Nebenbedingungen berücksichtigt, sodass keine Verletzung der Performanzanforderungen durch auftretende Moduswechsel hervorgerufen werden kann.

Energie aufgrund statischer Leistungsaufnahme Der Verbrauch von Energie aufgrund statischer Leistungsaufnahme hängt von der Ressource ab und wird deshalb genau dann verbraucht, wenn die jeweilige Ressource nicht ausgeschaltet werden kann, also nicht in den Zustand z_{off} wechseln kann. Zum einen ist das natürlich dann der Fall, wenn gerade eine Applikation aktiv auf der Komponente ausgeführt wird. Zum anderen sollte eine Ressource nicht abgeschaltet werden, wenn die Zeit, in der die Ressource inaktiv ist, kleiner ist als die Zeit, die benötigt wird, um die Ressource erst auszuschalten und sie für die Ausführung der nächsten Applikation wieder anzuschalten. Diese Break-Even-Zeit t_r^{BE} kann für jede Ressource statisch berechnet werden. Der Anteil des Energieverbrauchs aufgrund statischer Leistungsaufnahme ist folglich in mehrere Bereiche untergliedert.

Da die Ressource während der Ausführung einer Applikation angeschaltet bzw. aktiviert sein muss und deswegen einen Energieverbrauch aufgrund statischer Leistungsaufnahme aufweist, erfolgt diese Formulierung analog zum Energieverbrauch aufgrund dynamischer Leistungsaufnahme durch Term 6.4, jedoch unter der Verwendung der statischen Leistungsaufnahme P_r^{stat}. Da die statischen Leistungsaufnahme durch vorhandene Leckströme hervorgerufen wird und deswegen technologiebedingt ist, hängt sie weder vom jeweiligen Betriebsmodus, noch von der gerade ausgeführten Applikation, sondern lediglich von der ausführenden Ressource r ab.

Um das potentielle Abschalten einer Ressource durch einen Vergleich mit deren Break-Even-Zeit bestimmen zu können, müssen zuerst die direkten Vorgänger- und Nachfolgerbeziehungen berechnet werden. Mit s_i bzw. e_i wird im Folgenden der Startzeitpunkt bzw. der Endzeitpunkt einer Applikation i bezeichnet. Die Differenz $(s_i x_{i,r} - e_j x_{j,r})$ ermittelt alle Zeitspannen zwischen dem Ende einer gewählten Applikation j und dem Beginn von Applikation i auf Ressource r. Die Lösungsvariable $x_{i,r}$, die in dem relevanten Term 6.5 verwendet wird, markiert hierbei die Zuordnung einer Applikation i zu einer Ressource r ohne Berücksichtigung des gewählten Betriebsmodus. Die direkte Vorgänger-Applikation von i bildet den kleinsten positiven Wert aus dieser Menge aller Differenzen. Um diesen Wert zu extrahieren, werden zuerst durch $(1 - x_{j,r}) \cdot t^H$ alle Werte in den negativen Wertebereich verschoben, die nicht der Ressource von Applikation j zugeordnet sind. t^H bezeichnet hierbei die maximale (Hyper-)Periode, nach der sich der gesamte Ablaufplan periodisch wiederholt. Diese Differenz ist aufgrund der Annahme, dass der Endzeitpunkt einer Applikation j immer ungleich dem Startzeitpunkt einer nachfolgenden Applikation i ist, wenn diese auf der gleichen Ressource ausgeführt werden. Das Maximum über alle j des Kehrwerts bestimmt den gesuchten Wert, über den erneut der Kehrwert gebildet werden muss, um den ursprünglichen Zahlenwert zu bestimmen. Da bei allen Berechnungen eine globale Sicht auf das Hardware/Software-System existiert, muss weiterhin eine Multiplikation mit $x_{i,r}$ durchgeführt werden, da ebenfalls positive Werte geliefert werden, wenn i nicht auf Ressource r ausgeführt wird. Für ein korrektes Ergebnis dürfen diese aber an dieser Stelle nicht in die Energie aufgrund statischer Leistungsaufnahme eingerechnet werden.

Die Anwendung der Maximumsfunktion mit 0 eliminiert negative Werte, die entstehen, wenn i vor j ausgeführt wird. Durch den Doppelbruch in Term 6.5 wird schließlich der kleinste positive Wert aller Differenzen zwischen dem Ende von Applikation j und dem Start von Applikation i ermittelt. Das Minimum aus diesem und der Break-Even-Zeit t_r^{BE} ergibt die Zeit, für die zwischen zwei Applikationen Energie verbraucht wird – entweder der berechnete Wert aus $(s_i x_{i,r} - e_j x_{j,r})$ oder die Break-Even-Zeit, die trotz zwischenzeitlichem Ausschalten der Ressource anfällt. Die Multiplikation mit der statischen Verlustleistung P_r^{stat} liefert den entsprechenden Energieverbrauch.

Bisher wurden jedoch nur die Vorgänger- und Nachfolgerbeziehungen innerhalb einer Periode beachtet. Da die Ausführung der ersten Applikation einer Periode aber nicht mit deren Beginn zusammenfallen muss – gleiches gilt für das Ende – kann eine Ressource bzw. ein Device möglicherweise über die Periodengrenze t^H hinweg abgeschaltet werden. Diese Möglichkeit wird durch Term 6.6 berechnet. Dazu muss die erste Startzeit und die letzte Endzeit für die Ressource r ermittelt und die Differenz

mit t_r^{BE} verglichen werden. Weil das Minimum verwendet wird, muss der Unterschied zwischen Nullen, die aufgrund von $x_{i,r} = 0$ entstehen und darum nicht berücksichtigt werden dürfen, und Nullen bei den Startzeiten s_i beachtet werden. Deshalb wird zuerst die Periode t^H von s_i subtrahiert, dann mit $x_{i,r}$ multipliziert und schließlich wird t^H zur Korrektur wieder addiert. Das Ergebnis ist s_i für $x_{i,r} = 1$ und t^H für $x_{i,r} = 0$. Beim Applikationsende kann darauf verzichtet werden, da die Ausführungszeit einer Applikation immer größer Null ist. Es kann also direkt mit $x_{i,r}$ multipliziert werden. Das Maximum dieses Produkts muss noch von t^H subtrahiert werden. Das Minimum aus diesem Wert und t_r^{BE} ergibt die aktive Zeit zwischen zwei Zyklen, im Fall der Entscheidung zum zwischenzeitlichen Abschalten der Ressource mindestens deren Break-Even-Zeit. Um die benötigte Energie zu erhalten, wird diese Zeit mit P_r^{stat} multipliziert.

Insgesamt kann also die Energie E^{stat}, die aufgrund statischer Leistungsaufnahme verbraucht wird, durch die folgenden Terme 6.4 bis 6.6 formuliert werden.

$$E^{stat} = \underbrace{\sum_r \sum_m \sum_i x_{i,m,r} \cdot t_{i,m,r} \cdot P_r^{stat}}_{\text{Energie für Applikationen}} \tag{6.4}$$

$$+ \underbrace{\sum_r \sum_i \min\left(t_r^{BE}, \max\left(0, \underbrace{\frac{x_{i,r}}{\max_j \left(\frac{1}{s_i x_{i,r} - e_j x_{j,r} - \left(1 - x_{j,r}\right) \cdot t^H} \right)}}_{\text{Zeit von Applikationsende } j \text{ bis Applikationsanfang } i} \right) \right) \cdot P_r^{stat}}_{\text{Energie zwischen Applikationen}} \tag{6.5}$$

$$+ \underbrace{\sum_r \min\left(t_r^{BE}, \min_i \left(\left(s_i - t^H \right) \cdot x_{i,r} + t^H \right) + t^H - \max_i \left(e_i x_{i,r} \right) \right) \cdot P_r^{stat}}_{\text{Energie vor erster und nach letzter Applikation}} \tag{6.6}$$

Device-Energie Für die gesamtheitliche Betrachtung und Optimierung des Energieverbrauchs eines eingebetteten Hardware/Software-Systems ist aber nicht nur die Energie der Verarbeitungsressourcen bzw. Prozessoren von Bedeutung, sondern zunehmend auch die Energie, die bei der Benutzung von On-Chip- bzw. Off-Chip-Komponenten, I/O-Komponenten, Speicherkomponenten, Peripherie-Komponenten, u.a. benötigt wird. Zum einen liegt dies an der Anwendung der hier dargestellten eingebetteten Hardware/Software-Systeme in Bereichen, die durch eine hohe Berechnungs- bzw. Kommunikationsaktivität und Interaktion mit der Umgebung geprägt sind. Zum anderen an der Möglichkeit moderner Hardware-Architekturen, Bearbeitungsprozesse auf Komponenten auszulagern, um den Hauptprozessor zu entlasten, sodass sich dieser

länger in einem Modus mit verminderter Leistungsaufnahme befinden kann.

Für die Formulierung des Optimierungsproblems ergibt sich daraus, dass einige der für Verarbeitungsressourcen geltenden Randbedingungen für die eben genannten Komponenten entfallen. Diese können im Gegensatz zu Ressourcen durch verschiedene Applikationen parallel genutzt werden, wie es beispielsweise bei Anzeigekomponenten oder Sensoren durchaus der Fall ist. Die Schwierigkeit bei der Formulierung der durch Devices verbrauchten Energie liegt deshalb in der Parallelität, da mehrere Applikationen ein Device gleichzeitig aktivieren können und es somit zu zeitlichen Überlagerungen bei der Nutzung kommen kann. In diesem Fall darf die Device-Energie jedoch nicht mehrfach berechnet werden.

Die Berechnung der Energie E^{dev}, die durch Device-Komponenten während der Ausführung der Funktionalität verbraucht wird, ist durch die Terme 6.7 bis 6.11 beschrieben. Die von den Applikationen und dem Hardware/Software-System vorgegebene Zuordnung einer Device-Komponente d zu einer Applikation i wird dabei durch die Variable $z_{i,d}$ festgelegt.

Die Formulierung der Device-Energie erfolgt im Prinzip analog zu Term 6.5, jedoch wird statt des Startzeitpunkts s_j der Endzeitpunkt e_j verwendet. Das ist dadurch begründet, dass zunächst davon ausgegangen wird, dass beteiligte Devices nicht abgeschaltet werden können. Der Energieverbrauch eines Devices wird also berechnet, indem es zwischen je zwei zugeordneten Applikationen angeschaltet bleibt. Falls das entsprechende Device doch abgeschaltet werden kann, wird in einem separaten Schritt die Zeit zwischen den Applikationen unter Berücksichtigung der Break-Even-Zeit wieder abgezogen und der Energieverbrauch, der durch die Devices verursacht wird, dementsprechend reduziert. Es wird somit die Differenz aus $e_i z_{i,d}$ und $e_j z_{j,d}$ gebildet, der Korrekturterm $(1 - z_{j,d}) \cdot t^H$ setzt die Werte der Applikationen, die die entsprechende Device-Komponente nicht nutzen, auf $-t^H$. Der nächste Endzeitpunkt einer Applikation wird als das Maximum des Kehrwerts bestimmt, die erneute Bildung des Kehrwerts liefert den korrekten Zahlenwert dieses Abstands. Da die Nutzung eines Device durch eine Applikation unabhängig von der ausführenden Ressource ist, ist eine Multiplikation mit $z_{i,d}$ nicht unbedingt nötig, wird aber aus Konsistenzgründen zu den Termen 6.5 und 6.8 durchgeführt. Durch Anwendung der Maximumsfunktion mit 0 werden negative Werte eliminiert, die entstehen, wenn Applikation j vor Applikation i ausgeführt wird. Die Vorgänger-Nachfolger-Beziehung ist in diesem Fall vertauscht, sodass der richtige Wert bei der Betrachtung von i und j gefunden wird.

Wird eine Device-Komponente über eine bestimmte Zeit nicht benötigt, kann diese abgeschaltet werden. Diese Bereiche werden von der Gesamt-Device-Energie durch Term 6.8 subtrahiert. Wie bei der statischen Energie in Term 6.5 wird der nächste Startzeitpunkt als Kehrwert von $(s_i z_{i,d} - e_j z_{j,d} - (1 - z_{j,d}) \cdot t^H)$ gefunden. Der Rest des Terms ist äquivalent zu Term 6.4. Die Break-Even-Zeit t_d^{BE} wird subtrahiert, denn ein Abschalten der Device-Komponente d lohnt sich aus Sicht einer Energieoptimierung erst, wenn die Zeit zwischen Applikationen, die das Device d benutzen, größer als diese Break-Even-Zeit ist.

$$E^{dev} = \sum_d P_d^{dev} \cdot \left(\sum_i \underbrace{\max\left(0, \frac{z_{i,d}}{\max_j \left(\frac{1}{e_i z_{i,d} - e_j z_{j,d} - \left(1 - z_{j,d}\right) \cdot t^H} \right)} \right)}_{\text{Zeit von Applikationsende } j \text{ bis Applikationsende } i} \right. \tag{6.7}$$

$$\underbrace{- \max\left(0, \frac{z_{i,d}}{\max_j \left(0, \frac{1}{s_i z_{i,d} - e_j z_{j,d} - \left(1 - z_{j,d}\right) \cdot t^H} \right)} - t_d^{BE} \right)}_{\text{Zeit von Applikationsende } j \text{ bis Applikationsanfang } i} \tag{6.8}$$

$$\cdot \min_j \left(\underbrace{\frac{\max\left(0, \left(e_i z_{i,d} - e_j z_{j,d}\right)\left(e_i z_{i,d} - s_j z_{j,d}\right)\right) \cdot z_{i,d}}{\left(e_i z_{i,d} - e_j z_{j,d} - \left(1 - z_{j,d}\right) \cdot t^H\right)\left(e_i z_{i,d} - s_j z_{j,d} - \left(1 - z_{j,d}\right) \cdot t^H\right)}}_{\text{abschalten, wenn keine Applikation zwischen Ende von } j \text{ und Anfang von } i} + \left(1 - z_{i,d}\right) \right) \Bigg) \tag{6.9}$$

$$+ \sum_d P_d^{dev} \cdot \left(\underbrace{\min_i \left(\left(\left(e_i - t^H\right) \cdot z_{i,d} \right) + t^H \right) - \min_i \left(\left(\left(s_i - t^H\right) \cdot z_{i,d} \right) + t^H \right)}_{\text{Zeit vom ersten Beginn einer Applikation bis zum ersten Ende}} \right. \tag{6.10}$$

$$\left. + \underbrace{\min\left(t_d^{BE}, \min_i \left(\left(\left(s_i - t^H\right) z_{i,d} \right) + t^H \right) + t^H - \max_i \left(e_i z_{i,d}\right) \right)}_{\text{Zeit zwischen letztem Ende und erstem Beginn über Zyklusgrenze hinweg}} \right) \tag{6.11}$$

Bisher werden aber noch keine zeitlich überlagerten Benutzungen durch mehrere Applikationen berücksichtigt. Ein Beispiel dazu ist in Abbildung 6.1 dargestellt. In diesem Fall würde Device-Komponente d, deren Abhängigkeit durch einen Pfeil dargestellt ist, abgeschaltet werden, wenn der Abstand zwischen den Applikationen i und j größer als die Break-Even-Zeit von d wäre. Dies muss aber verhindert werden, da Applikation k ebenfalls d während dieses Zeitraums benötigt. Hierzu wird ein Faktor in Term 6.9 eingeführt, mit dem Term 6.8 multipliziert wird. Dieser muss den Wert 1 besitzen, wenn ein Abschalten möglich ist, ansonsten 0. Im Zähler des Bruchs steht das Maximum aus 0 und dem Produkt aus $(e_i z_{i,d} - e_j z_{j,d})$ und $(e_i z_{i,d} - s_j z_{j,d})$. Wenn eine zeitliche Überlagerung von i durch eine Applikation j vorliegt und beide das gleiche

Device nutzen, wird der Term negativ. Durch die Bildung des Maximums werden die entsprechenden Werte auf 0 gesetzt. In diesem Fall sind die zugehörigen Werte des Nenners belanglos, sie müssen jedoch aus Gründen der mathematischen Korrektheit ungleich Null sein. Deshalb muss von beiden Differenzen noch $(1 - z_{j,d}) \cdot t^H$ subtrahiert werden. Wenn keine zeitliche Überlagerung vorliegt, sind Zähler und Nenner gleich und der Bruch nimmt somit den Wert 1 an. Wie schon vorher erwähnt entsteht der Wert 0 auch, wenn eine Applikation keine oder eine andere Device-Komponente nutzt. Diese müssen durch die Addition von $(1 - z_{i,d})$ auf 1 gesetzt werden, da sie keine Bedeutung für das Abschalten haben sollen. Wenn aber der Wert 0 für alle j entsteht, also eine zeitliche Überlagerung vorliegt, so darf nicht abgeschaltet werden. Deshalb wird das Minimum des beschriebenen Terms gesucht und mit P_d^{dev} multipliziert.

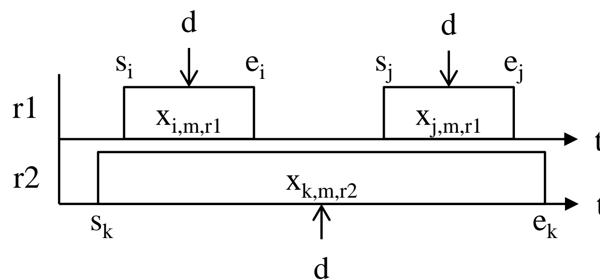


Abbildung 6.1: Zeitlich überlagerte Benutzung einer geteilten Device-Komponente durch mehrere Applikationen

Die beschriebene Herangehensweise beginnt mit dem ersten Applikationsende, sodass die Kosten vom ersten Startzeitpunkt bis zu diesem Endzeitpunkt gesondert durch Term 6.10 berechnet werden müssen. Dabei müssen die beiden Referenzpunkte nicht zwangsläufig zur gleichen Applikation gehören. Von e_i wird dazu t^H subtrahiert. Der Term wird mit $z_{i,d}$ multipliziert und schließlich wird t^H korrigierend wieder addiert, um mit dem Minimum den kleinsten Endzeitpunkt für die entsprechende Device-Komponente d zu finden. Dieser Aufwand ist nötig, um die oben beschriebenen verschiedenen Arten des Werts 0 zu trennen. Hinzu kommen noch die Zyklusübergänge, für die eine analoge Formulierung zu Term 6.6 verwendet wird. Da es sich hierbei um Device-Komponenten und nicht um Verarbeitungsressourcen handelt, muss in Term 6.11 $x_{i,r}$ durch $z_{i,d}$ und t_r^{BE} durch t_d^{BE} ersetzt werden. Der resultierende Wert wird anschließend mit der Leistungsaufnahme der jeweiligen Device-Komponente P_d^{dev} multipliziert, um die Device-Energie zu erhalten.

6.1.1.2 Nebenbedingungen

Die Menge der zulässigen Variablenbelegungen kann durch die Formulierung von Nebenbedingungen in Form von Gleichheits- oder Ungleichheitsbeziehungen eingeschränkt werden. In der Regel wird die Komplexität der zu lösenden Optimierungsprobleme dadurch vermindert. Die Nebenbedingungen stellen die Eigenschaften dar, die von der Lösung des Problems eingehalten werden müssen. Für modellierte

Hardware/Software-Systeme können dadurch physikalische, logische und funktionale Eigenschaften repräsentiert werden.

Eindeutigkeit der Zuordnungen Die Zuordnung von Applikationen zu Verarbeitungsressourcen und Betriebsmodi ist als logische Verknüpfung zu interpretieren, also als Binärvariable. Der Wertebereich für Lösungsvektors x, der diese Zuordnung repräsentieren soll, wird deshalb durch Bedingung 6.12 auf $0, 1$ beschränkt.

$$x \in \{0, 1\} \text{ Integer} \tag{6.12}$$

$$\sum_{m,r} x_{i,m,r} = 1 \quad \forall i \in A \tag{6.13}$$

Weiterhin soll die Zuordnung eindeutig sein, d.h. es ist nicht möglich, dass eine Applikation gleichzeitig in zwei verschiedenen Betriebsmodi oder auf zwei verschiedenen Ressourcen ausgeführt wird. Für ein i aus der Menge der Applikationen A darf unter der gegebenen Integerbedingung also nur ein Eintrag in x den Wert 1 besitzen für alle m und r , der Rest muss 0 sein. Dies lässt sich als Summe wie durch Bedingung 6.13 formuliert darstellen.

Gegenseitige Abhängigkeiten Der Lösungsvektor enthält neben der Zuordnung von Applikationen zu Verarbeitungsressourcen und Betriebsmodi auch die Startwerte s_i der Applikationen. Daraus lassen sich durch Multiplikation von $x_{i,m,r}$ mit $t_{i,m,r}$ die in der Kostenfunktion oftmals verwendeten Endzeitpunkte e_i berechnen, was durch Bedingung 6.14 gewährleistet ist.

$$e_i = s_i + t_{i,m,r} \cdot x_{i,m,r} \quad \forall i \in T \tag{6.14}$$

$$s_i \geq 0 \quad \forall i \in I \tag{6.15}$$

$$s_j > e_i \quad \forall i, j : j \text{ folgt auf } i \tag{6.16}$$

$$e_i \leq t_H \quad \forall i \in I \tag{6.17}$$

Außerdem müssen eventuell bestehende gegenseitige Datenabhängigkeiten der Applikationen berücksichtigt werden. Dazu werden die entsprechenden Start- und Endzeitpunkte in Relation zueinander gebracht. Der früheste Beginn einer Applikation ist der Zeitpunkt 0, das späteste mögliche Ende einer Applikation ist ihre Deadline bzw. die daraus ableitbare (Hyper-)Periode t^H. Dies wird durch die beiden Bedingungen 6.15 und 6.17 ausgedrückt. Dazwischen liegend werden Relationen wie in Bedingung 6.16 beschrieben – der Startzeitpunkt s_j der nächsten Applikation muss größer dem Endzeitpunkt seines Vorgängers sein.

Sollen zwischen datenabhängigen Applikationen auf verschiedenen Ressourcen auch Kommunikationsaufwände für die Übertragung dieser Daten berücksichtigt werden, und können diese Aufwände zeitlich abgeschätzt werden, z.B. durch eine Worst-Case-Kommunikationszeit, können diese zum Startzeitpunkt der nachfolgenden Applikation in Bedingung 6.16 addiert werden, sodass sich der früheste Startzeitpunkt entsprechend verschiebt.

Auch paralleles Verhalten kann durch diese Formulierungen vorgegeben werden. Sollen mehrere Applikationen parallel ausgeführt werden, müssen die Startzeitpunkte und Endzeitpunkte gleichgesetzt werden, sodass vom Lösungsalgorithmus nur solche Lösungen ausgewählt werden können, die diese Applikationen zur gleichen Zeit auf unterschiedlichen Ressourcen ausführen.

Zeitliche Überlagerungen von Ausführungen und Moduswechsel Bei der Zuordnung von Applikationen zu Verarbeitungsressourcen und Power-Modes muss verhindert werden, dass sich Applikationen auf derselben Ressource zeitlich überlagern. Weiterhin benötigt ein etwaiger Moduswechsel zwischen der Ausführung von Applikationen eine gewisse Umschaltzeit, während der keine Applikation auf der Ressource ausgeführt werden darf. Um die genauen Kosten für diese Übergänge im Sinne der zugrunde liegenden Kostenfunktion zu ermitteln, muss der jeweilige Modus vor und nach dem Wechsel identifiziert werden. Die Kombination zweier aufeinanderfolgender Modi wird durch eine quadratische Funktion abgebildet, das daraus resultierende Teilproblem bzw. die daraus ableitbare Bedingung ist also nicht linear.

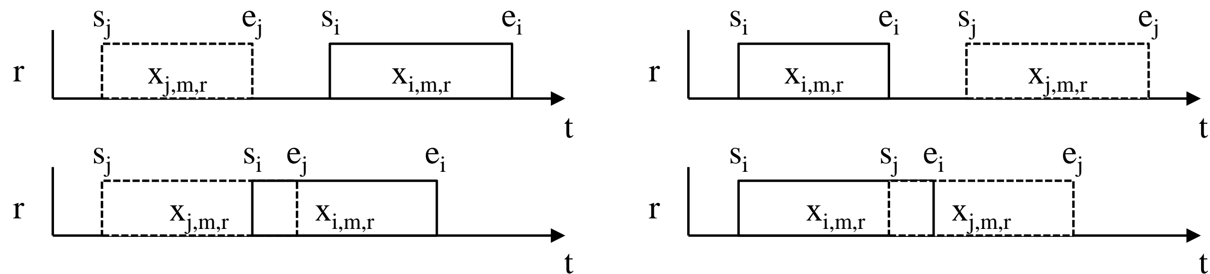


Abbildung 6.2: Zeitliche Überlagerung auf Ressource r von vorne bzw. von hinten

Abbildung 6.2 stellt die beiden Möglichkeiten der zeitlichen Überlagerung von Applikationen auf der gleichen Ressource dar – von vorne oder von hinten. Beide Fälle können mit einer Bedingung geprüft werden, wenn wiederum alle Kombinationen aus i und j überprüft werden, da dann auch immer der Fall abgedeckt ist, in dem Applikation i in Mode m und Applikation j in Mode n die dargestellte Abfolge haben.

$$\underbrace{\left(e_i x_{i,r} + t^{sw}_{m,n} x_{i,r} - s_j x_{j,r}\right)}_{\leq 0 \text{ wenn genügend Zeit für Moduswechsel}} \underbrace{\left(e_j x_{j,r} - s_i x_{i,r}\right)}_{\geq 0 \text{ wenn } i \text{ vor } j} \cdot x_{i,r} \leq 0 \tag{6.18}$$

$$\underbrace{\left(e_i x_{i,r} + t^{sw}_{m,n} x_{i,r} - s_j x_{i,r} - t_h\right)}_{\leq 0 \text{ wenn genügend Zeit für Moduswechsel}} \underbrace{\left(e_j x_{i,r} + t_H - s_i x_{i,r}\right)}_{\geq 0 \text{ wenn } i \text{ vor } j} \cdot x_{i,r} \leq 0 \tag{6.19}$$

$$\forall i, j \in A \quad \forall r \in R$$

Mit dem Teil-Term $(e_i x_{i,r} + t^{sw}_{m,n} x_{i,r} - s_j x_{i,r})$ in Nebenbedingung 6.18 wird nur der Fall einer zeitlichen Überlagerung von hinten überprüft. Implizit wird dabei ebenfalls geprüft, ob die Zeit für einen potentiellen Moduswechsel t^{sw} zwischen den Applikationen i und j ausreicht. Der Term ist negativ, wenn die Bedingung erfüllt ist, ansonsten positiv.

Wenn jedoch eine zeitliche Überlagerung von vorne vorliegt, wird dieser Term auch positiv, wenn keine Verletzung der Bedingung zur zeitlichen Überlagerung vorliegt. Um diesen Fall auszuschließen, wird der Term mit $(e_j x_{i,r} - s_i x_{i,r})$ multipliziert. Dieser ist für die Ausführung von i vor j positiv, ansonsten negativ. Die Multiplikation mit $x_{i,r}$ verhindert, dass Applikationen verglichen werden, die nicht auf der entsprechenden Ressource r ausgeführt werden. Eine Überlagerung über die Periodengrenze hinweg kann nicht vorkommen, da dies schon durch die Bedingungen 6.17 ausgeschlossen ist. Die Zeiten für einen Moduswechsel sind in diesem Fall nicht berücksichtigt. Da in Nebenbedingung 6.18 jedoch bereits alle Kombinationen von Applikationen geprüft werden, kann in einer zusätzlichen Bedingung 6.19 auch die Periode t_H zu den Start- und Endzeitpunkten von Applikation j addiert werden, was nur für die letzte Applikation in einer betrachteten Periode relevant ist.

6.1.2 Lösung des Optimierungsproblems

Eine effiziente und akkurate Lösung von Optimierungsproblemen hängt nicht nur von der Art und Anzahl der Lösungsvariablen ab, sondern vor allem auch von der Charakteristik der Zielfunktion und den Nebenbedingungen. Können beide durch lineare Funktionen beschrieben werden, handelt es sich um ein lineares Optimierungsproblem. Wird die Zielfunktion durch eine quadratische Funktion hinsichtlich der Lösungsvariablen beschrieben, handelt es sich um ein quadratisches Optimierungsproblem. Sind Zielfunktion oder Nebenbedingungen durch teilweise nichtlineare Terme formuliert, ist auch das Optimierungsproblem nichtlinear. Optimierungsprobleme, die in diese Komplexitätsklasse fallen, sind schwer zu lösen, da keine generell anwendbare analytische Methode existiert, die während der Optimierung garantiert zu einer Variablenbelegung führt bzw. die im Sinne der Zielfunktionsoptimierung besser ist als die ursprüngliche Variablenbelegung – geschweige denn, die ein globales Optimum finden kann.

Beliebige nichtlineare Optimierungsprobleme werden in der Regel durch ein Verfahren der nichtlinearen Programmierung (engl. *Non-Linear Programming* (NLP)) gelöst. In diesem Fall wird meist ein iteratives Vorgehen angewandt, bei dem in jeder Iteration eine Suchrichtung und eine daran angepasste Schrittweite berechnet wird. Dieser Berechnungsschritt wird im Allgemeinen durch das Lösen eines linearen oder quadratischen Teilproblems erreicht, das die ursprüngliche Zielfunktion möglichst genau approximiert. Somit wird eine schrittweise Annäherung an die nichtlineare Zielfunktion gewährleistet.

Zur Lösung der (teilweisen) ganzzahligen nichtlinearen Optimierung (engl.: *Mixed-Integer Non-Linear Programming (MINLP)*), wie es in diesem Fall vorliegt, soll in dieser Arbeit ein Branch-and-Bound-Verfahren angewandt werden, das auf dem eben dargestellten Muster der iterativen Annäherung basiert. Die Lösung des MINLP wird vereinfacht, indem es durch eine Relaxation in die einzelnen Bestandteile zerlegt wird, nämlich in ein ganzzahliges lineares Optimierungsproblem (engl.: *Mixed-Integer Linear Programming (MILP)*) und in die Lösung eines nicht-linearen Optimierungsproblems.

Die auf das vorliegende Optimierungsproblem angepassten Teilschritte – sowohl die Aufspaltung des Problems in Teilprobleme als auch die Berechnung geeigneter

Schranken – sollen im nächsten Abschnitt betrachtet werden. Weiterhin wird auf die Integration in einen Algorithmus zur Lösung des vorliegenden Problems der Energieeffizienzoptimierung näher eingegangen.

6.1.2.1 Aufspaltung in Teilprobleme

Der erste Schritt der Problemaufteilung (*Branching*) dient zur Einschränkung des Lösungsraums, indem die Freiheitsgrade des eigentlichen Optimierungsproblems reduziert werden. In der vorliegenden Optimierung beinhaltet der Lösungsvektor sowohl ganzzahlige Variablen zur Definition der Zuordnung von Applikationen zu Verarbeitungsressourcen und Power-Modi als auch reelle Variablen für die Definition der Startpunkte der Applikationen. Da die ganzzahligen Variablen in binärer Form vorliegen und als Entscheidungsvariablen für eine bestimmte Zuordnung interpretiert werden sollen, bieten sich diese Variablen zur Problemaufteilung an.

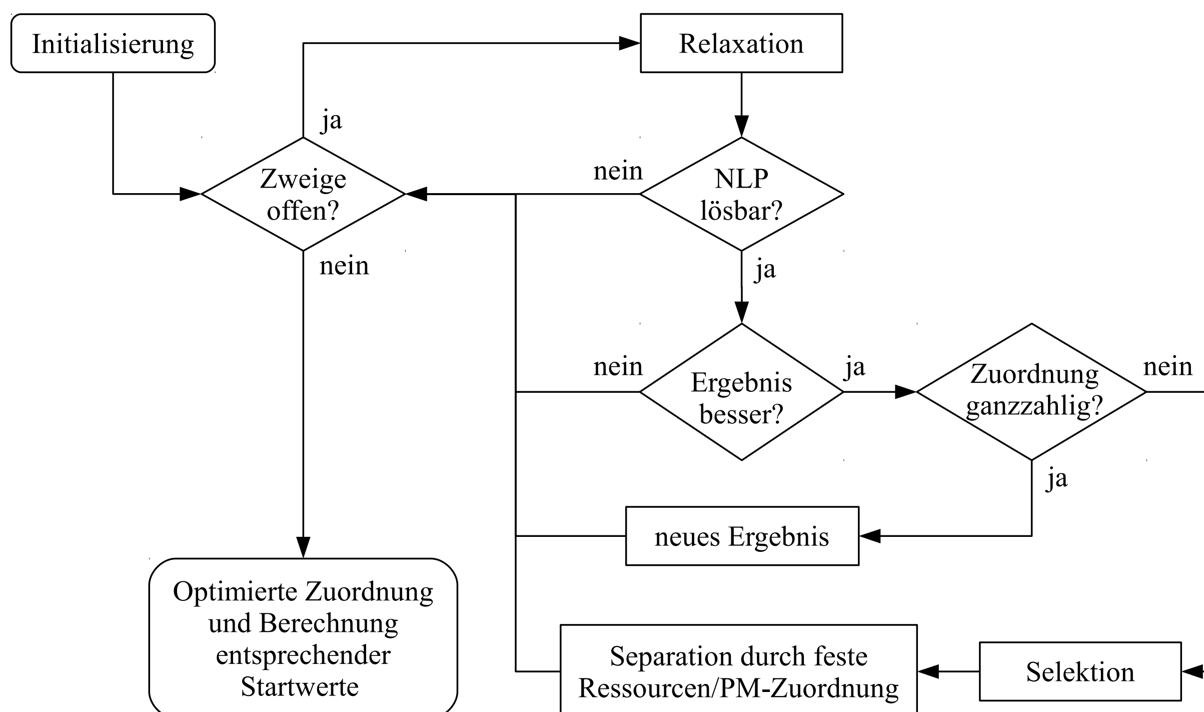


Abbildung 6.3: Prinzipieller Ablauf des zugrunde liegenden Optimierungsalgorithmus

Abbildung 6.3 stellt den Ablauf des Optimierungsalgorithmus dar, die in weiten Teilen die Grundprinzipien eines Branch-and-Bound-Verfahrens widerspiegelt, wie es bereits in Abschnitt 2.9.1 eingeführt wurde. In einem ersten Schritt werden die zulässigen Toleranzen für die Integerbedingung und die initiale Lösung festgelegt und daraus der initiale Ergebniswert berechnet. Danach wird bei allen Durchläufen – außer dem ersten – geprüft, ob offene und damit noch nicht geschlossene Zweige existieren. Falls keine offenen Zweige mehr existieren, wird die aktuell beste gültige Lösung als Ergebnis der Optimierung ausgegeben.

Wenn es allerdings noch offene Zweige gibt, wird das Problem zunächst vereinfacht,

indem in einer Relaxation auf die Integerbedingung 6.12 verzichtet wird. Das dadurch relaxierte Problem wird mithilfe eines nichtlinearen Lösungsverfahrens gelöst, das auf einer Approximation durch ein schrittweises quadratisches Problem basiert. Die Lösung des eingebetteten nichtlinearen Optimierungsproblems wird im nachfolgenden Abschnitt 6.1.2.2 noch im Detail erläutert. Ist das resultierende Problem nicht lösbar, wird dieser Zweig geschlossen. Das Problem kann in der Regel dann nicht gelöst werden, wenn bereits mehrere Lösungsvariablen festgelegt sind und durch diese Belegungen nicht mehr alle geltenden Nebenbedingungen erfüllt werden können. Wenn eine gültige Lösung existiert, wird das Resultat der nichtlinearen Optimierung mit dem aktuell besten Ergebnis verglichen. Falls das Resultat hinsichtlich der Zielfunktion schlechter ist, kann dieser Zweig geschlossen werden, da bei der Berücksichtigung der Integerbedingung jede weitere Lösung dieses Zweigs gleich oder noch schlechter sein würde. Damit existiert eine untere Schranke, sodass der komplette Teilbaum geschlossen werden kann, was abhängig von der momentanen Verzweigungstiefe zu einer deutlichen Reduktion der Komplexität führen kann. Wurde ein besseres Ergebnis als das aktuell gültige gefunden, muss überprüft werden, ob sich alle ganzzahligen Lösungsvariablen innerhalb der zu Beginn spezifizierten Toleranzwerte befinden, d.h. möglichst nahe an den Werten 0 oder 1, was bei Berücksichtigung der Integerbedingung eine gültige Lösung darstellen würde. Sind diese Toleranzwerte eingehalten, wird das momentane Resultat als beste gültige Lösung übernommen, und die noch offenen Zweige werden weiter abgearbeitet.

Werden die Toleranzwerte zumindest teilweise verletzt, beginnt die Selektionsphase. In dieser Phase wird die Lösungsvariable bestimmt, die die größte Abweichung erzeugt, wenn sie zu einer Ganzzahl gerundet wird.

In der anschließenden Separationsphase werden dann neue Zweige auf der in der Selektionsphase gewählten Variable entsprechend der Branching-Heuristik erzeugt. Die Zuordnung von Applikationen zu Verarbeitungsressourcen und Power-Modi erfüllt aufgrund der Nebenbedingungen die Kriterien eines Special-Ordered-Set-1. Laut der Kombination von Nebenbedingung 6.12 und 6.13 darf nämlich nur genau eine der Binärvariablen, die die Zuordnung anzeigen, den Wert 1 besitzen. Basierend auf der angewandten Branching-Heuristik werden die Werte des Special-Ordered-Sets ausgewählt, in denen der Wert jener Variablen enthalten ist, die in der vorangegangenen Selektionsphase bestimmt wurde. Das Special-Ordered-Set wird aufsteigend nach der Größe sortiert, wobei es sich aufgrund der Relaxation nicht um Ganzzahlenwerte, sondern um reelle Zahlen zwischen 0 und 1 handelt, deren Summe aufgrund der Bedingung 6.13 den Wert 1 ergeben. Zur oberen Hälfte dieser Werte y_k wird jeweils

$$\frac{1 - \sum_k y_k}{k}$$

addiert, wobei k der Menge an Werten im Special-Ordered-Set entspricht. Damit ergibt die Summe dieser Werte wieder den Wert 1, wobei die untere Hälfte der Werte bzw. deren obere Grenze auf den Wert 0 festgesetzt ist, sodass dieser Wert im daraus folgenden Zweig nicht mehr verändert werden kann. Dieses so veränderte Special-Ordered-Set bildet mit den restlichen unveränderten Werten den ersten Unterzweig. Der zweite

Zweig wird analog aus der unteren Hälfte des Special-Ordered-Sets erzeugt, wobei die obere Hälfte auf den Wert 0 gesetzt wird.

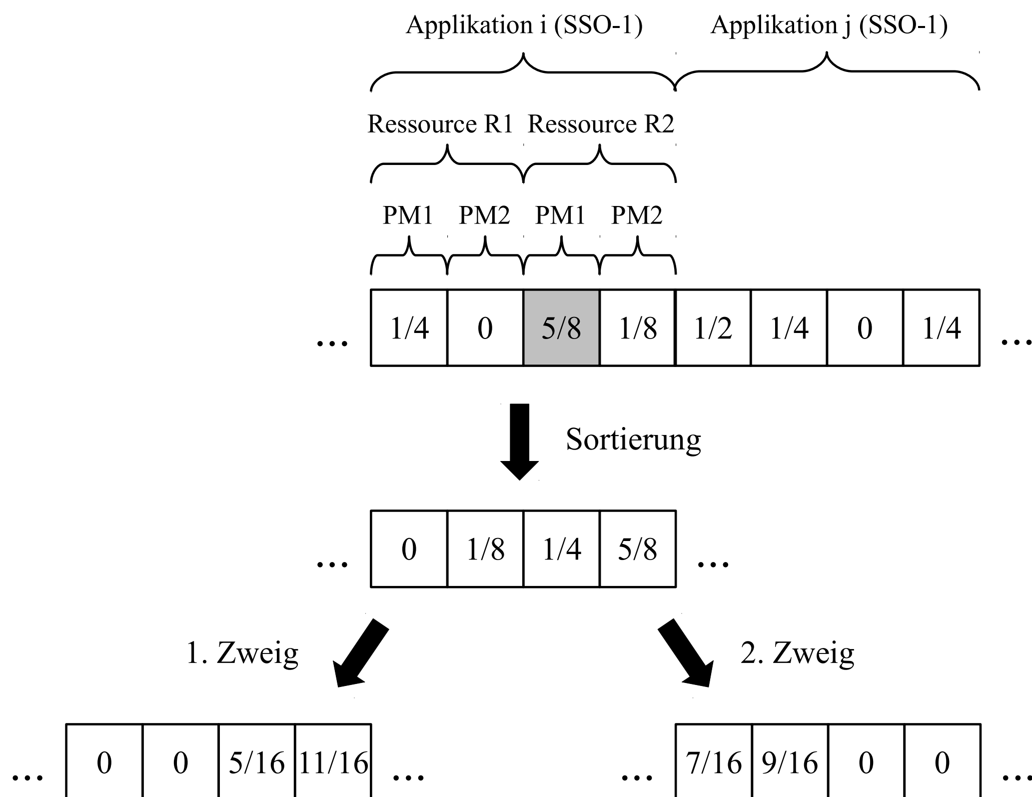


Abbildung 6.4: Separationsphase innerhalb des Branch-and-Bound-Verfahrens

Abbildung 6.4 soll die Separationsphase noch einmal bildlich verdeutlichen, wobei auf den grau markierten Wert verzweigt werden soll. Es wird das zu diesem Wert gehörige Special-Ordered-Set ausgewählt und der Größe nach sortiert. Für den ersten Unterzweig werden 0 und $1/8$ auf den Wert 0 gesetzt. Die Summe aus den beiden ursprünglichen Werten wird gleichmäßig auf die Werte der oberen Hälfte verteilt, sodass sich die Werte $1/4$ und $5/8$ zu $7/16$ und $9/16$ ändern. Für die Bildung des zweiten Unterzweigs werden $1/4$ und $5/8$ auf den Wert 0 gesetzt, sodass $(1/4 + 5/8)/2 = 7/16$ zu den Werten 0 und $1/8$ addiert werden müssen, um Nebenbedingung 6.13 zu erfüllen.

Algorithmus 6.1 zeigt den Ablauf des entwickelten Optimierungsalgorithmus noch einmal in Pseudocode-Darstellung.

6.1.2.2 Lösung des relaxierten Optimierungsproblems

Der Algorithmus zur Lösung des relaxierten nichtlinearen Problems ist modular in den im Abschnitt 6.1.2.1 beschriebenen Branch-and-Bound-Algorithmus integriert und könnte deshalb durch jeden nichtlinearen Lösungsalgorithmus mit ähnlichen Eigenschaften ersetzt werden. In diesem konkreten Fall wird ein Active-Set-Algorithmus aus dem Paket `fmincon` der *MATLAB Optimization Toolbox* zur Lösung des Problems verwendet. Grundlagenwissen über den Active-Set-Algorithmus findet sich in Abschnitt 2.10.3.3,

Algorithmus 6.1 Branch-and-Bound-Algorithmus mit allgemeiner Branching-Heuristik

```
x ∈ ℝ^{n+t}, F_x = {x_{0,...,n-1}}
z_opt ← ∞
x ← init_solution(x_init, xlb, xub)
x_opt ← x
S ← {x}
while S ≠ ∅ do
    x ← S.pop()
    xlb ← x.xlb; xub ← x.xub,
    z = solve_NLP(x, xlb, xub)
    if z < 0 then
        NLP unlösbar oder nicht konvergent (FR1)
    else if z ≥ z_opt then
        Wert der Zielfunktion größer als bisherige beste Lösung (FR2)
    else
        if abs(round(x_{0,...,n-1}) − x_{0,...,n-1}) < tol_x then
            x_opt ← x; z_opt ← z (FR3)
        else
            z_sep, x_sep, f_fix ← −1
            for all f ∈ F_x do
                x_sep_cand ← round_to_integer(x, f)
                z_sep_cand ← evaluate_objective(x_sep_cand)
                if z_sep_cand > z_sep then
                    x_sep ← x_sep_cand; z_sep ← z_sep_cand; f_fix ← f
                end if
            end for
            F_x = F_x \ {f_fix}
            x_SSO_sort ← sort_SSO(x_sep, f_fix)
            x_branch1 ← fix_upper_half(x_SSO_sort)
            x_branch2 ← fix_lower_half(x_SSO_sort)
            S. push(x_branch1, x_branch2)
        end if
    end if
end while
```

eine ausführliche Beschreibung der Implementierung dieses Algorithmus in der zugehörigen Dokumentation [1].

Da sowohl Kostenfunktion wie auch Nebenbedingungen nichtlineare Terme enthalten, muss für die Lösung des generellen Optimierungsproblems GP ein entsprechender Algorithmus verwendet werden. Der Active-Set-Algorithmus implementiert ein auf *Sequential Quadratic Programming (SQP)* basiertes Verfahren, um nichtlineare Optimierungsprobleme in einem iterativen Ansatz durch Approximation eines quadratischen Optimierungsproblems zu lösen. SQP-Verfahren gehören zu den effizientesten Lösungsformen für nichtlineare Probleme und wurden bereits in Verbindung mit grundlegenden Erklärungen zur Optimierungstheorie in Abschnitt 2.10 eingeführt. Als Grundlage dienen die *Karush-Kuhn-Tucker (KKT)*-Bedingungen, die notwendige Bedingungen dafür

sind, ein Optimum x^* zu finden. Falls die Zielfunktion und die Nebenbedingungen konvex sind, sind die KKT-Bedingungen sowohl notwendige als auch hinreichende Bedingungen für ein globales Optimum. Diese Voraussetzung ist bei den hier formulierten Kostenfunktionen und Nebenbedingungen allerdings nicht gegeben, sodass die Erreichung eines globalen Optimums nicht garantiert werden kann. Die daraus resultierenden Auswirkungen werden in Abschnitt 6.1.3 anhand verschiedener Experimente sichtbar.

6.1.3 Experimentelle Ergebnisse

Zur Durchführung synthetischer Experimente wird eine Modifikation der Implementierung aus [68] in MATLAB verwendet. Dabei soll in diesen Experimenten die Komplexität so gewählt und die für die Optimierung verwendeten Parameter mit vereinfachten Werten belegt werden, dass die Ergebnisse sowohl nachvollziehbar sind, als auch Schlussfolgerungen erlauben.

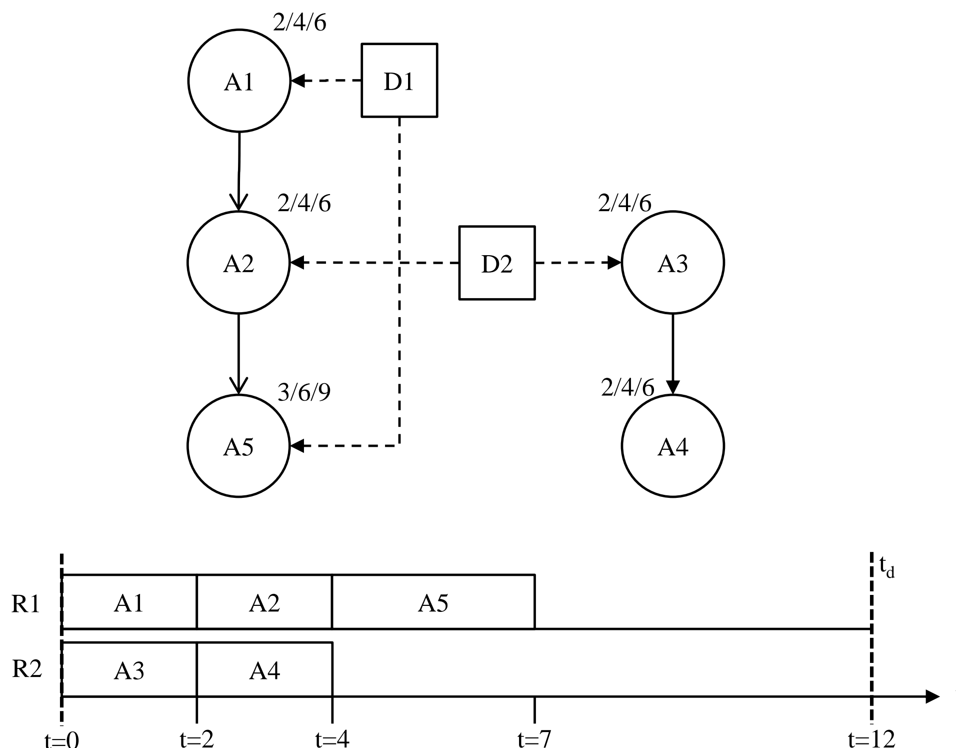


Abbildung 6.5: Beispiel mit 5 Applikationen und gültiger Startbelegung [30]

Abbildung 6.5 zeigt fünf Applikationen $A1$ bis $A5$, die in einem maximalen Periode von 12 Zeiteinheiten auf zwei zur Verfügung stehenden Verarbeitungsressourcen ausgeführt werden sollen. Die Ressourcen verfügen über drei verschiedene Betriebs- bzw. Power-Modi, die unterschiedliche Ausführungszeiten für die Applikationen implizieren. Diese Ausführungszeiten sind direkt neben den Applikationen annotiert. Während des Wechsels eines Betriebsmodus fällt zudem eine Wartezeit von einer Zeiteinheit an, die in Bezug auf die Leistungsaufnahme demjenigen Betriebsmodus zugeordnet wird,

aus dem gewechselt wird. Weiterhin existieren teilweise gegenseitige Abhängigkeiten zwischen den Applikationen, sodass diese in einer bestimmten Reihenfolge ausgeführt werden müssen. Weiterhin benötigen alle Applikationen außer $A4$ eine zusätzliche Device-Komponente, die parallel zur Ausführung aktiv sein muss. Zum Gesamtenergieverbrauch tragen somit sowohl die Verarbeitungsressourcen als auch die zusätzlichen Komponenten bei. Die statische Leistungsaufnahme der Verarbeitungsressourcen beträgt eine Leistungseinheit, die dynamische Leistungsaufnahme ist vom jeweiligen Betriebsmodus abhängig und beträgt 20 bzw. 7 bzw. 2 pro Zeiteinheit. Die Leistungsaufnahme der zusätzlichen Komponenten liegt konstant bei 20 Leistungseinheiten pro Zeiteinheit. Das Aus- und wieder Anschalten der zusätzlichen Komponenten zum Zweck der Energieminimierung benötigt jeweils eine Zeiteinheit. In Abbildung 6.6 ist die aus dem Gesichtspunkt des Energieverbrauchs optimale Konfiguration dargestellt.

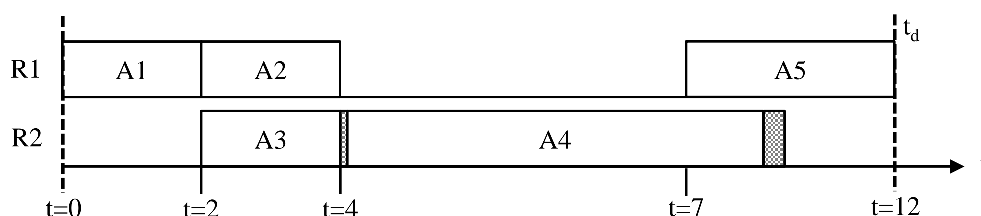


Abbildung 6.6: Optimale Ausführung der Beispiel-Applikationen [30]

Die Applikationen, die eine zusätzliche aktive Komponente während ihrer Ausführungszeit benötigen, werden in einem möglichst schnellen Betriebsmodus ausgeführt, da die verbrauchte Energie der Komponente größer ist als ein möglicher kleinerer Energieverbrauch, der durch einen Wechsel des Betriebsmodus erreicht werden könnte. Da $A1$ und $A5$ dieselbe Komponente benutzen, wird $A1$ zu Beginn des Zyklus und $A5$ am Ende des Zyklus ausgeführt, da dann die Komponente $D1$ nur einmal pro Zyklus an- und ausgeschaltet werden muss. $A2$ wird dazwischen ausgeführt, da eine Abhängigkeit zu $A1$ existiert. Gleichzeitig zu $A2$ wird $A3$ auf der zweiten Verarbeitungsressource $R2$ ausgeführt, da die Komponente $D2$ gleichzeitig benutzt und ansonsten abgeschaltet werden kann. Die Applikation $A4$ könnte prinzipiell auch auf Ressource $R1$ ausgeführt werden, wird aber auf Ressource $R2$ ausgeführt, da es dort im Gegensatz zu $R1$ möglich ist, in den langsamsten Betriebsmodus zu wechseln. Wohingegen ein System in der Startbelegung 473 Energieeinheiten benötigen würde, wird dieser Verbrauch auf 392 Einheiten in der optimalen Konfiguration minimiert.

Werden anhand dieses Beispiels die Ergebnisse analysiert, zeigt sich die Schwierigkeit der Optimierung eines teilweise ganzzahligen nichtlinearen Problems. Abbildung 6.7 zeigt das Resultat des Optimierungsalgorithmus mit 7 unterschiedlichen Startwerten. Hier werden die Ausführungszeit des Algorithmus und das Optimierungsergebnis gegenübergestellt. Wie in dem Diagramm zu sehen, lässt sich keine Korrelation zwischen diesen Werten feststellen. Die Testfälle 1 und 4 befinden sich relativ früh in einem lokalen Minimum, sodass keine weitere Optimierung mehr durchgeführt wird. Die Testfälle 6 und 7 erreichen zwar das globale Optimum, haben dabei aber eine nahezu um den Faktor 2 unterschiedliche Ausführungszeit. Auch die Gegen-

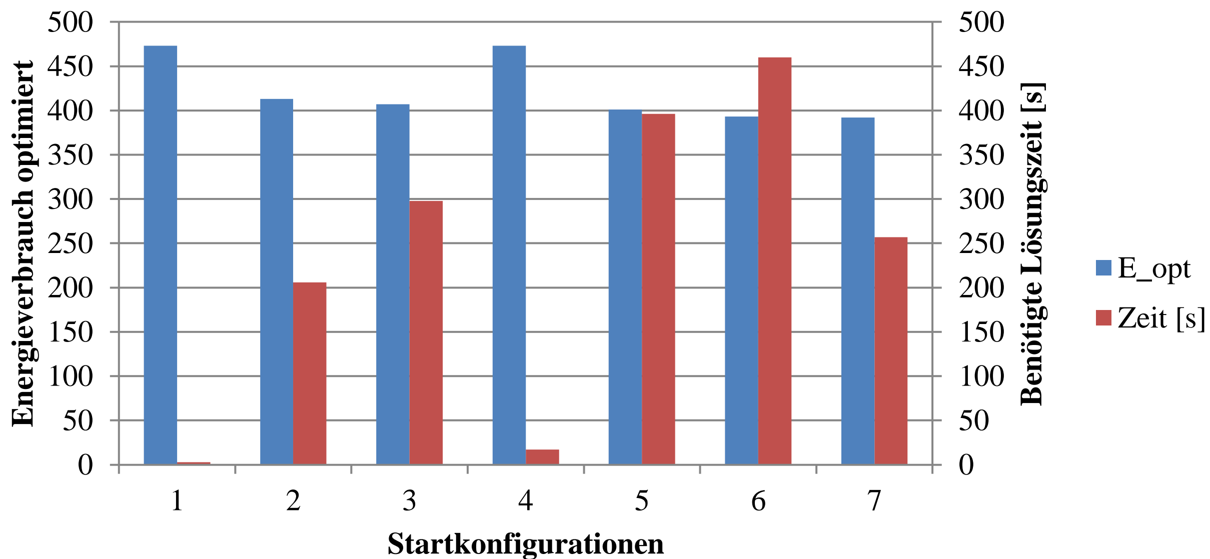


Abbildung 6.7: Abhängigkeit zwischen Ausführungszeit und Ergebnis der Optimierung

überstellung von Ausführungszeit und die in dieser Zeit durchlaufenen Zweige des Branch-and-Bound-Verfahrens, die als Diagramm in Abbildung 6.8 dargestellt ist, zeigt keine klare Abhängigkeit, was besonders bei den Testfällen 2 und 6 ersichtlich ist. Diese weisen eine ähnliche Zahl an Zweigen, jedoch eine unterschiedliche Ausführungszeit auf. Dies liegt vor allem daran, dass ein Großteil der Ausführungszeit zur Lösung des relaxierten nichtlinearen Problems benötigt wird und sich dieses iterativ in ein lokales Minimum bewegen kann. Bessere Lösungen müssen aufgrund der ganzzahligen Bedingungen der Entscheidungsvariablen oftmals verworfen werden. Insgesamt lässt sich feststellen, dass die Qualität des Optimierungsergebnisses stark von der Wahl der Startbelegung abhängt. Es hat sich gezeigt, dass möglichst viele der Entscheidungsvariablen im Startwert bereits zielgerichtet belegt werden sollten, z.B. die Abbildung auf Ressourcen sollte möglichst viel Parallelität nutzen, falls Applikationen dieselben zusätzlichen Komponenten benutzen.

Aus den eben demonstrierten Ergebnissen wird ersichtlich, dass die Qualität der Ergebnisse und teilweise damit verknüpft die Ausführungszeiten für die Berechnung der Ergebnisse stark von der Parametrisierung des Optimierungsalgorithmus abhängen. Aufgrund der nichtlinearen Formulierung und der Existenz von Entscheidungsvariablen im Lösungsvektor sollte die Optimierung somit mehrmals mit unterschiedlichen Startwerten durchgeführt werden. Eine weitere Lösung ist die Abstraktion des Optimierungsproblems und die damit verbundene Transformation des nichtlinearen Problems in ein lineares Optimierungsproblem. Dieser Ansatz wird in Abschnitt 6.1.4 verfolgt.

6.1.4 Abstraktion des Optimierungsproblems

Wie bereits in Abschnitt 2.9 erwähnt existieren effiziente und schnelle Lösungsverfahren für *lineare Optimierungsprobleme*, deren Lösungsvektor aus dem reellen Zahlenraum stammt, z.B. das Simplex-Verfahren. Es kann jedoch bei der Aufstellung eines mathematischen Optimierungsproblems generell vorkommen, dass keine Möglichkeit existiert,

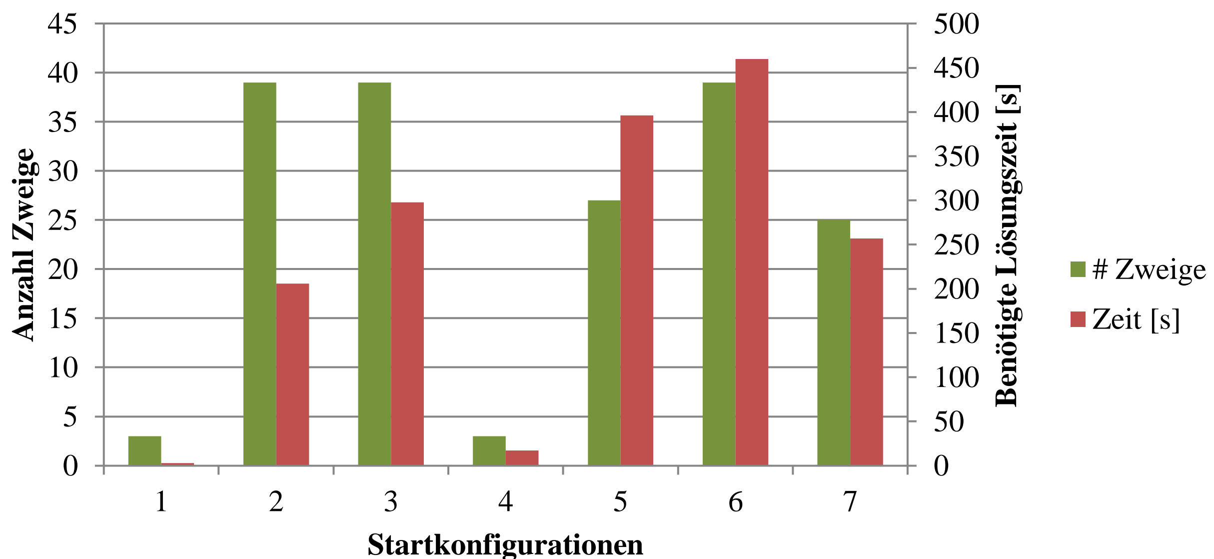


Abbildung 6.8: Abhängigkeit zwischen Ausführungszeit und Anzahl der durchlaufenen Zweige

die gewünschte Zielfunktion ausschließlich durch eine lineare Zielfunktion und durch ausschließlich lineare Gleichungen bzw. Ungleichungen als Nebenbedingungen zu formulieren. In diesem Fall muss untersucht werden, ob das Optimierungsproblem durch Einführung neuer Variablen oder durch Umformung der bereits vorhandenen Variablen umformuliert werden kann. Ist es auch damit nicht möglich, das Optimierungsproblem zu linearisieren, bleibt die Möglichkeit der Abstraktion. Dabei wird geprüft, ob und inwieweit sich das Problem vereinfachen lässt, sodass es mit einem geeigneten Verfahren schnell gelöst werden kann, das Ergebnis des abstrahierten Optimierungsproblems aber in vertretbarer Abweichung auch das Ursprungsproblem optimiert.

Wie das vorliegende Problem zur Optimierung der Energieeffizienz in eingebetteten Hardware/Software-Systemen abstrahiert und damit als lineares Optimierungsproblem formuliert werden kann, soll im nachfolgenden Abschnitt 6.1.4.1 diskutiert werden.

6.1.4.1 Formulierung als lineares Optimierungsproblem

Die lineare Optimierung bildet ein Spezialfall der allgemeinen mathematischen Optimierung, wie sie bereits in Abschnitt 6.1.1 eingehend erläutert wurde. Bei einer linearen Optimierung muss die Zielfunktion linear sein und die Nebenbedingungen durch ein System linearer Gleichungen bzw. Ungleichungen formulierbar sein. Eine entscheidende Eigenschaft linearer Optimierungsprobleme ist es, dass jedes lokale Optimum automatisch auch einem globalen Optimum entspricht. Dies liegt daran, dass der zulässige Lösungsbereich konvex ist und somit weder Sattelpunkte noch lokale Optima existieren können.

Generell ist es notwendig, sowohl nichtlineare als auch quadratische Formulierungen von Zielfunktion und Nebenbedingungen zu vermeiden. Speziell für quadratische Formulierungen lässt sich ein allgemeiner Ansatz entwickeln, um ein Produkt zwischen

einer binären Lösungsvariablen b und einer kontinuierlichen Lösungsvariablen c durch Einführung einer neuen Produktvariablen p in eine Menge von linearen Nebenbedingungen umzuformen.

Sei $\overline{c}$ die obere Schranke der kontinuierlichen Variable c und $\underline{c}$ dementsprechend die untere Schranke. Dann gelten für die Variable p, die das Produkt von b und c darstellen soll, folgende lineare Nebenbedingungen:

$$\underline{c} \cdot b \leq p \leq \overline{c} \cdot b \tag{6.20}$$

$$c - \underline{c} \cdot (1 - b) \leq p \leq c \tag{6.21}$$

$$\underline{c} \leq c \leq \overline{c} \tag{6.22}$$

$$0 \leq b \leq 1 \tag{6.23}$$

Mithilfe dieser Umformulierung lassen sich sowohl die Zielfunktion der Gleichungen 6.1 bis 6.11 als auch die Nebenbedingungen 6.12 bis 6.19 teilweise in linearisierter Form darstellen. Grundsätzlich werden dabei jeweils die unteren und oberen Schranken der binären Lösungsvariablen mit den Werten 0 und 1, und die kontinuierlichen Lösungsvariablen mit den Werten 0 und t^H belegt. Des Weiteren gelten die Bedingungen 6.13 der eindeutigen Zuordnung und 6.14 zur Abbildung gegenseitiger Abhängigkeiten zwischen Applikationen.

Energie aufgrund dynamischer Leistungsaufnahme Da der Energieverbrauch, der aufgrund dynamischer Leistungsaufnahme existiert, bereits durch Gleichung 6.2 linear formuliert ist, kann diese unverändert beibehalten werden.

Energie aufgrund statischer Leistungsaufnahme Für die Berechnung des Energieverbrauchs aufgrund statischer Leistungsaufnahme werden neue Lösungsvariablen für eine Lücke g eingeführt, die sich immer auf die davor liegende Applikation bezieht. Zwischen einer Applikation i und einer Applikation j befindet sich also die Lücke g_i. Wenn g_i^{end} das Ende und g_i^{dur} die Länge der Lücke nach dem Ende e_i von Applikation i definiert, lässt sich folgende Nebenbedingung ableiten:

$$g_i^{end} = e_i + g_i^{dur} \tag{6.24}$$

Weiterhin muss durch eine binäre Lösungsvariable $sr_{i,j}$ angezeigt werden, ob zwei Applikationen i und j auf einer geteilten Ressource $r \in R$ ausgeführt werden und durch zwei binäre Lösungsvariablen $before_{i,j}$ und $after_{i,j}$, ob i vor j oder i nach j ausgeführt wird. Diese Variablen lassen sich durch folgende Bedingungen einschränken:

$$x_{i,r} + x_{j,r} - sr_{i,j} \leq 1 \quad \forall i, j \in A, r \in R \tag{6.25}$$

$$before_{i,j} + after_{i,j} \geq sr_{i,j} \quad \forall i, j \in A, r \in R \tag{6.26}$$

Durch die Umformulierung von Produkten von Lösungsvariablen mithilfe der linearen Nebenbedingungen 6.20 bis 6.23 wird eine zusätzliche binäre Lösungsvariable $o_{i,r}$ eingeführt, die anzeigt, ob während einer Lücke $g_{i,r}$ die entsprechende Ressource r abgeschaltet, also in den Zustand z_{off} gesetzt wird und durch die kontinuierliche

Variable $o_{i,r}^{dur}$ wie lange sich die Ressource innerhalb der Lücke im Zustand z_{on} befindet. Dabei wird durch die folgenden Nebenbedingungen garantiert, dass die ausführende Ressource abgeschaltet wird, wenn die Lücke größer ist als die Break-Even-Zeit t_r^{BE} der Ressource ist:

$$o_{i,r} \geq g_{i,r} - t^H \cdot (1 - (1 - o_{i,r})) \quad \forall i \in A, r \in R \tag{6.27}$$

$$o_{i,r} \leq t^H \cdot (1 - o_{i,r}) \quad \forall i \in A, r \in R \tag{6.28}$$

Dadurch lässt sich die Energie aufgrund statischer Leistungsaufnahme durch Gleichung 6.29 berechnen.

$$E^{stat,LP} = \left(\sum_r \sum_m \sum_i x_{i,m,r} \cdot t_{i,m,r} + \sum_r \sum_i o_{i,r}^{dur} + t_r^{BE} \cdot o_{i,r} \right) \cdot P_r^{stat} \tag{6.29}$$

Energie für Moduswechsel Der genaue Wert der Energie, die durch zusätzliche Moduswechsel verbraucht wird, kann in einer linearen Formulierung nicht exakt bestimmt werden, da er immer sowohl vom vorhergehenden Zustand als auch vom jetzigen Modus abhängt, was einer quadratischen Abhängigkeit von zwei binären Lösungsvariablen entspricht. Vielmehr wird die Summe der Zeit, die für alle Moduswechsel benötigt wird, in die Kostenfunktion aufgenommen. Dies führt zwar nicht unweigerlich zum absoluten Minimum, da aber die Energie als Produkt der elektrischen Leistung und der Zeit berechnet wird, wird bei einer Minimierung der Zeit für Moduswechsel auch die dafür benötigte Energie minimiert.

Device-Energie Durch die Nebenbedingung, dass eine Device-Komponente im Zustand z_{on}, also aktiv sein muss, wenn die zugehörige Applikation ausgeführt wird, wird die Zeit, in der die Device-Komponente aktiv sein muss, an die Zeit der Applikation gekoppelt. Da diese Zeit als Faktor in die Berechnung der Energie enthalten ist, führt eine Minimierung der Ausführungszeit auch zwangsläufig zu einer Minimierung der Energie. Die parallele Nutzung von Device-Komponenten durch mehrere Applikationen und die daraus folgende anteilige Anrechnung der Energie kann durch die Abstraktion allerdings nicht mehr gewährleistet werden.

Zeitliche Überlagerungen von Ausführungen und Moduswechsel Mithilfe der Variablen sr, $before$ und $after$ können die zeitlichen Überlagerungen durch Nebenbedingungen 6.30 und 6.31 für den Fall definiert werden, dass Applikation i vor j bzw. j vor i auf derselben und somit einer geteilten Verarbeitungsressource ausgeführt werden, wobei m dem gewählte Power-Mode von Applikation i entspricht und n dem Power-Mode von Applikation j.

$$s_j - e_i - t_{i,j}^{sw} + t^H \cdot \left(1 - before_{i,j}\right) \geq 0 \quad \forall i, j \in A, r \in R \tag{6.30}$$

$$s_i - e_j - t_{i,j}^{sw} + t^H \cdot \left(1 - after_{i,j}\right) \geq 0 \quad \forall i, j \in A, r \in R \tag{6.31}$$

Die Variable $t_{i,j}^{sw}$, die die benötigte Zeit für einen möglichen Moduswechsel zwischen den Applikationen i und j definiert, wird durch Bedingung 6.32 festgelegt.

$$x_{i,m,r} + x_{j,n,r} - \frac{t_{i,j}^{sw}}{t_{m,n,r}^{sw}} \leq 1 \quad \forall i, j \in A, r \in R, m, n \in PM, t_{m,n,r}^{sw} > 0 \tag{6.32}$$

In Kombination mit der Verwendung von $t_{i,j}^{sw}$ in der Kostenfunktion als zusätzlichem Energieaufwand für den Wechsel der Power-Modi und der damit verbundenen Minimierung dieser Lösungsvariablen wird $t_{i,j}^{sw}$ gleich dem Wert von $t_{m,n,r}^{sw}$ gesetzt, wenn Applikation i in Power-Modus m und Applikation j in Power-Modus n auf der geteilten Ressource r ausgeführt werden.

Kann Ressource r abgeschaltet werden, müssen auch noch die Ab- und Anschaltzeiten in die Nebenbedingungen aufgenommen werden. Die Variablen g_i^{dur} in Nebenbedingungen 6.33 und 6.34 definieren die Lücke zwischen Applikationen und garantieren dadurch, dass die Lücke mindestens so groß ist, dass die Ressource nach einem vorherigen Abschaltvorgang vor dem Ausführen der nächsten Applikation wieder in den Zustand z_{on} versetzt werden kann.

$$s_j - e_i - g_i^{dur} + t^H \cdot \left(1 - before_{i,j}\right) \geq 0 \quad \forall i, j \in A, r \in R \tag{6.33}$$

$$s_i - e_j - g_j^{dur} + t^H \cdot \left(1 - after_{i,j}\right) \geq 0 \quad \forall i, j \in A, r \in R \tag{6.34}$$

6.1.4.2 Fazit der Abstraktion des Optimierungsproblems

Durch die Einführung zusätzlicher Variablen lässt sich die Energie aufgrund statischer und dynamischer Leistungsaufnahme in die Zielfunktion eines linearen Optimierungsproblems integrieren, sodass alle Optima der Zielfunktion global gelten. Ebenfalls werden die zeitlichen Überlagerungen von Ausführungen und Wechsel der Betriebsmodi durch die Formulierung der Nebenbedingungen garantiert. Bezüglich der Integration der Energie für die Wechsel der Betriebszustände und der Energie der Devices führt die Abstraktion der Problemformulierung dazu, dass diese zwar optimiert bzw. minimiert werden, da sie in die Optimierungsfunktion eingehen, der Wert der daraus resultierenden Zielfunktion jedoch nicht dem tatsächlich repräsentierten Wert entspricht. Dadurch kann nicht mehr garantiert werden, dass ein globales Optimum erreicht wird. Der tatsächliche Wert der Zielfunktion kann nach der Ausführung des Lösungsalgorithmus unter Verwendung des resultierenden Lösungsvektors berechnet werden. Die Verwendung des Algorithmus zur Lösung des abstrahierten und dadurch linearisierten Optimierungsproblems während der Entwurfsphase wird in Abschnitt 7.3.1 anhand einer Anwendung, die aus dem Bereich der Fahrerassistenzfunktionen stammt, demonstriert.

6.2 Optimierung während der Laufzeit

Das Potential von Verfahren zur Optimierung der Energieeffizienz erhöht sich, je größer die Differenz zwischen der maximal möglichen Ausführungsleistung und

der gerade benötigten Leistung hinsichtlich der zu diesem Zeitpunkt an das System gestellten Anforderungen ist. Im Gegensatz zu Methoden, die auf eine Maximierung der Leistungsfähigkeit eines Systems abzielen, werden existierende Freiheitsgrade unter diesen Umständen dazu benutzt, den Energieverbrauch zu minimieren. Gleichzeitig sollen aber geltende Anforderungen an das System weiterhin bestehen bleiben und eingehalten werden.

Offensichtlich reicht es bei der Minimierung des Energieverbrauchs nicht aus, das Worst-Case-Verhalten einzelner Anwendungen/Applikationen und Tasks zu analysieren, da diese keine variable und somit gerade benötigte Leistungsfähigkeit abbilden kann. Vielmehr gehen darauf basierende Ansätze von einem vor der Ausführung abgeschätzten Verhalten aus, das sich konstant über die gesamte Ausführungszeit verteilt. In der Realität zeichnen sich eingebettete Hardware/Software-Systeme dadurch aus, dass sich die an das System gestellten Anforderungen während der Ausführungszeit ständig ändern. Da besonders bei harten Echtzeitbedingungen die gestellten Anforderungen auch im Worst-Case-Fall zwingend erfüllt werden müssen, unterliegen diese Systeme großen Schwankungen bezüglich der Auslastung bzw. des benötigten Durchsatzes (vgl. Abschnitt 4.1.3).

Wie vom Autor bereits in [139] vorgeschlagen, wird in diesem Abschnitt ein Ansatz vorgestellt, der es erlaubt, die von den derzeitig an das System gestellten Anforderungen abgeleiteten Leistungsunterschiede für eine Optimierung der Energieeffizienz nutzbar zu machen. Dazu werden die in Abschnitt 4.1.4 erläuterten Techniken DVFS und DPM zur Minimierung der Leistungsaufnahme in digitalen Hardware/Software-Systemen eingesetzt und entsprechend einer aus der Optimierung abgeleiteten Konfiguration parametrisiert. Ausgangspunkt für die Optimierung ist ein mathematisches Modell, das es erlaubt, die gegenseitigen Abhängigkeiten der Tasks in einen zeitlichen Verlauf zu überführen. Durch Abstraktion dieses zeitlichen Verlaufs kann abhängig von den jeweils aktuell geltenden Leistungsanforderungen eine Strategie zur Optimierung der Energieeffizienz abgeleitet werden, die während der eigentlichen Laufzeit eingesetzt wird.

Durch die oftmals nicht eindeutige Definition von Applikationen und Tasks kann diese Ebene der Optimierung auch als *Intra-Task-Optimierung* bezeichnet werden. Dieser Begriff soll ausdrücken, dass die Mechanismen zur Minimierung der Leistungsaufnahme nicht nur während der Abfolge mehrerer Applikationen sondern auch innerhalb einer Applikation angewandt werden können. Die Minimierung des Energieverbrauchs basiert dabei auf der Kenntnis der aktuellen Betriebsszenarien des eingebetteten Hardware/Software-Systems.

6.2.1 Modellierung gegenseitiger Abhängigkeiten

Wie in Abschnitt 6.1 wird auch hier von einer Unterteilung der Applikationen in einzelne Tasks ausgegangen. Diese bilden somit eine eigene Abstraktionsebene, welche sich durch einen teilweise hohen Grad an gegenseitigen Abhängigkeiten auszeichnet. Das liegt hauptsächlich daran, dass Tasks in ihrer Gesamtheit zu der Erbringung der Funktionalität einer Applikation beitragen. Das dementsprechend zugrunde liegende

Task-Modell, das auch schon in den Definitionen in Abschnitt 4.2.1 enthalten ist, ist so definiert, dass eine Quelle und eine Senke existiert. Weiterhin warten Tasks jeweils auf die Daten ihrer Vorgänger und geben Daten nach der Bearbeitung an ihre Nachfolger weiter. Es soll an dieser Stelle darauf hingewiesen werden, dass das nachfolgende Verfahren unabhängig von der Abstraktionsebene ist, in der Tasks modelliert sind. Tasks können somit also z.B. auch Funktionen oder Basisblöcke in programmiertechnischem Sinne darstellen. Ohne Beschränkung der Allgemeinheit wird aber davon ausgegangen, dass es sich um logisch abgetrennte Unterabschnitte bzw. „Bausteine" einer Funktionalität oder Applikation handelt, die ein gewisses Ergebnis bzw. Zwischenergebnis aus einer bestimmten Eingabe heraus produzieren, welches wiederum von einer nächsten Einheit erwartet wird. Dieses Verhalten setzt sich sukzessive bis zum Erreichen der endgültigen Funktionalität einer Applikation fort.

Die gegenseitigen Abhängigkeiten zwischen Tasks entstehen folglich dadurch, dass ein Task auf das Resultat eines anderen Tasks warten muss, bis er mit der eigenen Ausführung beginnen kann. Die entstehende logische Reihenfolge bei der Abarbeitung der Tasks muss bei der Ablaufplanung berücksichtigt werden, da sonst die Semantik der Ausführung nicht eingehalten werden könnte. Deswegen wird für die Verhaltensmodellierung innerhalb einer Applikation das Modell des *Synchronen Datenflusses (SDF)* verwendet. Dessen grundlegende Details wurden bereits in Abschnitt 4.1.2 beschrieben, weswegen an dieser Stelle verstärkt die Anwendung des Modells auf die vorliegende Problemstellung behandelt werden soll. So stellen synchrone Datenflussmodelle für das Task-Modell, auf dem das in diesem Abschnitt beschriebenen Verfahren beruht, eine gute Basis dar, um das funktionale und nicht-funktionale Verhalten und die gegenseitigen Abhängigkeiten von Tasks modellieren zu können. Da dieses Verhalten jedoch auch abhängig ist von der zugrunde liegenden Hardware-Plattform und deren Betriebszuständen, sind dahingehende Erweiterungen zwingend notwendig.

Das in Abbildung 6.9 gezeigte Beispiel-SDF-Modell bzw. der zugehörige SDF-Graph ist aufgrund der gegebenen einfachen Übersichtlichkeit aus [37] entnommen und entsprechend den in dieser Arbeit thematisierten Aspekten erweitert worden, was ausführlich in Abschnitt 6.2.2 beschrieben ist.

In SDF-Modellen werden Tasks durch *Aktoren* ausgeführt, die vorkommenden Daten sind durch *Token* repräsentiert. Wie durch die Semantik von SDF-Modellen vorgegeben, können Aktoren nur dann feuern, wenn eine genügend große Menge an Token an allen Eingängen anliegt. Beim Ausführen des Aktors wird diese Menge an Eingangs-Token verbraucht, wobei diese Anzahl auch der *Konsumrate* entspricht. Als Resultat der Ausführung wird eine Menge an Ausgangs-Token an jedem der Ausgänge ausgegeben, was auch als *Produktionsrate* bezeichnet wird. Für die Modellierung von Softwaresystemen wird den Aktoren eine Ausführungszeit zugeordnet, die meist der WCET der dadurch modellierten Tasks entspricht.

Ein bestimmter Aktor kann also erst ausgeführt werden, wenn alle seine Eingänge mit der zum Ausführen des Aktors notwendigen Anzahl an Token belegt sind. Der frühestmögliche Ausführungszeitpunkt dieses Aktor ist also direkt abhängig von allen Vorgängern des entsprechenden Aktors im Datenflussmodell. Da ein Aktor aber wiederum Vorgänger weiterer Aktoren sein kann, hängen jene vom Ausführungszeitpunkt

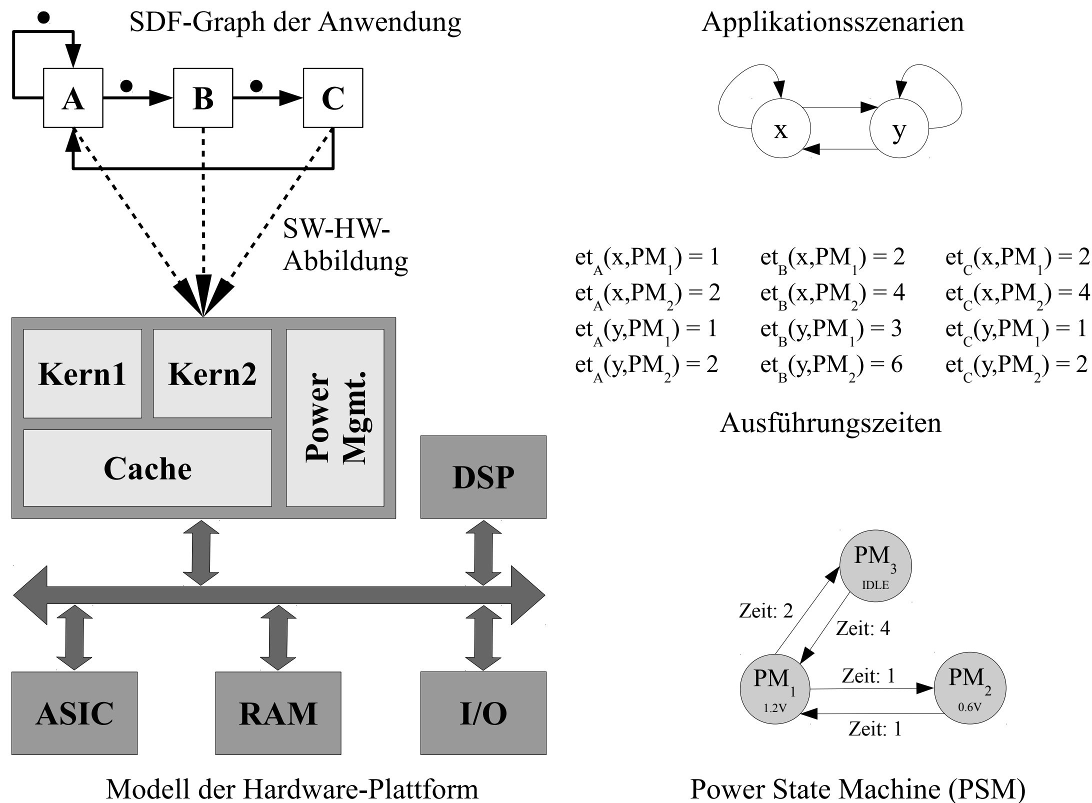


Abbildung 6.9: Erweiterung des SDF-Modells zur Berücksichtigung der Hardware-Plattform und der Ausführungszeiten abhängig vom Betriebsmodus

und der Ausführungsdauer dieses Aktors und aller anderen Vorgänger-Aktoren ab. Es gilt also für die Vorgänger- bzw. Nachfolgebeziehung eines beliebigen Aktors a und aller seiner n Vorgängeraktoren v_i für $(i = 1 \dots n)$ folgende grundsätzliche Abhängigkeit:

$$\text{Endzeitpunkt}_a \geq \max_i(\text{Endzeitpunkt}_{v_i}) + \text{Ausführungszeit}_a \tag{6.35}$$

Aufgrund dieser Beziehungen der Ausführungszeitpunkte kann das zeitliche Systemverhalten in ein dazu passendes mathematisches Modell zur Repräsentation der gegenseitigen Abhängigkeiten überführt werden. Dieses wird in Abschnitt 6.2.3 motiviert und im Detail beschrieben.

6.2.2 Multi-Domänen-Szenarien zur Abbildung dynamischen Verhaltens

In SDF-Modellen wird das Verhalten des modellierten Systems durch die Eigenschaften der Aktoren dargestellt. In reinen SDF-Modellen sind diese Aktor-Eigenschaften jedoch statisch modelliert, sodass eine dynamische Verhaltensmodellierung an sich nicht möglich ist. Im Übrigen sind die Eigenschaften der Kanten, die die Produzenten- und Konsumentenraten der jeweilig verbundenen Aktoren spezifizieren, ebenso statisch durch das SDF-Modell vorgegeben.

Soll also durch die gleiche Modellierungsmethodik auch dynamisches Verhalten modelliert werden können, muss das Modell des SDF-Graphen erweitert werden. Wie in den nachfolgenden Abschnitten dargestellt, können in einem Szenarien-basierten Ansatz mehrfache Konzepte umgesetzt werden, die jeweils einen signifikanten Einfluss auf das Hardware-/Software-System haben. Durch diese Konzepte kann somit auch dynamisches Verhalten modelliert werden – und zwar sowohl Software-seitige als auch Hardware-seitige Dynamik bei der Ausführung einer bestimmten Applikation.

6.2.2.1 Software-basierte Eigenschaften

Auf Seiten der Software-Sicht hängen die dynamischen Eigenschaften der Aktoren eines SDF-Graphen vom Zustand ab, in dem sich die dargestellte Applikation gerade befindet. Die Zustände repräsentieren Variationen in der Applikationsausführung, die meist abhängig vom vorherigen Zustand eines Systems und den aktuell geltenden Bedingungen sind, z.B. abhängig von der Eingabe. Alle möglichen Zustände der Applikation werden in einem endlichen Automaten (engl. *Finite State Machine (FSM)*) modelliert [36]. Ein solcher endlicher Automat ist neben dem eigentlichen SDF-Graphen in Abbildung 6.9 dargestellt. Ein möglicher Zustand wird auch als *Applikationsszenario* bezeichnet und ist wie folgt definiert.

Definition 6.2 (Applikationsszenario)
Ein Applikationsszenario as_i^a *ist der i-te Laufzeitmodus eines Aktors a, der dessen nicht-funktionalen Eigenschaften, z.B. Zeitverhalten, in Abhängigkeit der Eingangsdaten von a beeinflusst.*

Die Kombination aus SDF-Graph und endlichem Automat, der die Applikationsszenarien und deren Übergänge definiert, wird im Folgenden auch *ASADF-Graph* (engl. Application Scenario-aware Dataflow Graph) bezeichnet.

Definition 6.3 (ASADF-Graph)
Ein ASADF-Graph ist ein 4-Tupel $(V_{SDF}, E_{SDF}, \Phi, \Psi)$ *eines SDF-Graphen* (V_{SDF}, E_{SDF}) *mit Knoten* V_{SDF} *und Kanten* E_{SDF}*. Der ASADF beinhaltet sowohl ein SDF-Modell zur Repräsentation der Applikation inklusive der gegenseitigen Abhängigkeiten zwischen Aktoren als auch einen endlichen Automaten, der die verschiedenen Applikationsszenarien beschreibt. Für jedes Applikationsszenario as existiert eine Funktion* $\Phi: V \rightarrow \mathbb{R}^n$*, die die Eigenschaften der Aktoren (z.B. Zeitverhalten, Leistungsaufnahme) in dem jeweiligen Zustand des endlichen Automaten bestimmt. Die Funktion* Ψ *definiert die Datenfluss-Eigenschaften, z.B. Produktions- und Konsumraten, der Kanten* E_{SDF} *in den spezifizierten Szenarien.*

Generelle weiterführende Definitionen von Szenarien-basierten SDF-Graphen (SADF) können den Arbeiten von Theelen et al. [122] entnommen werden. Zusätzlich sind dort grundsätzliche Beweise für SADF-Graphen, wie z.B. für die Endlichkeit und Deadlock-Freiheit von Markov-basierten SADF-Graphen, enthalten.

Beispielweise könnte die Kodierberechnung einer Multimedia-Applikation in einem Applikationsszenario as_1 10 ms benötigen, wohingegen dieselbe Berechnung in einem anderen Applikationsszenario as_2 nur 4 ms dauert. Ob sich das System in as_1 oder as_2 befindet, kann in diesem Fall davon abhängen, welcher Typ von Frame gerade kodiert

wird. Im MPEG-2-Standard existieren z.B. drei verschiedene Frame-Typen: Schlüssel-Frames (I-Frames), Vorhersage-Frames (P-Frames) und bidirektionale Vorhersage-Frames (B-Frames).

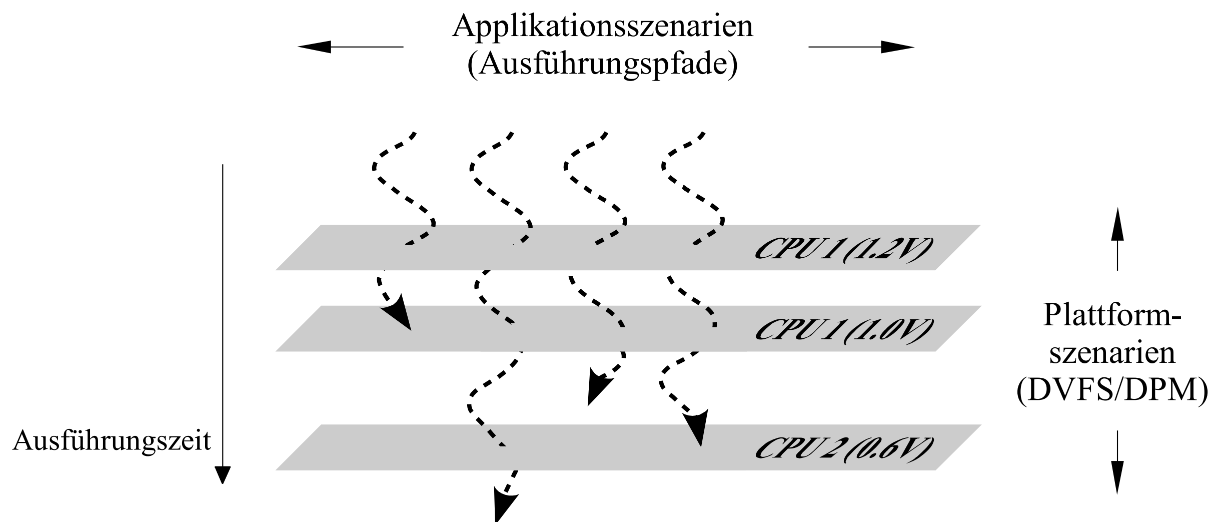


Abbildung 6.10: Orthogonales Konzept der Multi-Domänen-Szenarien

Applikationsszenarien werden in dieser Arbeit dazu genutzt, endlich viele *Ausführungspfade* bzw. *Timing-Pfade* innerhalb einer Applikation zu modellieren. Als Ausführungspfade werden voneinander verschiedene Programmausführungen, also verschiedene Daten- und Kontrollflussabläufe derselben Applikation bezeichnet. Aufgrund der unterschiedlichen Abfolge der Daten- und Kontrollflüsse besitzen verschiedene Ausführungspfade in der Regel auch unterschiedliche nicht-funktionale Eigenschaften. Abbildung 6.10 stellt schematisch die Applikationsszenarien einer gegebenen Applikation in horizontaler Richtung dar.

Die in einem Hardware/Software-System vorkommenden Applikationsszenarien können durch verschiedene Ansätze ermittelt werden:

- Spezifizierung anhand Vorgaben oder bekannter Ausführungszeiten
- Datenabhängige Analyse der Worst-Case Execution Time
- Simulation eines repräsentativen Modells

Wie im letzten Punkt erwähnt, können die vorkommenden Applikationsszenarien anhand einer durchgeführten Simulation ermittelt werden. Hierzu werden die bei der Simulation auftretenden Ausführungspfade analysiert und anhand eines vorgegebenen Kriteriums gruppiert. Die Möglichkeit zur simulativen Ableitung der Applikationsszenarien unter Verwendung virtueller Prototypen wurde bereits in Abschnitt 5.4.3 diskutiert.

6.2.2.2 Hardware-basierte Eigenschaften

Die Ausführungszeit eines Aktors hängt aber grundsätzlich nicht nur von dem jeweils aktuellen Applikationsszenario, sondern auch maßgeblich von den dynamischen

Eigenschaften der ausführenden Verarbeitungsressource ab. Diese Betrachtung entspricht einer Hardware-orientierten Sichtweise. Die dynamischen Eigenschaften der Hardware-Plattform werden maßgeblich von dem aktuellen Betriebsmodus bzw. Power-Modus beeinflusst, in dem sich die Ressource zur Zeit der Applikationsausführung befindet. Für die Menge AS an möglichen Applikationsszenarien und die Menge PM an verfügbaren Betriebsmodi wird die Ausführungszeit et_a eines Aktors a auf einer Verarbeitungsressource $r \in R$ durch die Relation $AS \times PM^r \rightarrow \mathbb{R}$ bestimmt.

Die eben beschriebenen Szenarien, die von den auf den Verarbeitungsressourcen zur Verfügung stehenden Betriebsmodi abhängen, werden im weiteren Verlauf als *Plattformszenarien* bezeichnet. Ein Plattformszenario ist dementsprechend wie folgt definiert.

Definition 6.4 (Plattformszenario)
Ein Plattformszenario ps_j ist der j-te Betriebsmodus einer Plattform- bzw. Verarbeitungsressource, welcher die nicht-funktionalen Eigenschaften derjenigen Aktoren beeinflusst, die diese Verarbeitungsressource für die Erbringung ihrer Funktionalität verwenden.

Plattformszenarien beschreiben demnach die verschiedenen Zustände der zugrunde liegenden Hardware-Plattform, wie z.B. Kerne, Caches, oder ein Speichersystem. Aufgrund der Darstellung von diskreten Zuständen werden für die Modellierung von Plattformszenarien analog zu den Applikationsszenarien endliche Automaten benutzt. Ohne Beschränkung der Allgemeinheit sollen in dieser Arbeit endliche Automaten zur Modellierung von Power-Zuständen einer gegebenen Hardware-Plattform verwendet werden. In dieser so genannte *Power State Machine (PSM)* werden nicht nur die möglichen einzelnen Power-Zustände, sondern auch die für die Zustandswechsel benötigten Umschaltzeiten und der daraus resultierende Mehraufwand bei der elektrischen Leistung modelliert. Eine PSM wird also wie folgt definiert.

Definition 6.5 (PSM)
Eine Power State Machine (PSM) einer Ressource $r \in R$ eines Hardware/Software-Systems ist ein 7-Tupel $(V_{PSM}, E_{PSM}, \Theta_{PSM}, \vartheta_{PSM}, \delta_{PSM}, \rho_{PSM}, v_0)$, in dem V_{PSM} die Menge an unterschiedlichen Power-Zuständen von r und $E_{PSM} \subseteq V_{PSM} \times V_{PSM}$ die Menge an Zustandswechsel definiert. Die Funktion $\delta_{PSM}(v_i, v_j) : V_{PSM} \times V_{PSM} \rightarrow \mathbb{R}$ definiert die für einen Zustandswechsel zwischen den Zuständen $v_i, v_j \in V_{PSM}$ benötigte Umschaltzeit und die Funktion $\rho_{PSM}(v_i, v_j) : V_{PSM} \times V_{PSM} \rightarrow \mathbb{R}$ die für den Zustandswechsel benötigte elektrische Leistung. Eine Transitionsfunktion $\vartheta_{PSM} : V_{PSM} \times \Theta_{PSM} \rightarrow V_{PSM}$ schaltet den endlichen Automat in einen neuen Zustand und weist den Mehraufwand für den Zustandswechsel dem gerade ausgeführten Aktor zu. $v_0 \in V_{PSM}$ beschreibt den Ausgangszustand des modellierten Systems.

Aus der kombinierten Sichtweise von Hardware und Software und der Integration in einem gemeinsamen Ansatz basierend auf diesen Multi-Domänen-Szenarien soll in einem analytischen Verfahren ein Entscheidungsraum zur Entwurfszeit gebildet werden, der die Möglichkeiten zur Steigerung der Energieeffizienz während der Laufzeit exploriert. Hauptsächliches Kriterium ist hierbei die strikte Einhaltung aller an das modellierte Hardware/Software-System gestellten Anforderungen. Ohne Beschränkung der Allgemeinheit wird in den nachfolgenden Abschnitten von einer Orientierung der spezifizierten Anforderungen am logischen Durchsatz ausgegangen, jedoch wären

durch triviale Umformungen auch andere Performanzanforderungen denkbar, z.B. eine maximal erlaubte Periode. Zur Definition des logischen Durchsatzes sei an dieser Stelle auf den Abschnitt 4.1.3 verwiesen.

6.2.3 Mathematisches Modell zur Erstellung des Zustandsraums

Die Optimierung der Energieeffizienz auf Taskebene wird maßgeblich von der Notwendigkeit bestimmt, dass die dynamischen Eigenschaften des zugrunde liegenden Hardware/Software-Systems zur Laufzeit ausgewertet und die daraus resultierenden Informationen zur Ableitung einer Energieverbrauchs-optimierenden Strategie angewandt werden müssen. Aus dieser Tatsache leitet sich die wichtige Voraussetzung ab, dass das Laufzeitverhalten so in einem Modell repräsentiert werden muss, dass es schnell genug ausgewertet werden kann.

Die Kategorisierung durch den zuvor beschriebenen Ansatz basierend auf Multi-Domänen-Szenarien wird überführt in ein mathematisches Modell, das eine abstrahierte Sichtweise auf das zeitliche Verhalten des Hardware/Software-Systems bietet. Weiterhin muss das mathematische Modell die gegenseitigen Abhängigkeiten bei der Ausführung von Aktoren, wie sie abstrahiert in Gleichung 6.35 dargestellt sind, abbilden.

Aufgrund der eben genannten Anforderungen wird die SDF-Semantik und damit der zeitliche Verlauf der Ausführungen der Aktoren durch eine (max,+)-Algebra ausgedrückt. Grundsätzlich wird hierfür die Standard-Semantik einer (max,+)-Algebra verwendet, wie sie bereits im Abschnitt 2.4.4 des Grundlagenkapitels eingeführt wurde. Allerdings muss diese Definition, die auf der Formulierung aus [37] basiert, im Hinblick auf die späteren Berechnungsschritte und die damit verbundene Verwendung von Vektoren erweitert werden. Die semantischen Erweiterungen sollen im Folgenden beschrieben werden.

Definition 6.6 (Erweiterungen (max,+)-Algebra)
Seien $b = (b_1, ..., b_n)^T$ und $c = (c_1, ..., c_n)^T$ Vektoren mit $b_i, c_i \in \mathbb{R}^n \cup \{-\infty\}$, $d \in \mathbb{R}$ ein Skalar und $M \in \mathbb{R}^{n \times n}$ eine Matrix mit Spaltenvektoren $m_j \in \mathbb{R}^n$.

- $\max(-\infty, d) = \max(d, -\infty) = d$ *bezeichnet den Maximum-Operator mit $-\infty$ als dem neutralen Element.*
- *$b + d$ bedeutet $(b_1 + d, ..., b_n + d)^T$, sodass das Skalar d zu jedem Vektoreintrag in b addiert wird.*
- *$b + c$ resultiert in einem Vektor $(b_1 + c_1, ..., b_n + c_n)^T$.*
- *$||b|| = \max_i b_i$ beschreibt den maximalen Eintrag in Vektor b.*
- *$M \otimes b$ wird definiert als $\max_j(m_j + b)$.*
- *$b^{mod\ d} := (b_1 \bmod d, ..., b_n \bmod d)^T$ definiert die Modulo-Operation auf jedem Eintrag in Vektor b.*

Diese Erweiterungen sind konsistent zur Standarddefinition einer (max,+)-Algebra, wie sie in den Grundlagen in Abschnitt 2.4.4 beschrieben wurde, sodass die dort ebenfalls definierten Eigenschaften entsprechend weiterhin gelten.

Als *Iteration* wird in der SDF-Semantik die Zeitspanne bezeichnet, die benötigt wird, damit jede Kante im SDF-Modell genau wieder diejenige Menge an Token besitzt, die sie auch im initialen Zustand hatte. Die Produktionszeiten aller im Modell vorhandenen Token sind in einem *Zeitstempel-Vektor* v enthalten, der dementsprechend genauso viele Einträge hat wie initial Token im Modell vorhanden sind. Im übertragenen Sinne heißt das, dass ein Zeitstempel-Vektor v^{i-1} nach dem Ablauf einer Iteration i zu einem Vektor v^i wird, der aber nun die Token-Produktionszeiten der i-ten Iteration enthält. Im Allgemeinen bedeutet das, dass das zeitliche Verhalten der nächsten Iteration durch das zeitliche Verhalten der aktuellen Iteration bestimmt wird.

Weiterhin existiert eine *Ausführungsmatrix* $G_{as,ps}$ für jede Kombination möglicher Applikationsszenarien $as \in AS$ und Plattformszenarien $ps \in PS$. Die Matrix $G_{as,ps}$ beinhaltet sowohl Informationen über den Repetitionsvektor des zugehörigen SDF-Modells, als auch die Ausführungszeiten der im SDF-Modell vorhandenen Aktoren und der Ablaufplanung (engl.: Scheduling), welche die Ausführungsreihenfolge von Aktoren vorgibt, falls diese nicht schon durch die im SDF-Modell inhärenten gegenseitigen Abhängigkeiten bereits fest vorgegeben ist. Die Einträge in dieser quadratischen Ausführungsmatrix, deren Zeilen- und Spaltengröße der Menge an initialen Token entspricht, bestimmen den minimalen Zeitabstand der Produktionszeit der Token in Zeile und Spalte zwischen zwei aufeinanderfolgenden Iterationen. Der Eintrag in Zeile 1 und Spalte 2 bestimmt also beispielsweise den Abstand zwischen Token 1 und Token 2. Wenn es keine Abhängigkeit zwischen Token gibt, wird der entsprechende Matrixeintrag mit $-\infty$ belegt.

Die Berechnung dieser Ausführungsmatrix $G_{as,ps}$ basiert auf der symbolischen Ausführung von Aktoren, die den minimalen Repetitionsvektor, die Ausführungszeit der Aktoren und die Scheduling-Strategie umfasst. Im Allgemeinen wird dabei von der Strategie der *Self-Timed Execution* ausgegangen, da diese den höchstmöglichen Durchsatz garantiert, indem die Aktoren sofort ausgeführt werden, sobald alle zur Ausführung notwendigen Eingangs-Token produziert sind. Details über die Implementierung dieser symbolischen Ausführung sind in [35] enthalten, deren Anwendung auf das in diesem Abschnitt verwendete SDF-Beispiel ist in Abbildung 6.11 dargestellt. Die im jeweils nächsten Schritt ausgeführten Aktoren sind grau hinterlegt, die Ausführungszeit ist im Aktor selbst annotiert.

Die Ausführungsmatrix kann berechnet werden, wenn alle Aktoren gemäß ihres minimalen Repetitionsvektors und unter Berücksichtigung der gegenseitigen Abhängigkeiten ausgeführt wurden, was einer Iteration entspricht und in Abbildung 6.11 durch die Schritte ①-③ gekennzeichnet ist. So stellt die erste Reihe der Matrix die Abhängigkeit des ersten Token von allen anderen Token dar. In Schritt ③ ergibt sich durch die Abhängigkeitsbedingung $\max(t_3 + 3, t_1 + 1)$ des ersten Token aus Schritt ① eine Abhängigkeit zu Token 3 mit dem Wert 3 und eine Abhängigkeit zu sich selbst mit dem Wert 1. Zu Token 2 existiert keine Abhängigkeit. Wird diese Berechnung mit allen Token durchgeführt, ergibt sich die gesamte Ausführungsmatrix G dieses

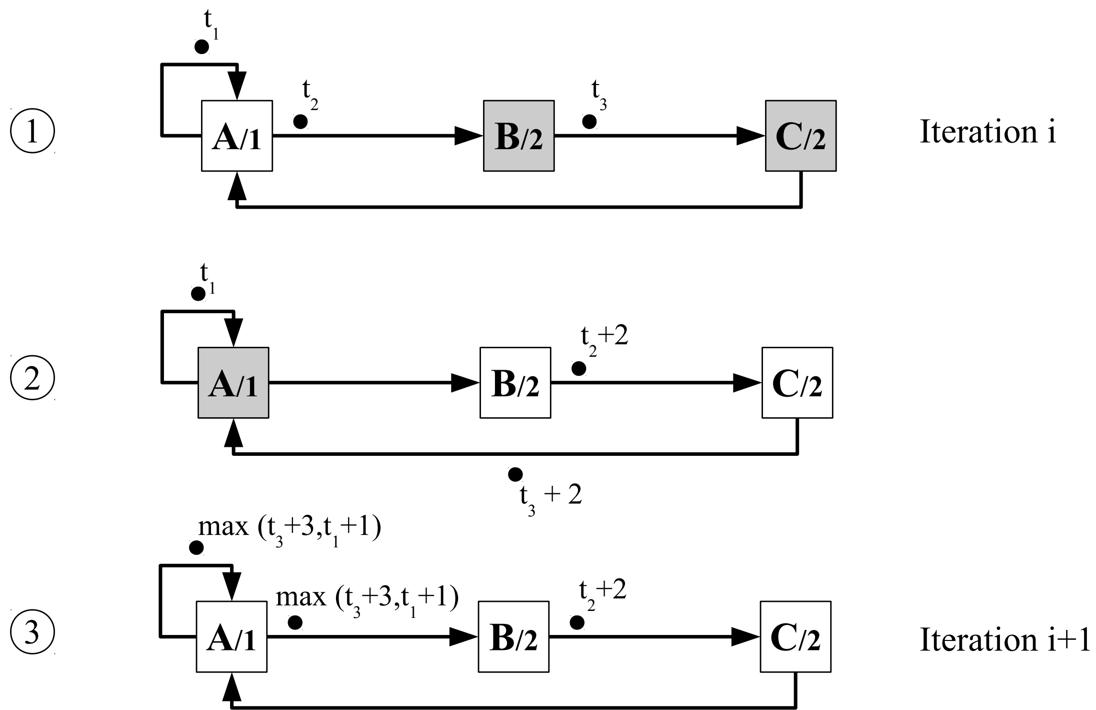


Abbildung 6.11: Symbolische Ausführung der Aktoren eines SDF-Modells

Beispiel-SDF-Modells zu

$$G = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix}$$

Da die Ausführungsmatrix $G_{as,ps}$ nicht nur von den auftretenden Applikationsszenarien sondern auch von den Plattformszenarien der zugrunde liegenden Zielarchitektur abhängt, müssen auch die Aufwände $\delta(ps^i, ps^{i+1})$ für mögliche Wechsel des Betriebsmodus der Plattform zwischen Iteration i und $i+1$ beachtet werden. ps^i bezeichnet in diesem Fall das Plattformszenario, das in Iteration i aktiv ist.

Die Berechnung eines Zeitstempel-Vektors v^{i+1} für Iteration $i+1$ kann dann unter Hinzunahme des vorherigen Zeitstempel-Vektors v^i und der Ausführungsmatrizen nach Gleichung 6.36 berechnet werden. *as* und *ps* sind jeweils die Applikations- bzw. Plattformszenarien in Iteration $i+1$.

$$v_{i+1} = \begin{cases} G_{as,ps} \otimes v^i + \delta(ps^i, ps^{i+1}) & \text{, wenn } ps^i \neq ps^{i+1} \\ G_{as,ps} \otimes v^i & \text{, sonst} \end{cases} \tag{6.36}$$

Für den Beispiel-Graphen in Abbildung 6.9 sind die Plattformszenarien als unterschiedliche Power-Modi PM_1 und PM_2 spezifiziert. Mit gegebener Ausführungsmatrix G_{y,PM_2} und einem Zeitstempel-Vektor v^1 mit dem Wert $[3,3,2]^T$ in Plattformszenario ps_1 ergibt sich mit spezifiziertem Aufwand für den Zustandswechsel $\delta(PM_1, PM_2) = 1$ für den Zeitstempel-Vektor v^2 der nachfolgenden Iteration in Plattformszenario ps_2

$$v^2 = G_{y,PM_2} \otimes v^1 + \delta(ps^1, ps^2) = \begin{bmatrix} 2 & -\infty & 4 \\ 2 & -\infty & 4 \\ -\infty & 6 & -\infty \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} + 1$$
$$= \begin{bmatrix} max\{2+3, -\infty+3, 4+2\} \\ max\{2+3, -\infty+3, 4+2\} \\ max\{-\infty+3, 6+3, -\infty+2\} \end{bmatrix} + 1 = \begin{bmatrix} 7 \\ 7 \\ 10 \end{bmatrix}$$

In Abbildung 6.12 sind die Produktionszeitpunkte der initialen Token in der ersten und zweiten Iteration gemäß einer Ausführung auf der in Abbildung 6.9 dargestellten Hardware-Plattform mit zwei Berechnungskernen skizziert. Der Zeitstempel-Vektor der ersten Iteration hat den Wert $[3,3,2]^T$, was gleichbedeutend ist mit der Aussage, dass Token 1 und Token 2, die nach Abbildung 6.11 Ausgangs-Token von Aktor A darstellen, zum Zeitpunkt 3 produziert werden, wohingegen Token 3 schon zum Zeitpunkt 2 produziert wird. Da danach beide Kerne der Zielarchitektur in den Power-Modus PM_2 überführt werden sollen, kann wegen $\delta(PM_1, PM_2) = 1$ für eine Zeitdauer von einem Zeitschritt kein Aktor auf den Kernen ausgeführt werden. Nach dieser Wechselzeit wird das Applikationsszenario y im neuen Plattform-Zustand PM_2 ausgeführt. Diese zweite Iteration ergibt nach der vorigen Beispielberechnung den Zeitstempel-Vektor $[7,7,10]^T$, was die in der Abbildung markierten Produktionszeitpunkte der Token ergibt.

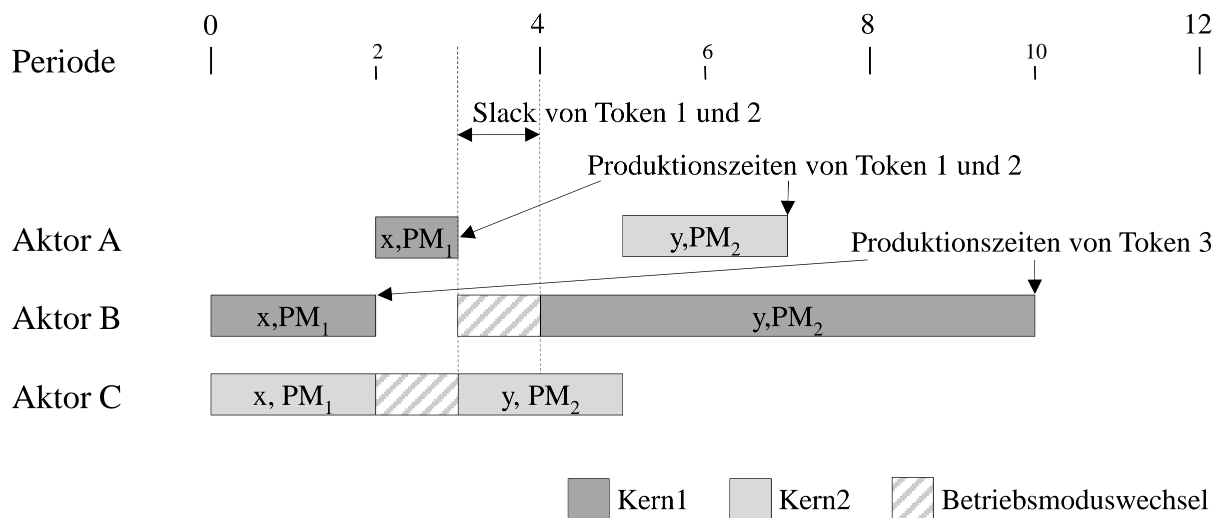


Abbildung 6.12: Durch Zeitstempel-Vektoren berechnete Produktionszeiten der Token

Die zeitliche Abstraktion der einzelnen Ausführung der Aktoren auf Iterationen und somit auf die Zeitpunkte der Produktion der initial im SDF-Modell vorhandenen Token ermöglicht die Überführung des Ansatzes der Multi-Domänen-Szenarien in SDF-Modellen auf eine (max,+)-Algebra und somit auf ein mathematisches Modell, das schnell berechnet werden kann. Diese Abbildung auf ein mathematisches Modell ist notwendig, um prinzipiell viele unterschiedliche Szenarien-Domänen bei der Bewertung berücksichtigen zu können. Die Abstraktion von einzelnen Aktorausführungen

auf Iterationen kann deshalb problemlos durchgeführt werden, weil datenabhängige Szenarien typischerweise mit den vorkommenden Ausführungspfaden korrelieren.

Bei der Optimierung der Energieeffizienz wird das zu optimierende Hardware/Software-System nicht an dessen Leistungsfähigkeit gemessen, sondern an der Einhaltung der an das System gestellten Anforderungen. Unterschiede zwischen den Anforderungen und der Leistungsfähigkeit können dabei zur Optimierung der Energieeffizienz genutzt werden, indem das System in einem Betriebsmodus ausgeführt wird, der die angeforderte Leistung bzw. Funktionalität auch mit einem niedrigeren Energieverbrauch erreicht. In der Regel wird ein bestimmter Durchsatz eines Hardware/Software-Systems gefordert, an dessen Einhaltung sich der zu wählende Betriebsmodus orientiert. In diesem könnte die Ausführung einer Funktionalität also möglichst langsam sein, um den Energieverbrauch zu minimieren (vgl. Abschnitt 4.1.4), aber doch schnell genug, um die Anforderungen an den Durchsatz bzw. die maximale Periode erfüllen zu können.

Der quantitative Unterschied zwischen der maximalen Periode und der aktuellen Ausführung wird als *Slack* (dt.: Schlupf) bezeichnet. Vorhandener Slack bedeutet also, dass das Hardware/Software-System die auszuführende Funktionalität vor Ende der maximalen Periode abgeschlossen hat. Wenn aber eine energieeffiziente Ausführung möglichst langsam sein soll – z.B. durch den Einsatz von Low-Power-Modi auf einer dementsprechenden Zielarchitektur – kann dieser Slack zur Optimierung der Energieeffizienz verwendet werden. Um den Slack im mathematischen Modell der (max,+)-Algebra nutzbar zu machen, wird ein *Slack-Vektor* σ wie folgt definiert bzw. berechnet.

Definition 6.7 (Slack-Vektor)
Sei v^i *der Zeitstempel-Vektor der Iteration* i *und* $\gamma \in \mathbb{R}^+$ *die spezifizierte Anforderung an den (logischen) Durchsatz des modellierten Hardware/Software-Systems. Weiterhin sei der Slack-Vektor vor der ersten Iteration* $\sigma^0 = 0$*. Dann wird der Slack-Vektor* σ^i *der Iteration* i*,* $i > 0$*, durch folgende Gleichung berechnet:*

$$\sigma^i := \sigma^{i-1} - (\frac{1}{\gamma} - (v^i - v^{i-1})) \tag{6.37}$$

Bei der Definition des Slack-Vektors ist entscheidend, dass dieser nur vom Slack-Vektor der letzten Iteration bzw. vom Zeitstempel-Vektor der letzten und aktuellen Iteration abhängig ist. Dadurch sind lediglich diese Informationen zur Berechnung des aktuellen Slack-Vektors notwendig. Dieser relative Abhängigkeit wird bei der Erstellung des Zustandsraums ausgenutzt, welcher die möglichen Lösungen zur optimierten Konfiguration der Hardware-Plattform enthält.

6.2.4 Exploration des Zustandsraums

Durch die Berechnung der Zeitstempel-Vektoren und Slack-Vektoren ist es möglich, einen Zustandsraum aufzubauen, der alle Applikations- und Plattformszenarien enthält, die möglich sind, ohne dass die an das System gestellten Anforderungen – z.B. in Form

des logischen Durchsatzes – verletzt werden. Der daraus resultierende Zustandsraum und die Transitionsfunktion sind folgendermaßen definiert.

Definition 6.8 (Szenarien-Zustandsraum)
Ein Zustand s_j im Zustandsraum S ist ein 3-Tupel bestehend aus einem Zeitstempel-Vektor v, einem Slack-Vektor σ und einem Plattformszenario ps, in dem der Zustand initial erstellt wurde.

Definition 6.9 (Szenarien-Transitionsfunktion)
Die Transitionsfunktion Π berechnet aus dem Zeitstempel-Vektor v^i und dem Slack-Vektor σ^i der Iteration i den nächsten Zeitstempel-Vektor v^{i+1} durch Gleichung 6.36 und den nächsten Slack-Vektor σ^{i+1} durch Gleichung 6.37 für jede Kombination von Applikationsszenarien und Plattformszenarien. Die für Π verwendeten Szenarien werden an der Kante (s_j, s_{j+1}) zwischen zwei Zuständen $s_j, s_{j+1} \in S$ annotiert.

Ausgehend von einer initialen Konfiguration s_0 wird der Zustandsraum iterativ durch die Ausführung der Transitionsfunktion Π aufgebaut. Hierbei wird die Strategie der Breitensuche angewandt, um neue Zustände zu erhalten. Werden aufeinanderfolgende Zustände durch ein- oder mehrmaliges Ausführen der Transitionsfunktion Π erzeugt, bilden diese Zustände einen Pfad durch den Zustandsraum. Im folgenden Abschnitt 6.2.4.1 werden Möglichkeiten aufgezeigt, diesen Zustandsraum zu limitieren. In Algorithmus 6.2 ist die Funktion `buildStateSpace()` zum Aufbau des Zustandsraums noch einmal in Pseudocode dargestellt. Der durch die Transitionsfunktion Π erzeugte Zustand wird nur zum Zustandsraum hinzugefügt, wenn die Funktion `pruneStateSpace()` einen wahren Rückgabewert zurückgibt. Diese Funktion ist als Pseudocode in Algorithmus 6.3 enthalten und berechnet anhand bestimmter Regeln, ob ein Zustand gültig ist und somit in den Zustandsraum aufgenommen werden darf.

Wie bereits erwähnt sollen in diesem Zustandsraum nur diejenigen Konfigurationen enthalten sein, die es dem System ermöglichen, seine Anforderungen einzuhalten. Der quantitative Unterschied zwischen der Leistungsfähigkeit des Systems und den Anforderungen ist durch den Slack-Vektor definiert. Wenn also $||\sigma|| > 0$ für irgendeinen Zustand s nach Ausführung der Transitionsfunktion Π gilt, dann ist Zustand s ungültig, da er die Anforderungsbedingungen nicht erfüllt. Der Slack-Vektor der zweiten Iteration in Abbildung 6.12 mit Zeitstempel-Vektor $[7, 7, 10]^T$ wäre zum Beispiel

$$\begin{bmatrix} -1 \\ -1 \\ -2 \end{bmatrix} - \left(4 - \left(\begin{bmatrix} 7 \\ 7 \\ 10 \end{bmatrix} - \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} \right) \right) = \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}.$$

Das bedeutet, dass $||\sigma|| = 2$ gilt, wodurch die Produktionszeit des Token 3 gegen die Anforderung verstößt, dass der Durchsatz in diesem Beispiel $1/4$ betragen soll, also nach jedem vierten Zeitschritt eine Ausgabe erwartet wird. Die maximale Periode von vier Zeiteinheiten ist in Abbildung 6.12 mit großen Ziffern dargestellt, die kleinen Ziffern dienen lediglich als Orientierungshilfe. Da ein Pfad, der diesen Zustand enthält, somit ebenfalls ungültig wäre, braucht dieser Zustand bei der Suche nicht mehr verfolgt werden.

Algorithmus 6.2 Algorithmus zum Aufbau des Zustandsraums

function buildStateSpace()
  AS Menge der Applikationsszenarien
  PS Menge der Plattformszenarien
  $ps_{init} \in PS$ initiales Plattformszenario
  γ logischer Durchsatz
  $G = \{as \times ps \mid as \in AS, ps \in PS\}$ Menge der Ausführungsmatrizen
  $s = (v_s, \sigma_s, ps_s)$ Zustand bestehend aus Zeitstempel-Vektor, Slack-Vektor, Plattformszenario
  $s_0 = (0, 0, ps_{init})$ initialer Zustand
  $S = \{s_0\}$
  while $S \neq \emptyset$ **do**
    for all $s \in S$ **do**
      $v_{last} \leftarrow v_s$
      $\sigma_{last} \leftarrow \sigma_s$
      for all $g \in G$ **do**
        if $ps_s \neq ps_g$ **then**
          $v_s \leftarrow g \otimes v_{last} + \delta(ps_s, ps_g)$
        else
          $v_s \leftarrow g \otimes v_{last}$
        end if
        $\sigma_s \leftarrow \sigma_{last} - (\frac{1}{\gamma} - (v_s - v_{last}))$
        if `!pruneStateSpace(`s`, `S`, `v_{last}`, `σ_{last}`) &&`
          `containsAllApplicationScenarios(`s`, `S`)` **then**
          $S \leftarrow S \cup \{s\}$
        end if
      end for
    end for
  end while

Als Grundannahme gilt jedoch immer, dass es mindestens einen gültigen Pfad im Zustandsraum gibt, da ansonsten keine Konfiguration zur Ausführung des Hardware/Software-Systems existiert, welche die Anforderungen überhaupt erfüllen kann.

Eine zwingende Voraussetzung für die Nutzung des entwickelten Zustandsraum zur *Ableitung einer optimierten Konfiguration* für das Hardware/Software-System ist dessen geschlossene Form. Das bedeutet, dass der Zustandsraum alle möglichen Konfigurationen abdeckt und dabei eine endlich große Menge an Zuständen umfasst. Gleichzeitig müssen alle vorkommenden Zustände mindestens einen Vorgänger- und einen Nachfolgezustand haben, sodass aus der endlichen Menge an Zuständen eine potentiell unendliche Folge von Zuständen gebildet werden kann. In Abbildung 6.13 ist die Entwicklung des Zustandsraums für das Beispiel-SDF-Modell in den ersten Iterationen und für eine Anforderung an den logischen Durchsatz von 1/4 grafisch dargestellt. Des Weiteren ist der maximal verfügbare Slack auf die Dauer von einer Periode begrenzt, was bei der Limitierung des Zustandsraums eine entscheidende Rolle spielt und in Abschnitt 6.2.4.1 als *Regel 2* erklärt wird. Von dieser Regel betroffene Zustände werden mit gestricheltem Rahmen dargestellt. Zur besseren Übersichtlichkeit sind die aus den Szenarien resultierenden Ausführungsmatrizen in der rechten unteren

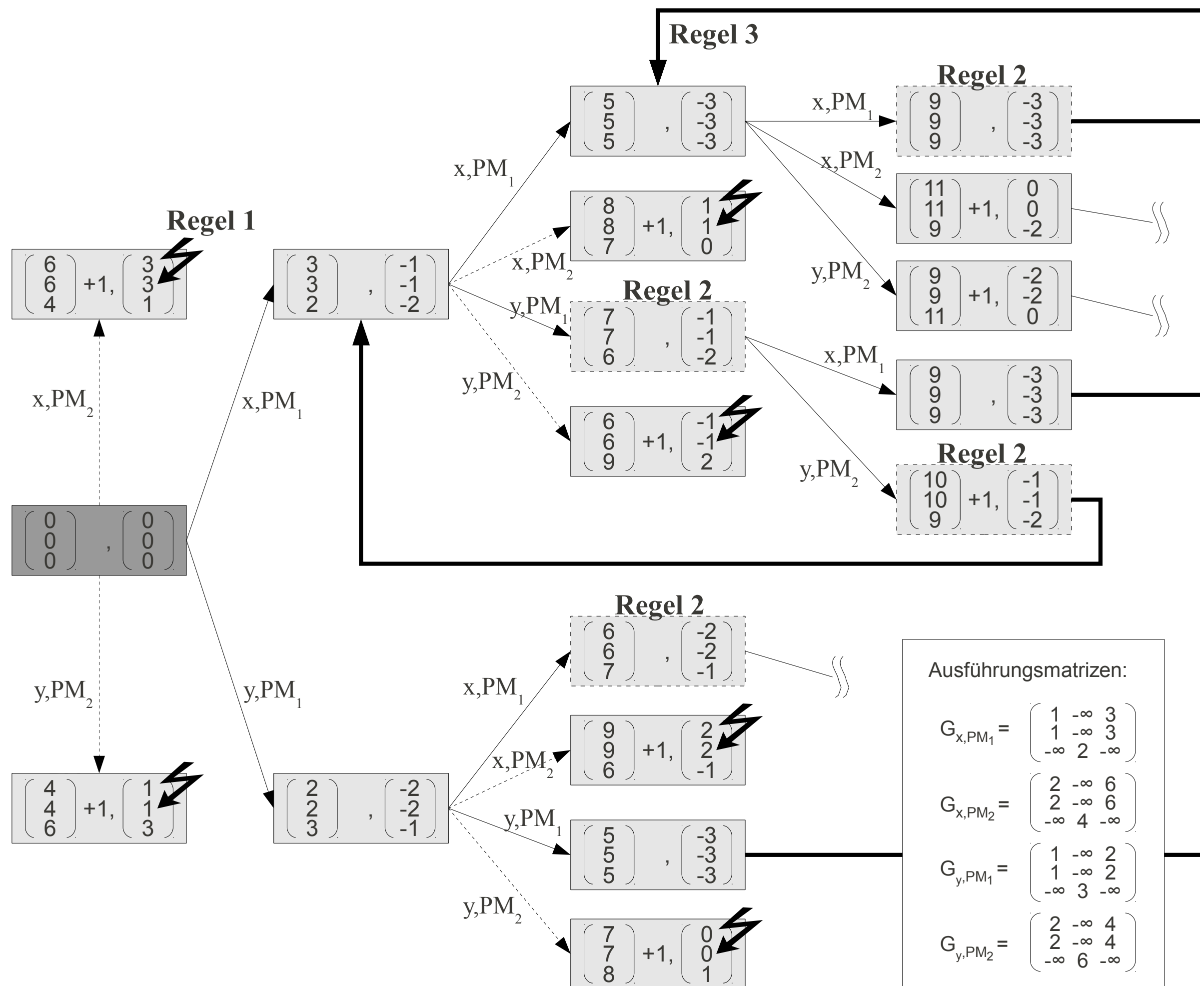


Abbildung 6.13: Aufbau des Zustandsraums mit Durchsatzanforderung 1/4

Ecke aufgelistet. Ausgehend vom Ausgangszustand, der dunkelgrau gekennzeichnet ist, wird auf jeden nachfolgenden Zustand Gleichung 6.36 für vier Ausführungsmatrizen, die wiederum aus zwei Applikationsszenarien und zwei Plattformszenarien bzw. Power-Modi resultieren, angewandt. Gleichzeitig wird für jeden Zustand der zugehörige Slack-Vektor berechnet und die Gültigkeit des Zustands anhand des Slack-Vektor bewertet.

Dieses Verfahren wird solange wiederholt, bis keine Knoten mehr im Zustandsraum existieren, die noch nicht exploriert wurden. Zur Limitierung der Zustandsmenge und zur Überführung des Zustandsraums in eine geschlossene Form müssen allerdings weitere Regeln angewandt werden. Diese werden ausführlich in Abschnitt 6.2.4.1 erläutert.

6.2.4.1 Regelbasierte Limitierung des Zustandsraums

Zur Überführung des Zustandsraums in eine geschlossene Form sind einige Regeln notwendig, die definieren, welche Pfade ungültig sind und welche Pfade bzw. Zustände zusammengefasst werden können. Diese Regeln dienen ebenfalls dazu, die

Zahl der Zustände so gering wie möglich zu halten, wobei trotzdem alle möglichen Konfigurationen abgedeckt werden sollen.

Für die Limitierung des Zustandsraums und dessen Überführung in eine geschlossene Form wurden folgende Regeln entwickelt:

(1) Zustände sind wegen verletzter Anforderungen ungültig, wenn gilt: $\|\sigma\| > 0$. Eventuell auftretende Mehraufwände aufgrund von Moduswechsel auf der Hardware-Plattform sind bereits in der Berechnung des Zeitstempel-Vektors und deswegen auch in der Berechnung des Slack-Vektors enthalten.

(2) Ein Zustand wird zunächst nicht weiter exploriert, wenn er ein spezifiziertes Vielfaches λ der maximalen Periode $1/\gamma$ an Slack angesammelt hat. Wird diese Grenze an potentiellem Slack überschritten, wird der Zustand markiert und dessen Zeitstempel-Vektor wie folgt berechnet:

$$v^i = v^{i-1} + \frac{1}{\gamma}$$

Die Berechnung des Slack-Vektor σ^i wird auf v^i basierend durchgeführt. Die dadurch markierten Zuständen können als *Wartezustände* interpretiert werden. λ grenzt somit auch die Menge der Daten ein, die potentiell im Eingangspuffer des durch den SDF-Graphen modellierten Systems verfügbar sind.

(3) Da die Slack-Vektoren σ^i von der relativen Distanz zwischen den Zeitstempel-Vektoren v^{i-1} und v^i abhängen, sowie von der Durchsatzanforderung γ und dem Slack-Vektor σ^{i-1} des vorigen Iterationsschritts, können zwei Zustände p und q ($p \neq q$) zusammengeführt werden, wenn für ein spezifiziertes ϵ gilt:

$$(\sigma_q \geq (\sigma_p + \delta(ps_p, ps_q))) \ \wedge \ ((\sigma_q - (\sigma_p + \delta(ps_p, ps_q))) \leq \epsilon)$$

Wie zu erkennen ist, gilt dann auch $v_p^{mod\ \gamma} = v_q^{mod\ \gamma}$, wodurch ein Zyklus im Zustandsraum entsteht. Diese Regel ist wegen der relativen Distanz sowohl zwischen Zeitstempel-Vektoren, als auch den Slack-Vektoren aufeinanderfolgender Iterationen unabhängig von der gerade betrachteten Iteration.

Der Wert ϵ legt hierbei den Schwellwert an Slack fest, ab dem Zustände nicht zusammengeführt werden. Unterhalb dieses Schwellwerts können demzufolge Zustände zugunsten einer geringeren Zustandsmenge im Zustandsraum zusammengeführt werden, wobei sich der Slack des resultierenden Zustands aus dem kleinsten absoluten Slack-Vektor der zusammengeführten Zustände ergibt. Für alle anderen Zustände bedeutet das, dass sie durch die Zusammenführung einen Teil des vorhandenen Slack verlieren und dieser nicht für eine Optimierung genutzt werden kann. Die Obergrenze für diesen Slack-Verlust wird durch ϵ bestimmt. Zustände, die als Wartezustände markiert sind, können anhand einer DPM-Strategie behandelt werden, z.B. durch das Abschalten der dadurch nicht benötigten Verarbeitungsressource. Die Optimierung der Energieeffizienz durch DPM wurde bereits in Abschnitt 4.1.4 erläutert.

Algorithmus 6.3 zeigt die Umsetzung der regelbasierten Limitierung des Zustandsraums in Pseudocode-Darstellung. Wie bereits in Algorithmus 6.2 besteht der Zustand s aus einem Zeitstempel-Vektor v_s, einem Slack-Vektor σ_s und einem initialen Plattformszenario ps_s. Durch den Aufruf der Methode `merge()` wird der Zustand st und der Vorgänger von Zustand s verbunden, falls dies aufgrund der Slack-Vektoren möglich ist.

Algorithmus 6.3 Algorithmus zur Limitierung des Zustandsraums

```
function pruneStateSpace(s, S, v_last, σ_last)
s = (v_s, σ_s, ps_s) Zustand bestehend aus Zeitstempel-Vektor, Slack-Vektor, Plattformszenario
// Regel 1
for all token entries te_i in σ_s do
    if te_i > 0 then
        return true
    end if
end for
// Regel 2
for all token entries te_i in σ_s do
    if te_i < (λ ∗ γ) then
        v_s ← v_last + γ
        σ_s ← σ_last
        mark(s)
        return false
    end if
end for
// Regel 3
for all st ∈ S do
    if (σ_st ≥ (σ_s + δ(ps_s, ps_st)))&&((σ_st − (σ_s + δ(ps_s, ps_st))) ≤ ϵ) then
        merge(s, st)
        return true
    end if
end for
return false
```

Während der Erstellung und Limitierung des Zustandsraums kann es vorkommen, dass von einem Zustand aus nicht alle Applikationsszenarien erreichbar sind. Dies geschieht vor allem dann, wenn die Ausführungszeit im schnellsten Betriebszustand und ein Wechsel in diesen Betriebsmodus länger dauern als die maximale Periode. Wenn zur Laufzeit kein oder nur wenig Slack zur Verfügung steht, ist der Wechsel in diesen Betriebsmodus und ein anschließendes Ausführen des Applikationsszenarios aufgrund der Anforderungen nicht möglich. Zumeist ist dann auch die Ausführung in einem anderen Betriebszustand nicht erlaubt, sodass es keine Fortsetzung im Zustandsraum gäbe. Um dies zu verhindern, muss die Verfügbarkeit mindestens eines Betriebsmodus für jedes mögliche Applikationsszenario sichergestellt werden. Diese Anforderung wird beim Aufbau des Zustandsraums überprüft, bevor ein neuer Zustand zum Zustandsraum hinzugefügt wird (siehe Algorithmus 6.2). Ist dies nicht der Fall,

wird ein Algorithmus ausgeführt, der rekursiv solange die vorangehenden Zustände löscht, bis diese Bedingung erfüllt ist. Existieren in einem zu löschenden Zustand andere vorwärtsgerichtete Kanten, d.h. zu anderen Applikationsszenarien, werden diese und alle erreichbaren Zustände bis einschließlich der nächsten schließenden Kanten ebenfalls gelöscht. Da der Zustandsraum generell so aufgebaut wird, dass von einem Zustand zuerst alle nächsten Zustände erzeugt werden, wird der Zustandsraum nach Beendigung der Rekursion automatisch ab dem Zustand weiter aufgebaut, der für alle Applikationsszenarien mindestens einen Übergang besitzt. Algorithmus 6.4 beschreibt den zugehörigen rekursiven Algorithmus in Pseudocode-Darstellung.

Algorithmus 6.4 Überprüfung auf vollständige Abdeckung der Applikationsszenarien

```
function containsAllApplicationScenarios(s, S)
AS Menge der Applikationsszenarien
for all as ∈ AS do
    if !∃ Transition für as then
        eraseAllSuccessors(s)
        s* = getPredecessor(s)
        S* = S\{s}
        containsAllApplicationScenarios(s*, S*)
        return false
    end if
end for
return true
```

In Abbildung 6.14 ist die Abhängigkeit der Anzahl an Zuständen des Zustandsraums des Beispiel-SDF-Modells aus Abbildung 6.9 sowohl von der Durchsatzanforderung als auch vom Wert für ϵ dargestellt. Die Durchsatzanforderung wird dabei in äquidistanten Schritten von $1/4$ bis $1/3$ gesteigert, wobei es sich hierbei wieder um die Angabe des logischen Durchsatzes handelt. Das spezifizierte Vielfache λ der maximalen Periode ist auf den Wert 2 gesetzt.

Anhand der analysierten Werte lassen sich folgende Tendenzen über den Zustandsraum feststellen:

(1) Die Anzahl der gültigen Zustände verringert sich mit Erhöhung von ϵ. Das ist dadurch begründet, dass mehr Zustände zusammengefasst werden können. Dadurch wird aber auch mehr Slack aufgegeben.

(2) Ebenso nimmt die Anzahl der Zustände mit steigender Durchsatzanforderung γ ab, da im Durchschnitt seltener in einen Betriebsmodus mit geringerer Leistungsfähigkeit gewechselt werden kann.

Abbildung 6.15 zeigt schematisch den gesamten Zustandsraum des Beispiel-SDF-Modells aus Abbildung 6.9 für die Belegung $\epsilon = 0$ und die Durchsatzanforderung $1/4$. Zusammengeführte Zustände sind in der Abbildung mit grünen Pfeilen markiert, ungültige Zustände werden rot umrandet und während der Exploration nicht weiter verfolgt. Mit blauem Rahmen sind diejenigen Zustände markiert, die durch die Überschreitung

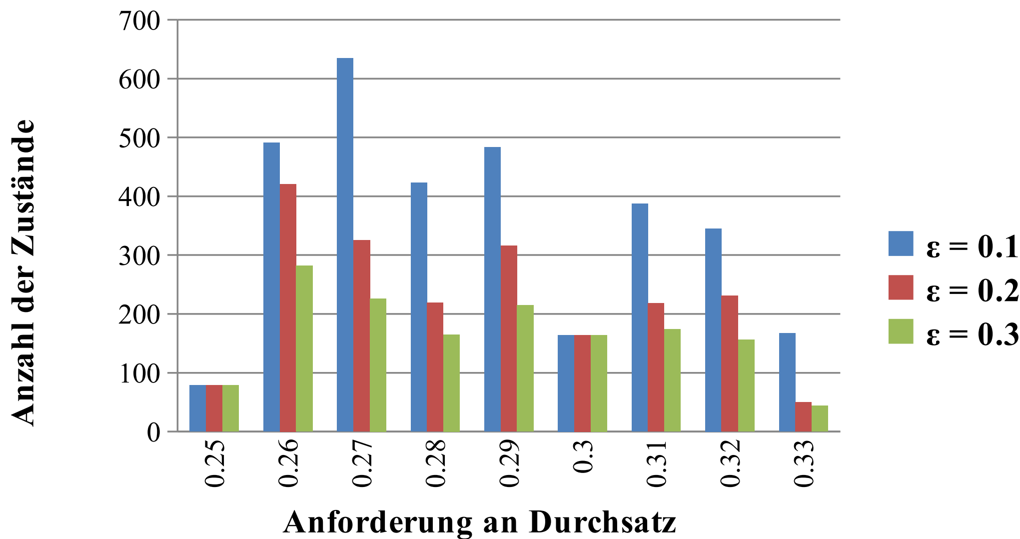


Abbildung 6.14: Größe des Zustandsraums in Abhängigkeit von Durchsatzanforderung und potentiellem Slack-Verlust

des spezifizierten Vielfachen der maximalen Periode entstehen. Insgesamt besteht der Zustandsraum dieses SDF-Modells aus 127 Zuständen, inklusive der ungültigen Zustände. Ein Pfad durch den Zustandsraum beschreibt eine Konfigurationsfolge, die den jeweils enthaltenen Zuständen entspricht.

6.2.4.2 Heuristiken zur Strategieableitung während der Laufzeit

Ein geschlossener Zustandsraum, wie er allgemein in Abschnitt 6.2.4 und speziell in Abschnitt 6.2.4.1 aufgezeigt wurde, beinhaltet alle möglichen und gleichzeitig gültigen Konfigurationen, die das modellierte System während der Ausführung der Tasks annehmen kann. Vor bzw. nach jeder Iteration muss also überprüft werden, in welchem Zustand sich das Hardware/Software-System gerade befindet und welche potentiellen Konfigurationen in Frage kommen.

Der nächste Zustand hängt dabei maßgeblich vom Applikationsszenario ab, das in der nächsten Iteration verarbeitet werden muss. Dieses wird in einem Szenarien-basierten SDF-Modell durch einen *Detektionsschritt* erkannt. In diesem Detektionsschritt werden die Eingangsdaten zur Identifizierung des Applikationsszenarios bei Beginn einer neuen Iteration verwendet. Im SDF-Modell handelt es sich hierbei generell um verschiedene Typen von Eingangs-Token, in einer konkreten Implementierung, die durch das SDF-Modell modelliert werden kann, können sich die enthaltenen Typen von Eingangs-Token aber z.B. auch durch die Menge an zu bearbeitenden Daten oder den Datentypen unterscheiden. Diese müssen allerdings vor der Ausführung der jeweiligen Iteration insoweit bekannt sein, sodass eine eindeutige Abbildung auf die jeweiligen Applikationsszenarien erfolgen kann.

Nach dem Detektionsschritts des Applikationsszenarios erfolgt der *Selektionsschritt*, also die Auswahl des Plattformszenarios bzw. des Power-Modus anhand des zuvor aufgebauten Zustandsraums. Hierzu ist also eine definierte Schnittstelle zum Zugriff

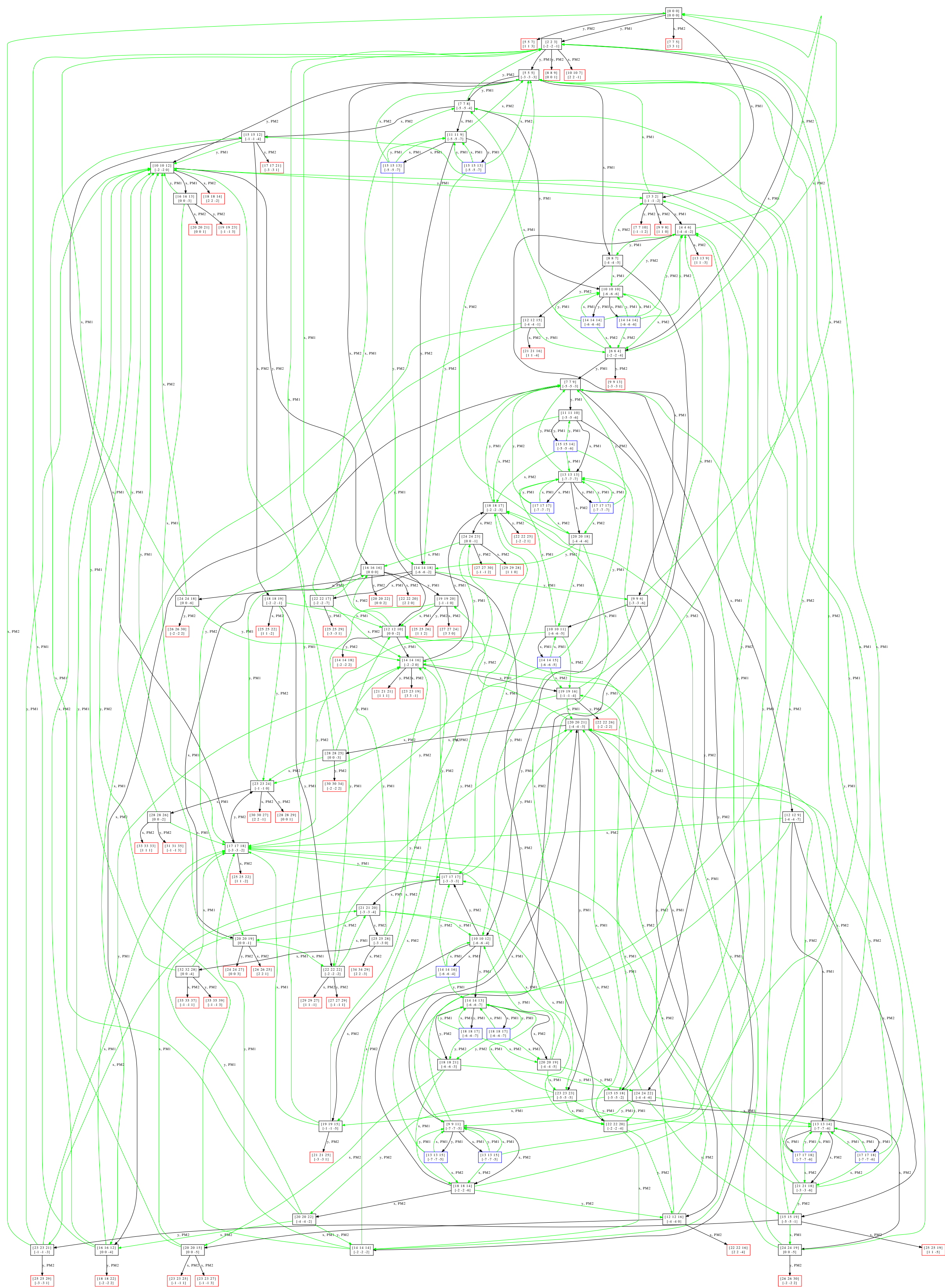


Abbildung 6.15: Überführung des Zustandsraums in eine geschlossene Form

auf den aufgebauten Zustandsraum notwendig, der in einer Implementierung z.B. über ein spezifisches XML-Austauschformat oder über eine spezielle Datenstruktur realisiert werden kann. Der Selektionsschritt erfordert als Eingabe das zuvor detektierte Applikationsszenario und bestimmt abhängig vom aktuellen Zustand im geschlossenen Zustandsraum einen Folgezustand, der gleichzeitig auch den dafür notwendigen Power-Mode bestimmt. Da bei der Erstellung des Zustandsraums auch die auftretenden Mehraufwände durch Zustandsübergänge berücksichtigt sind, wird durch diesen Folgezustand das genaue Zeitverhalten eines etwaigen Wechsels des Betriebsmodus und der Ausführung der Iteration in dem gewählten Betriebsmodus abgebildet. Bei der Auswahl des nächsten Zustands können verschiedene *Heuristiken* angewandt werden, die jeweils eine bestimmte Strategie und somit ein bestimmtes Optimierungsziel verfolgen. Diese Strategien können sowohl durch andere Entwurfsentscheidungen beeinflusst werden, als auch Entwurfsentscheidungen aktiv leiten und Anforderungen an Teile des Hardware/Software-Systems definieren.

Energieoptimierung ohne Vorkenntnis Ein eingebettetes Hardware/Software-System, das keine Rückschlüsse auf das zukünftige Verhalten erlaubt, wird zur Maximierung der Energieeffizienz sobald wie möglich in den Power-Modus versetzt, der den geringsten Energieverbrauch für die Ausführung der nächsten Iteration verspricht. Hierbei werden auch eventuell anfallende Mehraufwände durch notwendige Wechsel der Betriebsmodi einbezogen. Eine Minimierung dieser Mehraufwände über mehrere aufeinanderfolgende Iterationen hinweg bleibt dabei jedoch unberücksichtigt. Diese Heuristik entspricht somit einer Vorausschau, die lediglich eine Iteration umfasst. In markierten Wartezuständen kann zusätzlich überprüft werden, ob der Energieverbrauch durch ein mögliches komplettes Abschalten der ausführenden Ressource weiter minimiert werden kann. Im Allgemeinen kann bei der Energieoptimierung ohne Vorkenntnis kein optimales Ergebnis bezüglich der Minimierung des Energieverbrauchs garantiert werden.

Energieoptimierung mit Vorkenntnis Liegen mehr oder weniger genaue Kenntnisse über das Eintreten bestimmter Applikationsszenarien vor und können daraus Rückschlüsse über das zukünftige Verhalten des Hardware/Software-Systems getroffen werden, kann ein Teil der Mehraufwände beim Zustandswechsel der Power State Machine vermieden werden. Dadurch wird eine Vorausschau über mehrere Iterationen möglich, die in der Strategie berücksichtigt werden kann. Das Vorhandensein von genügend Slack resultiert dabei nicht zwangsläufig in einem Zustandswechsel, sondern es muss ein spezifizierter Schwellwert überschritten werden, der dadurch garantiert, dass das System für eine längere Zeit in einem bestimmten Betriebsmodus ausgeführt werden kann.

Betrachtung der Zuverlässigkeit Die Temperaturentwicklung einer integrierten Schaltung ist ein wichtiger Faktor für deren Robustheit. Hierbei spielen aber nicht nur Maximaltemperaturen, sondern vor allem auch häufig auftretende Temperaturunterschiede eine wesentliche Rolle [107]. Da oftmalige Zustandswechsel der Power State Machine bzw. das Abschalten der Ressource in Wartezuständen unter Umständen direkt zu solchen Temperaturzyklen beitragen, können diese vermieden

werden, indem auch hier ein Schwellwert für die erlaubten Zustandswechsel spezifiziert wird. Hierbei kann auch festgelegt werden, in welcher Häufigkeit bzw. nach wie vielen Iterationen der Betriebsmodus wieder gewechselt werden darf. Da der Zustandsraum prinzipiell eine Propagierung des zur Verfügung stehenden Slacks unterstützt, kann dieser auch nach mehreren Iterationen noch genutzt werden, um dann einen entsprechenden Betriebsmodus zu wählen.

Ein Beispiel für ein Hardware/Software-System mit entsprechender Vorkenntnis wäre eine klassische Multimedia-Applikation, wie z.B. die Dekodierung von multimedialen Daten, bei der eine feste Reihenfolge bestimmter Datenpakete bzw. unterschiedlicher Frames existiert. Die Abfolge der Applikationsszenarien ist also durch die Semantik des Algorithmus bestimmbar und kann deshalb in die Ableitung der Strategie einbezogen werden. Dies ist insbesondere dann effizienter als eine Optimierung ohne Vorkenntnisse, da weniger Moduswechsel durchgeführt werden müssen und deshalb der daraus entstehende Mehraufwand minimiert werden kann.

Im allgemeinen Fall können jedoch keine gesicherten Annahmen über das Auftreten einer bestimmten Reihenfolge von Applikationsszenarien getroffen werden. Da hierbei nur das jeweils nächste Applikationsszenario bekannt ist, muss die Heuristik zur Energieoptimierung ohne Vorkenntnis angewandt werden. Darin wird zwar der für Ausführung der Iteration optimale Betriebsmodus hinsichtlich des Energieverbrauchs gewählt, allerdings erfolgt keine Minimierung der für anfallende Wechsel der Betriebszustände benötigten Energie über mehrere Iterationen hinweg. Deshalb kann wie bereits zuvor erwähnt nicht garantiert werden, dass ein minimales Ergebnis hinsichtlich des Gesamtenergieverbrauchs erzielt wird. Unabhängig davon werden aber auf jeden Fall die an das Hardware/Software-System gestellten Anforderungen, wie z.B. der benötigte Durchsatz, eingehalten, da der Zustandsraum durch die Algorithmen 6.3 und 6.4 lediglich gültige Zustände umfasst.

In Abschnitt 7.3.2 wird die Heuristik zur Energieoptimierung ohne Vorkenntnis auf einen Teil-Algorithmus eines Kamera-basierten Fahrerassistenzsystems aus der Anwendungsdomäne Automobil angewandt. Weiterhin werden Ergebnisse bezüglich des Gesamtenergieverbrauchs bei der Ausführung auf einer Mehrkern-Architektur unter Verwendung verschiedener Betriebsmodi dargestellt.

Kapitel 7

Anwendung und Ergebnisse

Bestrebungen zur Anwendung eines Energiemanagements mit der Ziel der Optimierung der Energieeffizienz sind in vielen Anwendungsdomänen offensichtlich und vom Markt auch ausdrücklich gefordert. Kunden entscheiden sich immer häufiger für Lösungen, die unter permanenter Berücksichtigung der Energieeffizienz entwickelt wurden. Durch die starke Abhängigkeit von tragbaren Energiespeichern spielen dabei Endgeräte für mobile Anwendungen und Kommunikation eine Führungsrolle. In dieser Anwendungsdomäne werden bereits seit geraumer Zeit nicht zu vernachlässigende und zum Teil kostenintensive Anstrengungen unternommen, die Leistungsaufnahme der Endgeräte unter allen möglichen Szenarien der Anwendung zu analysieren und ein möglichst zielgerichtetes Energiemanagement zu entwickeln.

Aber auch in der Anwendungsdomäne Automobil findet ein Umdenken statt, das nicht zuletzt durch den Trend zur elektrischen Mobilität und der damit verbundenen Abhängigkeit der Reichweite vom Gesamtenergieverbrauch des Fahrzeugs begründet ist. Dass dabei auch dem Aspekt des Energiemanagements auf Ebene der E/E-Architektur[1] in Form eines *Teilnetzbetriebs* eine immer größere Rolle zu Teil wird, lässt sich an folgendem Zitat aus [85] erkennen: „In Standardisierungsaktivitäten wie AUTOSAR werden Basismechanismen als Erweiterung des Fahrzeugzustands-Management spezifiziert, die das Abschalten von Teilnetzen, Steuergeräten oder Funktionen ermöglichen sollen. Es werden im Wesentlichen drei Konzepte verfolgt: Beim ersten Konzept (ECU Degradation) ist es den auf der AUTOSAR-Basis-Software aufbauenden Software-Komponenten erlaubt, einen Wechsel des Hardware-Betriebsmodus zu veranlassen. Dieser Wechsel ist ein rein lokales Phänomen und ermöglicht es beispielsweise, nicht benötigte Lasten oder auch Berechnungseinheiten abzuschalten oder mit verminderter Leistung zu betreiben. [...]“

In diesem Kapitel soll deshalb die Anwendungen der entwickelten Methoden zur Simulation und Analyse von eingebetteten Hardware/Software-Systemen mit dem Ziel der Optimierung der Energieeffizienz sowohl domänenübergreifend als auch speziell in den Anwendungsdomänen Automobil und der elektrischen Mobilität dargestellt werden. Zuerst wird die Generierung und Verwendung virtueller Plattformen zur Überprüfung funktionaler und nicht-funktionaler Eigenschaften inklusive eines reaktiven

[1] Elektrisch/Elektronische Architektur

Energiemanagement anhand mehrerer Beispielanwendungen vorgestellt. Danach wird gezeigt, wie die Simulation virtueller Plattformen durch die Integration domänenspezifischer Simulationsumgebungen zur Simulation Cyber-Physischer Systeme erweitert werden kann, um somit eine realistische Stimulierung des virtuellen Hardware/Software-Systems zu erreichen. Hierdurch können neben der Überprüfung von funktionalen und nicht-funktionalen Eigenschaften auch Rückschlüsse auf die Leistungsaufnahmen einzelner Systemkomponenten und die damit verbundene Temperaturentwicklung gezogen werden. Zuletzt sollen die entwickelten Optimierungsalgorithmen auf eine reale Hardware-Architektur mit mehreren verfügbaren Berechnungskernen für ein komplexes Fahrerassistenzsystem angewandt werden.

7.1 Konfiguration und Optimierung auf virtuellen Plattformen

Der Schwerpunkt dieses Abschnitts liegt auf der Anwendbarkeit der entwickelten Methoden zur Modellierung, Generierung und Simulation virtueller Plattformen für eingebettete Hardware/Software-Systeme mit dem Ziel der Überprüfung funktionaler und nicht-funktionaler Eigenschaften. Insbesondere soll dabei die Konfiguration der Ausführungsparameter und des Energiemanagements evaluiert und das Verhalten der entsprechenden Komponenten unter dem Einfluss dieser Konfiguration analysiert werden. Anhand mehrerer Beispielanwendungen sollen für eine Konfiguration relevante Fragen beantwortet werden, z.B. inwieweit das Energiemanagement von einer parallelisierten Ausführung der Funktionalität profitiert oder welchen Anteil einzelne Komponenten am Gesamtverbrauch besitzen.

7.1.1 Automatische Generierung der Ausführungsplattform

Zur Beurteilung der Leistungsfähigkeit des hauptsächlich in Abschnitt 5.3 beschriebenen Generierungsprozesses, der auf einem Systemmodell in UML für die Funktionalität und SysML für die zugrunde liegende Hardware-Plattform basiert, dienen die generierten Zeilen Quelltext des Simulationsmodells, das die Ausführung des Systemmodells auf einer virtuellen Ausführungsplattform erlaubt, als Metrik. Die Menge des generierten Quelltexts wird in den nachfolgenden Erläuterungen und der Ergebnistabelle 7.1 durch die generierten Zeilen Quelltext – unter Verwendung der englischsprachigen Notation *Lines of Code (LoC)* – angegeben. Als Eingabemodell für die automatischen Transformationen wird das *X-By-Wire*-System aus Abschnitt 5.1 verwendet.

Die in Ergebnistabelle 7.1 enthaltenen generierten Quelltextzeilen entsprechen dem statisch generierten Teil des virtuellen Hardware/Software-Systems. Eingebundener funktionaler Quelltext, wie z.B. der Zugriff auf die Programmierschnittstelle und Treiber der Steuereinheit, und Quelltext aus Bibliotheken, wie z.B. der Scheduler-Komponente, sind in dieser Ergebnisübersicht nicht enthalten.

Die Anwendung der Transformationen, sowohl von Modell zu Modell als auch von Modell zu Quelltext, dauert auf einem Standard-Entwicklungsrechner weniger

Tabelle 7.1: Ergebnisse generierter statischer Quelltext für X-By-Wire-System

Beschreibung	Typ	LoC
SteeringDevice	Software-Komponente	30
wheel_data	Datenstruktur	15
input_event	Datenstruktur	12
SteeringInterface	Schnittstelle der Software-Komponente	13
SteeringDeviceModule	Wrapper für Software-Komponente	33
SteeringDeviceInterfaceModule	Schnittstelle des Wrapper	15
SteeringUnit	Hardware-Komponente	44
CockpitControl	Software-Komponente	23
CockpitControlModule	Wrapper Software-Komponente	34
CockpitControlThread	Ausführender Thread in Wrapper	37
CPU1	Hardware-Komponente	61
SteerByWireSystem	Integration in Gesamtsystem	31
Gesamt		348

als 5 Sekunden. Selbst für das relativ kleine Beispielmodell des X-By-Wire-Systems werden insgesamt 348 LoC automatisch generiert. Zum einen ist der generierte Quelltext *„correct by construction“*, d.h. aufgrund des automatischen Generierungsprozesses und fehlender manueller Entwicklungsarbeit ohne Implementierungsfehler. Durch die Verlagerung von manuellen hin zu automatisierten Schritten wird auch der Aufwand für die Verifizierung des generierten Simulationsmodells deutlich reduziert. Zum anderen ist der automatische Generierungsprozess *skalierbar*, da er aufgrund der linearen Abhängigkeit der Transformationen von den im Eingabemodell enthaltenen Modellelementen auf beliebig große Eingabemodelle anwendbar ist. Insbesondere mit steigender Größe und Komplexität der Systemmodelle bedeutet dies eine zum Teil erhebliche *Aufwandsreduktion* gegenüber einer manuell durchgeführten Implementierung.

7.1.2 Ausführung und Energiemanagement in virtuellen Prototypen

Die Ergebnisse bei der Verwendung der *virtuellen Ausführungsplattform* (VEP) zur Simulation eingebetteter Hardware/Software-Systeme sind in mehrere Bereiche aufgeteilt. Zunächst wird der Aufwand für die Ausführung von Simulationsmodellen, deren Basisblöcke mit analysierten Ausführungszeiten und der Leistungsaufnahme annotiert sind, anhand mehrerer repräsentativer Anwendungen dargestellt. Hierbei soll insbesondere der Mehraufwand der in der VEP enthaltenen abstrakten Verwaltungsschicht abgeschätzt werden, die z.B. für die Ausführung auf mehreren virtuellen Berechnungskernen oder für das Energiemanagement verantwortlich ist. Dieser Mehraufwand soll dabei mit den unterschiedlichen Konfigurationen in Verbindung gebracht werden, in denen das Simulationsmodell auf der VEP ausgeführt wird. Zuletzt wird gezeigt, wie

ein Energiemanagement auf dem virtuellen Prototypen durchgeführt werden kann und welche Auswirkungen dieses Management auf die Leistungsaufnahme des Hardware/ Software-Systems impliziert.

Die hier verwendeten Applikationen bzw. Programme bestehen aus modifizierten Testalgorithmen aus dem Mälardalen WCET Benchmark [47]. Darin enthalten sind ein synthetischer Algorithmus mit vielen bedingten Verzweigungen (Cover), ein Algorithmus für Matrixmultiplikationen (Matmult), ein Programm zur Simulation eines Petri-Netzes (Nsichneu) und ein Sortieralgorithmus (InsertSort). In den folgenden Messungen werden diese Programme jeweils 1000 Mal ausgeführt. Um die Ergebnisse bezüglich der Simulationsdauer vergleichbar zu machen, teilt sich dieser Wert der Wiederholungen bei einer Ausführung auf der VEP auf die verwendeten virtuellen Berechnungskerne der Zielplattform auf, da diese nur quasi-parallel ausgeführt werden. Aufgrund der Durchführung von mehr als 50 Simulationsdurchläufen wird der Median verwendet, um einen stabilen Mittelwert für die Simulationsdauer zu erhalten.

Tabelle 7.2: Mehraufwand durch Integration der VEP

Benchmark	Zyklen	Basisblöcke	VEP	Simulationsdauer (Median)	Faktor
Matmult	35858000	890000		0,298 s	
			✓	0,354 s	1,1 ×
Nsichneu	7631000	507000		0,245 s	
			✓	0,288 s	1,2 ×
InsertSort	820000	57000		0,037 s	
			✓	0,135 s	3,6 ×
Cover	2787000	489000		0,210 s	
			✓	0,261 s	1,2 ×

Zuerst wird in Tabelle 7.2 die Simulationsgeschwindigkeit der virtuellen Plattform unter Verwendung der VEP und die Ausführung der gleichen Anwendung direkt im SystemC-Simulationskern verglichen (zur Unterscheidung siehe Abschnitt 5.2). Die VEP ist in diesem Vergleich mit einer maximal möglichen lokalen Abweichung der Simulationszeit von der globalen Simulationszeit von 10 µs und einer Ablaufplanung mit Zeitschlitzen der Länge 100 µs konfiguriert. Es werden dabei jeweils 2 Dispatcher zur Ausführung verwendet, d.h. die Verwaltungsschicht verteilt die zu bearbeitenden 1000 Wiederholungen auf 2 virtuelle Berechnungskerne. Die Zykluszeit der virtuellen Berechnungskerne beträgt immer 2 ns. Als Ergebnis ist festzustellen, dass die VEP keinen signifikanten Aufwand bezüglich der Simulationsgeschwindigkeit hat, obwohl sie eine abstrakte Verwaltungsschicht bzw. ein abstraktes Betriebssystem implementiert, die die nicht-funktionalen Eigenschaften der ausgeführten Benchmarks nicht direkt, sondern anhand der definierten Ablaufstrategie an den SystemC-Simulationskern weiterleitet. Der durch die Integration der VEP verursachte Aufwand wird kleiner, je mehr

Basisblöcke innerhalb des Programms ausgeführt werden – zu sehen an der InsertSort-Benchmark. Dies liegt daran, dass die VEP die nicht-funktionalen Eigenschaften der Ablaufstrategie entsprechend akkumuliert an den Simulationskern weiterleitet und sich bei steigender Blockanzahl somit der dadurch erreichte Geschwindigkeitsvorteil erhöht.

Nach der allgemeinen Untersuchung der VEP soll im nächsten Schritt die Simulationsgeschwindigkeit bei unterschiedlichen Konfigurationen der VEP verglichen werden. Dazu werden in entsprechenden Experimenten, die in Tabelle 7.3 dargestellt sind, jeweils die Parameter für den Zeitschlitz der Ablaufplanung und die maximale lokale Abweichung signifikant verändert. In einer Simulation wird der Zeitschlitz von 100 µs auf 0,5 µs reduziert, um die Aktivität des Scheduler in der VEP zu erhöhen. In der darauf folgenden Simulation wird die maximale lokale Abweichung von 10 µs auf 0,2 µs herabgesetzt. Dadurch wird in gleichem Maße die Anzahl der benötigten Synchronisationsprozesse erhöht. Der Vergleich der Simulationsdauer in den unterschiedlichen VEP-Konfigurationen zeigt, dass die Simulationsgeschwindigkeit im Fall der um den Faktor 50 reduzierten maximalen lokalen Abweichung leicht erhöht wird, da die lokale Simulationszeit der Threads entsprechend öfter mit der globalen Simulationszeit synchronisiert werden muss. Der erhöhte Aufwand durch öfter durchgeführte Thread-Wechsel im Scheduler beeinflusst die Simulationsgeschwindigkeit dagegen kaum.

Tabelle 7.3: Simulationsdauer bei unterschiedlichen VEP-Konfigurationen

Benchmark	TMDA-Slot	max. lokaler Offset	Simulationsdauer (Median)	Vergleich
Matmult	100 µs	10 µs	0,354 s	
	0,5 µs	10 µs	0,361 s	+ 2 %
	100 µs	0,2 µs	0,715 s	+ 202 %
Nsichneu	100 µs	10 µs	0,288 s	
	0,5 µs	10 µs	0,287 s	+/- 0
	100 µs	0,2 µs	0,337 s	+ 17 %
InsertSort	100 µs	10 µs	0,135 s	
	0,5 µs	10 µs	0,136 s	+/- 0
	100 µs	0,2 µs	0,136 s	+/- 0
Cover	100 µs	10 µs	0,261 s	
	0,5 µs	10 µs	0,261 s	+/- 0
	100 µs	0,2 µs	0,286 s	+ 9 %

In weiteren Experimenten wird der Aufwand eines zusätzlichen Energiemanagements auf der virtuellen Ausführungsplattform gegenüber einer Ausführung ohne Energiemanagement in der Standardkonfiguration untersucht. Die Ergebnisse die-

ser Untersuchung sind in der nachfolgenden Tabelle 7.4 dargestellt. Zusätzlich zur Auswertung der Annotationen der Leistungsaufnahme, die auf einem instruktionsbasierten Leistungsmodell basiert, wird der Betriebsmodus der zur Verfügung stehenden Berechnungskerne an die derzeitige Auslastung angepasst. Dieses Kombination aus instruktionsbasiertem und zustandsbasiertem Leistungsmodell zur Steuerung eines reaktiven Energiemanagements wurde in Abschnitt 5.4.2 erklärt. Auch in diesen Resultaten zeigt sich, dass die Simulationsperformanz der VEP trotz der Integration einer energieeffizienten Ablaufplanung nur unwesentlich erhöht wird.

Tabelle 7.4: Mehraufwand durch Energiemanagement

Benchmark	Energiemanagement	Simulationsdauer (Median)	Vergleich
Matmult		0,354 s	
	✓ (Nominal/Low-Power Mode)	0,365 s	+ 3 %
Nsichneu		0,288 s	
	✓ (Nominal/Low-Power Mode)	0,291 s	+ 1 %
InsertSort		0,135 s	
	✓ (Nominal/Low-Power Mode)	0,139 s	+ 3 %
Cover		0,261 s	
	✓ (Nominal/Low-Power Mode)	0,268 s	+ 3 %

Zuletzt wird das Ergebnis eines Energiemanagements auf der virtuellen Plattform untersucht. Tabelle 7.5 zeigt dazu einen Vergleich dreier unterschiedlicher Abläufe der Simulation. Nsichneu wird dabei so häufig ausgeführt, dass ein Berechnungskern über die ganze Simulationsdauer hinweg nahezu ausgelastet ist. Für eine Gesamtsimulationsdauer von 10 ms sind dies insgesamt 500 Ausführungen. Während dieser Zeit wird der Cover-Algorithmus 1000 Mal ausgeführt. Ohne ein Energiemanagement ergeben sich somit akkumulierte 20 ms im Betriebsmodus *Nominal Mode*. Da die Laufzeit des Nsichneu-Algorithmus ca. 2,7 mal länger ist, kann im zweiten Vergleichsszenario mit Energiemanagement eine ausführende Ressource entsprechend zwischen den beiden zur Verfügung stehenden Betriebsmodi *Nominal Mode* und *Low-Power Mode* wechseln. Für jeden Wechsel des Betriebsmodus muss jeweils die Zeit bis zur Erreichung eines stabilen Zustand des jeweiligen Kerns gewartet werden, bevor mit der Ausführung begonnen werden kann, was auf dieser Plattform 200 ns entspricht. Insgesamt ergeben sich damit ca. 600 µs an Wechselzeit im ersten Vergleichsszenario. Im dritten Vergleichsszenario wird statt dem Cover-Algorithmus der Algorithmus InsertSort ausgeführt. Da dessen Laufzeit ca. ⅓ der Laufzeit des zu ersetzenden Cover-Algorithmus beträgt, kann er in derselben Zeit drei Mal öfter ausgeführt werden, was zu einer vergleichbaren akkumulierten Zeit im Betriebsmodus *Nominal Mode* führt. Durch die periodische Ausführungen der Algorithmen entstehen in diesem Fall allerdings wesentlich mehr Wechsel zwischen den Betriebsmodi, was im Gegensatz zu Szenario zwei zu einer deutlich reduzierten Laufzeit im Betriebsmodus *Low-Power Mode* führt. Trotz

der Mehraufwände aufgrund der Moduswechsel ist in beiden Szenarien jedoch die Leistungsaufnahme wegen der zeitweisen Ausführung des Betriebsmodus *Low-Power Mode* über die Zeit geringer als im Szenario ohne Energiemanagement. Der Vergleich zwischen den Szenarien zwei und drei zeigt, wie durch die virtuelle Ausführungsplattform die Aspekte Leistungsaufnahme und Energiemanagement in einen virtuellen Prototypen eines eingebetteten Hardware/Software-Systems ganzheitlich integriert werden. Dies erlaubt neben der funktionalen Simulation eine Analyse der durch diese Aspekte abgedeckten nicht-funktionalen Eigenschaften, die sogar das dynamische Verhalten des simulierten Systems berücksichtigt.

Tabelle 7.5: Vergleich der Ausführung mit bzw. ohne Energiemanagement

Energie-Mngt.	Ausführungen			Akk. Ausführungszeit pro Mode		
	Nsichneu	InsertSort	Cover	Nominal	Low-Power	Wechsel
	500	-	1000	20000 µs	-	
✓	500	-	1000	13505 µs	5894,6 µs	600,4 µs
✓	500	3000	-	13251 µs	5348,6 µs	1400,4 µs

Eine Aktivierung des Energiemanagement kann also zu einer insgesamt geringeren Leistungsaufnahme über dieselbe Ausführungszeit gesehen führen, was einer Maximierung der Energieeffizienz entspricht. Welche weiteren Auswirkungen eine geringere Leistungsaufnahme pro definierter Zeit hat, wird in Abschnitt 7.2.2.3 mithilfe einer Analyse der Temperaturentwicklung auf Chip-Ebene visualisiert. Die Leistungsaufnahme wird dabei jedoch nicht akkumuliert, sondern zur Erhöhung der räumlichen Genauigkeit für jede funktionale Einheit der Hardware-Plattform bewertet.

7.2 Simulative Analyse Cyber-Physischer Systeme

Bei der Verwendung virtueller Prototypen für Hardware/Software-Systeme wird oftmals die Umgebung, in die das System eingebettet ist, nur unzureichend berücksichtigt [74]. Tatsächlich stehen beide Seiten jedoch in starker gegenseitiger Abhängigkeit, was durch eine isolierte Betrachtung nicht abgebildet werden kann. *Cyber-Physische Systeme (CPS)* stellen einen Ansatz dar, computertechnische Aspekte und die physische Welt hinsichtlich einer ganzheitlichen Betrachtungsweise zu kombinieren.

In diesem Abschnitt wird deshalb die Simulation und Analyse virtueller Plattformen bestehend aus Hardware und Software durch die Integration domänenspezifischer Simulationsumgebungen für die Repräsentation der Umgebung erweitert. Zum einen kann dadurch eine dem umgebenden System und dem jeweiligen Szenario entsprechende Stimulierung des virtuellen Hardware/Software-Systems erreicht werden. Zum anderen kann die Wirkung des Hardware/Software-Systems auf das Verhalten des umgebenden Systems dargestellt werden. Das hauptsächliche Ziel dieses Ansatzes ist die simulative Analyse funktionaler und nicht-funktionaler Eigenschaften unter Berücksichtigung der gegenseitigen Abhängigkeiten zwischen dem eingebetteten System

und dessen Umgebung.

Dazu wird in einem erste Schritt die Entwicklung einer *generischen Plattform zur Co-Simulation* beschrieben, welche die Integration verschiedener spezifischer Simulationsumgebungen aus der Anwendungsdomäne Automobil verwaltet [70]. Basierend auf dieser Co-Simulationsplattform können durch die resultierende Stimulierung des enthaltenen virtuellen Hardware/Software-Systems relevante Eigenschaften wie das funktionale und temporale Verhalten, sowie die Leistungsaufnahme und die dadurch hervorgerufene Temperaturentwicklung einzelner Komponenten analysiert werden.

7.2.1 Plattform zur Co-Simulation

Um exemplarisch den Nutzen der Co-Simulationsplattform in der Anwendungsdomäne Automobil aufzuzeigen, wird im Folgenden die Kopplung einer Fahrdynamiksimulation, einer Simulation des Antriebsstrangs und einer Simulation eines virtuellen Hardware/Software-Systems in Form einer *E/E-Architektur* dargestellt. Die Kopplung dieser verschiedenen Simulationsumgebungen soll vor allem zur Verifikation des funktionalen und nicht-funktionalen Verhaltens bzw. der damit einhergehenden Prüfung der funktionalen Sicherheit eingesetzt werden. Es soll aber dadurch auch ermöglicht werden, energieeffiziente Fahr- und Betriebsstrategien im Zusammenspiel einer gesamtheitlichen Repräsentation des Fahrzeugs simulieren und bewerten zu können, was im Hinblick auf eine umfassende Energieverbrauchsanalyse unumgänglich ist [65].

Insbesondere muss bei der Umsetzung der Co-Simulationsplattform auf die Synchronisation der Simulationskerne geachtet werden. Dies stellt aufgrund der unterschiedlichen Charakteristika der beteiligten Simulationskerne eine große Herausforderung dar. So müssen zeitgesteuerte und ereignisbasierte Simulationskerne auf eine Art und Weise kombiniert werden, die folgende Anforderungen erfüllt [106]:

- synchronisiertes Voranschreiten der Simulationszeit
- gegenseitige Aktivierung von Ereignissen
- Austausch von Datenwerten

Eine generelle Anforderung an die Entwicklung der Co-Simulationsplattform ist deren *modularer Aufbau*, sodass die Funktionalität nicht von einzelnen Simulationsumgebungen abhängig ist. Weiterhin ist der Aufwand für die Synchronisation der Simulationskerne ein entscheidender Faktor für die Simulationsgeschwindigkeit der gesamten Plattform. Daraus folgt die Notwendigkeit einer möglichst *losen Kopplung* der Simulationskerne, um eine gegenseitige Beeinträchtigung der Simulationsgeschwindigkeit weitestgehend zu verhindern.

7.2.1.1 Aufbau der Co-Simulationsplattform

Die Landschaft existierender Simulationsumgebungen zeichnet sich durch eine große Heterogenität aus, sodass in der Regel eine spezifische Simulationsumgebung für die jeweilige Domäne existiert. Um die individuellen Stärken der einzelnen Umgebungen

nutzen zu können und eine Interaktion untereinander zu ermöglichen, muss eine generische Co-Simulationsplattform für die Anwendungsdomäne Automobil, wie in Abbildung 7.1 dargestellt, entwickelt werden.

Als Integrationsplattform für die Co-Simulation wird *MATLAB/Simulink* (vgl. Abschnitt 2.3.3 des Grundlagenkapitels) verwendet. Zur Simulation der Fahrzeugdynamik wird die Simulationsumgebung *IPG Carmaker* integriert, die sowohl ein detailliertes Fahrzeugmodell inklusive Fahrwerk, Aerodynamik, Lenkung und Reifen, als auch ein konfigurierbares Umgebungsmodell für Fahrer, Fahrbahn und Fahrszenarien bietet. Da Carmaker lediglich ein rudimentäres Modell eines Antriebsstrangs besitzt, wird dieses durch die Integration von *AVL Cruise* als Simulationsumgebung für detaillierte Antriebsstrangmodelle verfeinert. Zur Simulation der E/E-Architektur wird *SystemC* in die Co-Simulationsplattform integriert. Auf diesem virtuellen Modell des Hardware/Software-Systems lassen sich typische Anwendungen wie Fahrerassistenzsysteme unter Berücksichtigung der zugrunde liegenden Zielarchitektur ausführen.

Eine entscheidende Komponente zur Synchronisation und Interaktion der Simulationskerne ist die Kommunikation über einen gemeinsamen Speicherbereich, in dem die jeweils benötigten Simulationswerte in Kombination mit einem Zeitstempel abgelegt werden, um sie den Simulationsumgebungen zum richtigen Zeitpunkt zur Verfügung stellen zu können. Diese Komponente ist somit ebenfalls für die globale Synchronisation zuständig, die im nachfolgenden Abschnitt 7.2.1.2 näher beschrieben wird. Die Kommunikation zwischen gemeinsamem Speicher und den jeweiligen Simulationsumgebungen erfolgt über Socket-Verbindungen des auf den Simulationsrechnern verwendeten Betriebssystem. Da Socket-Verbindungen durch den ISO/OSI-Standard plattformunabhängig spezifiziert sind, ist sogar die parallele Ausführung der Simulationsumgebungen auf unterschiedlichen Plattformen und auf unterschiedlichen Simulationsrechnern möglich.

7.2.1.2 Synchronisation

Um eine möglichst hohe Simulationsgeschwindigkeit zu erreichen, wird eine Synchronisation der Simulationskerne nur dann durchgeführt, wenn eine Interaktion zwischen den einzelnen Simulationsumgebungen notwendig ist. Zur weiteren Beschleunigung werden die zeitgesteuerten Simulationsumgebungen mit einer variablen Schrittweite durchgeführt, sodass auf eine periodische Synchronisation nach der kleinsten gemeinsamen Schrittweite verzichtet werden kann. Die zeitgesteuerten Simulationsumgebungen passen ihre Schrittweite dadurch so an, dass die Differenz der Ausgabewerte in aufeinanderfolgenden Simulationsschritten unter einem definierten Schwellwert liegt.

Bestimmung des nächsten Simulationsschritts Entscheidend für die Möglichkeit der Co-Simulation durch Synchronisierung der Simulationskerne ist die Fähigkeit, den nächsten Simulationsschritts bzw. die Schrittweite akkurat vorhersagen zu können. Da der SystemC-Simulationskern ereignisbasiert arbeitet, existiert innerhalb des Simulationskerns eine Liste mit ablaufbereiten Prozessen und deren nächste Aktivierung. Nach Beendigung eines Simulationszyklus kann also die nächste Simulationsschrittweite

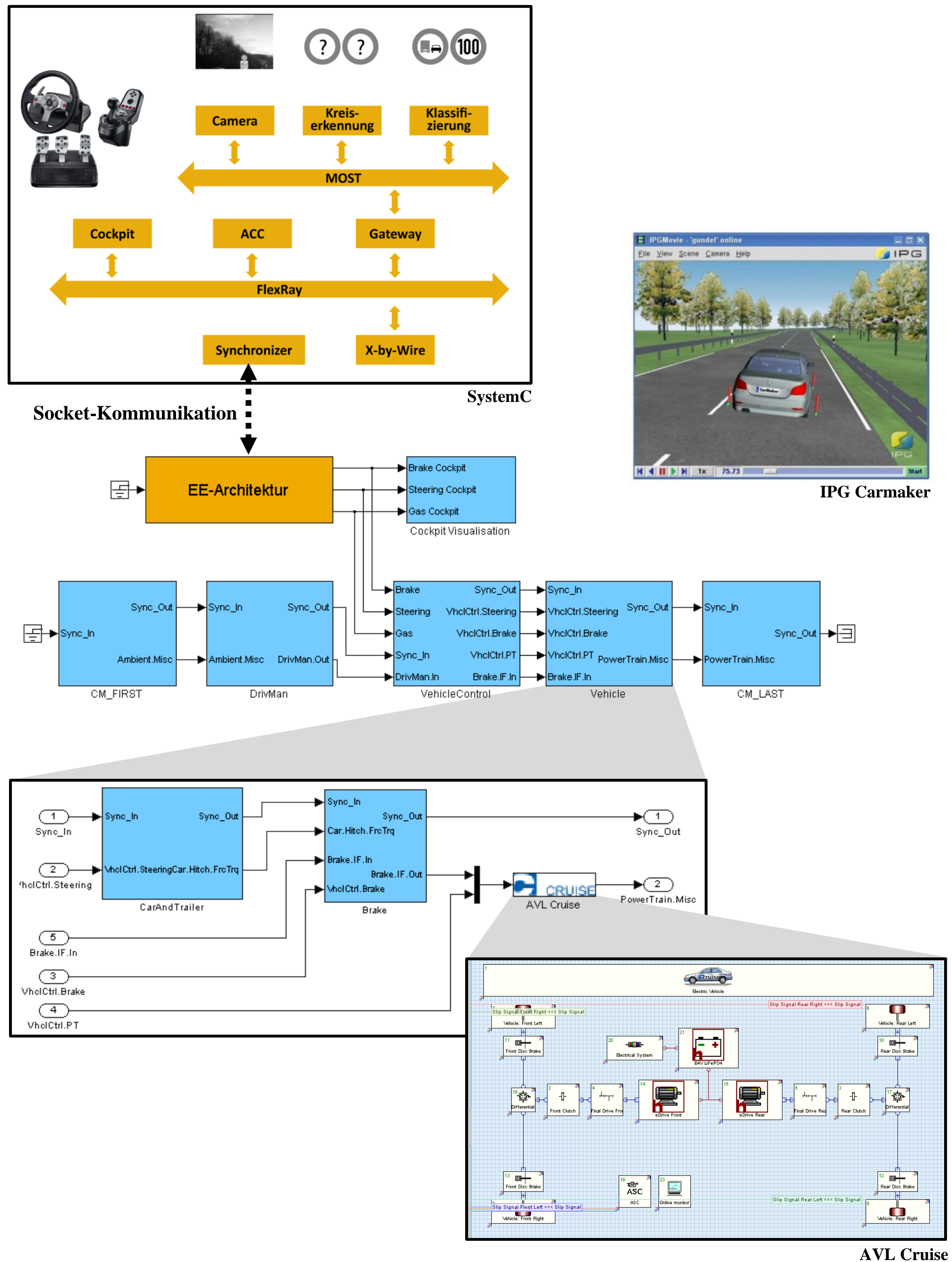


Abbildung 7.1: Generische Co-Simulationsplattform

durch einfache Differenzbildung von nächster Simulationsschrittweite und aktueller Simulationszeit bestimmt werden. Der Vergleich mit den aktuellen Simulationszeiten der zeitgesteuerten Simulationsumgebungen ermöglicht die Bestimmung des nächsten auszuführenden Simulationsprozesses. Für die zweifelsfreie Vorhersage des nächsten Simulationsschritts ist zumindest in der aktuell verwendeten SystemC-Version 2.2 eine Erweiterung des ereignisbasierten Simulationskerns notwendig, da die Liste der abzuarbeitenden Prozesse innerhalb eines Simulationsschritts zuerst von denjenigen Prozessen bereinigt werden muss, deren Aktivierung innerhalb des betrachteten δ-Zyklus liegt.

Erweiterung des ereignisbasierten Simulationskerns Für eine entsprechende Erweiterung des ereignisbasierten Simulationskerns ist zuerst eine genaue Analyse des Ablaufs im SystemC-Simulationskerns notwendig, der die folgenden Schritte umfasst:

(1) Führe alle registrierten Prozesse in zufälliger Reihenfolge und unter Bewahrung der Konsistenz der Nebenläufigkeit aus (**Initialisierungsphase**).

(2) Wähle einen Prozess aus der Liste der ablaufbereiten Prozesse und führe ihn aus. Es kann dabei vorkommen, dass diese Prozesse in dieselbe Liste eingetragen werden (**Evaluierungsphase**).

(3) Springe zu Schritt 2, falls sich weitere Prozesse in der Liste der ablaufbereiten Prozesse befinden.

(4) Führe alle Aktualisierungen aus, die durch Prozesse in Schritt 2 entstanden sind (**Update-Phase**).

(5) Falls im aktuellen δ-Zyklus ein Ereignis ausgelöst und ein anderer Prozess dadurch aktiv wird, füge diesen in die Liste der ablaufbereiten Prozesse ein und springe zu Schritt 2.

(6) Da der aktuelle Simulationszyklus beendet ist, lasse die Simulationszeit bis auf den Zeitpunkt des zeitlich nächsten Ereignisses voranschreiten.

(7) Lege die dann ablaufbereiten Prozesse in die Liste und fahre mit Schritt 2 fort.

Anhand dieses Ablaufs ist ersichtlich, dass aufgrund eventuell vorkommender δ-Zyklen nicht direkt nach Abarbeitung der ablaufbereiten Prozesse die nächste Aktivierung festgestellt werden kann, da in der Update-Phase Ereignisse ausgelöst werden können, die in einer Aktivierung anderer Prozesse resultieren. Diese könnten wiederum einen nächsten Simulationsschritt definieren, der ungleich dem ursprünglichen ist. Dies wird durch Einführung eines spezifischen Synchronisationsprozesses verhindert, der sich selbst solange für den nächsten δ-Zyklus aktiviert, bis nur noch dieser Synchronisationsprozess in der Liste der abzuarbeitenden Prozesse und der durch Ereignisse im aktuellen und nächsten δ-Zyklus aktivierten Prozesse befindet. Ist das der Fall, kann die nächste Simulationszeit eindeutig bestimmt und die Synchronisation basierend auf diesem Zeitschritt durchgeführt werden. Eine benötigte Verzögerung des spezifischen Synchronisationsprozesses um jeweils einen δ-Zyklus wird durch Aufruf der SystemC-Methode `wait()` und einem Parameter `SC_ZERO_TIME` erreicht.

7.2.1.3 Ablauf der Co-Simulation

Der genaue Ablauf der Co-Simulation ist in Abbildung 7.2 dargestellt, wobei Anforderungen von Werten aus anderen Simulationsumgebungen mit „S“ und der Empfang dieser Werte mit „E“ gekennzeichnet sind. Zu berücksichtigen ist hierbei, dass aus Übersichtsgründen die direkte Verbindung zwischen SystemC und MATLAB/Simulink bzw. einer darin enthaltenen S-Function dargestellt ist. Sind mehrere unabhängige zeitgesteuerte Simulationsumgebungen vorhanden, so entstehen zusätzliche Abhängigkeiten.

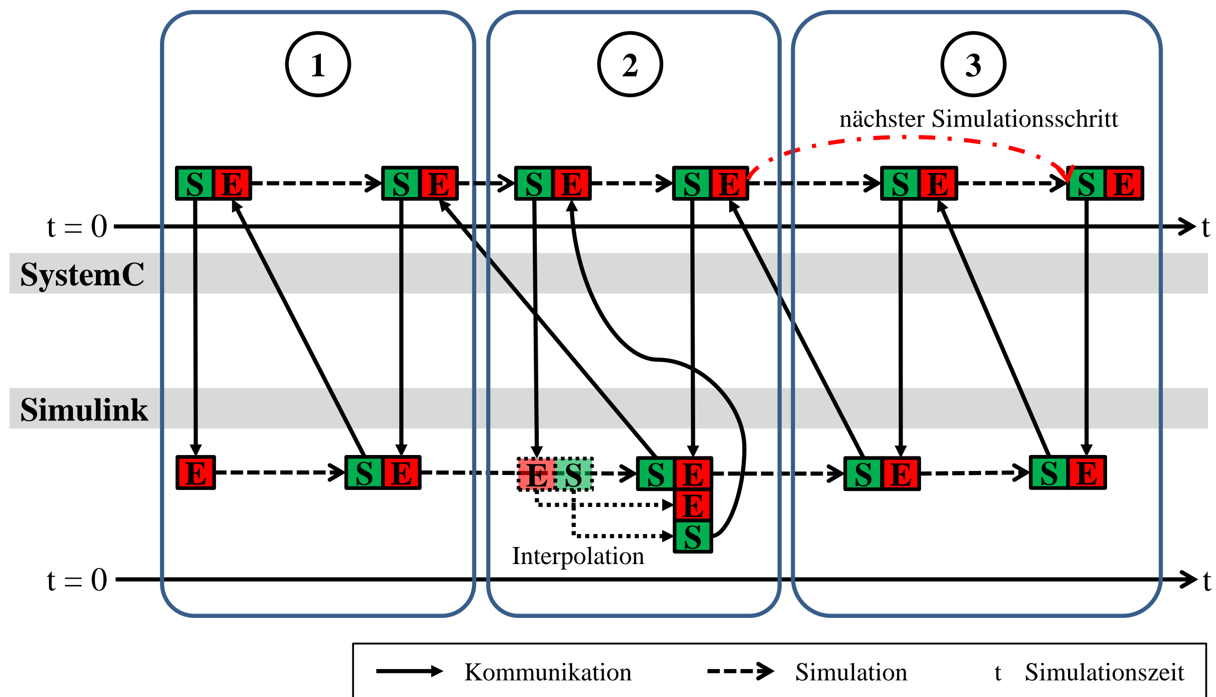


Abbildung 7.2: Ablauf der Co-Simulation und Synchronisation der Simulationskerne

Wie bereits erwähnt, wird eine Synchronisation der einzelnen Simulationen nur durchgeführt, wenn sie notwendig ist, d.h. mehrere Simulationsumgebungen interagieren. Im allgemeinen Fall, der in Abbildung 7.2 durch ① gekennzeichnet ist, wird eine Synchronisation dann durchgeführt, wenn der nächste Simulationszeitpunkt der ereignisbasierten Simulation über die aktuelle Simulationszeit der zeitgesteuerten Simulation hinausgehen würde. Die Simulation kann also auf der ereignisbasierten oder der zeitgesteuerten Simulationsumgebung aus mehreren Simulationsschritten bestehen, ohne dass dazwischen eine Synchronisation stattfinden muss. Dadurch findet eine prinzipielle Entkopplung der Simulationsumgebungen statt, wenn in bestimmten Simulationsschritten keine Interaktion besteht.

Bei einer Interaktion kann es durchaus vorkommen, dass die ereignisbasierte Simulationsumgebung zur Durchführung der eigenen Simulation zum Zeitpunkt t_h auf Simulationswerte zurückgreifen muss, die zwischen zwei Simulationsschritten t_{i-1} und t_i der zeitgesteuerten Simulation liegen. Dieser Fall ist in Abbildung 7.2 mit ② gekennzeichnet. Dann wird der zum Zeitpunkt t_h ($t_{i-1} \leq t_h \leq t_i$) geltende Wert durch lineare Interpolation nach Gleichung 7.1 ermittelt. Hierbei ist zu erwähnen,

dass die Genauigkeit der durch die Interpolation errechneten Werte genügend hoch ist, da wie bereits zuvor erwähnt in zeitgesteuerten Simulationsumgebungen mit variabler Schrittweite der Zeitschritt so gewählt wird, dass die Differenz der Werte in unterschiedlichen Zeitschritten unter einem definierten Schwellwert liegen.

$$wert_{t_h} = wert_{t_{i-1}} + \frac{wert_{t_i} - wert_{t_{i-1}}}{t_i - t_{i-1}} (t_h - t_{i-1}) \tag{7.1}$$

Wird bei einer zeitgesteuerten Simulation ein Ereignis ausgelöst, wie z.B. das Erreichen von definierten Wegpunkten auf einer simulierten Teststrecke oder eine bestimmte Signalisierung der ESP[2]-Sensorik, muss dieses unter Umständen der ereignisbasierten Simulationsumgebung mitgeteilt werden. Der nächste Zeitschritt, der durch einen roten Pfeil gekennzeichnet ist, wird daraufhin von der ursprünglich angenommenen auf die neue Schrittweite modifiziert. Für die Berechnung der aktuellen Simulationszeit bedeutet dies, dass nur die daraus resultierende Differenz in der Zeit voranschreitet und somit ein zusätzlicher Zeitschritt unter Verwendung der SystemC-Methode `wait()` eingeführt wird, was mit Fall ③ in Abbildung 7.2 markiert ist.

7.2.2 Interaktion von eingebettetem System und Umgebung

Durch die in Abschnitt 7.2.1 beschrieben Plattform zur Kopplung domänenspezifischer Simulationsumgebungen lassen sich die einzelnen Systeme unter Berücksichtigung der gegenseitigen Beeinflussung durch verschiedene Domänen simulieren. Im Fall eingebetteter Systeme ist dadurch die Möglichkeiten gegeben, nicht nur die berechneten Ergebnisse und deren Wirkung im umgebenden System zu evaluieren, sondern auch eine *realistische Stimulierung* der Simulation des Hardware/Software-Systems zu erreichen und damit deren Auswirkungen zu analysieren [73]. Somit können ebenfalls Rückschlüsse auf die Leistungsaufnahmen einzelner Systemkomponenten und die damit verbundene Temperaturentwicklung gezogen werden.

Weiterhin erlaubt der Ansatz der ganzheitlichen Systemsimulation einen Explorationsprozess, um Parameter für die E/E-Architektur oder konfigurierbare Strategien zur Energieoptimierung zu bestimmen und die Auswirkungen dieser Exploration aufzuzeigen. Die mit dem Energiemanagement verbundene gezielte Abschaltung elektrischer Verbraucher wird im Automobilbereich auch als *Teilnetzbetrieb* bezeichnet, da eine bestimmte Menge an Verbrauchern, wie z.B. elektronische Steuergeräte[3], logisch gruppiert und – meist in Verbindung mit der Kommunikationsstruktur – zeitweise deaktiviert werden [111] [46].

7.2.2.1 Einfluss der E/E-Architektur auf Regelsysteme

Im realen Fahrzeugbetrieb können auf Kommunikationsbussen dynamische Lasten auftreten, da bestimmte Teilnehmer durch Ereignisse aktiviert und durch deren Kommunikation die Buslasten erhöht werden können. Dies kann abhängig von der Art

[2] Elektronisches Stabilitätsprogramm
[3] Electronic Control Unit (ECU)

der Busarchitektur zu erhöhten Latenzen führen. Sind die nicht-funktionalen Eigenschaften der E/E-Architektur von dynamischen Kontexten abhängig, wie z.B. die Abhängigkeit der Buslatenz von der Benutzung des dynamischen Segments im FlexRay-Kommunikationsprotokoll, können die Auswirkungen meist nur simulativ festgestellt werden.

Im Folgenden wird mithilfe der ganzheitlichen Simulation von Hardware/Software-System, Fahrzeug und Umwelt ermittelt, welche Auswirkungen eine definierte Buslatenz auf ein Fahrzeug haben kann und am Beispiel des ABS[4]-Regelsystems verdeutlicht. Hierzu werden die Regelabweichungen an der Vorderachse betrachtet. Zur Bestimmung der Bremsaktivität wird zuerst ein Schaltkriterium S_K bestimmt. Dabei werden Radumfangsbremsung z_r und relativer Schlupf λ_{rel} jeweils gewichtet und der relative Schlupf aus der Referenzgeschwindigkeit v_{ref} und der Radumfangsgeschwindigkeit v_R wie folgt ermittelt [16]:

$$\lambda_{rel} = \frac{v_{ref} - v_R}{v_{ref}}.$$

Somit ergibt sich für das Schaltkriterium S_K mit den jeweils konstanten, aber abhängig vom Fahrzeug gewählten Gewichtungsfaktoren z_r und K_λ:

$$S_K = K_z \cdot z_r + K_\lambda \cdot \lambda_{rel}.$$

Zur Aktivierung der Bremszylinder des Fahrzeugs durch das ABS existiert ein unteres Schaltkriterium S_{K_u}, zur Deaktivieren ein oberes Schaltkriterium S_{K_o}, wobei gilt:

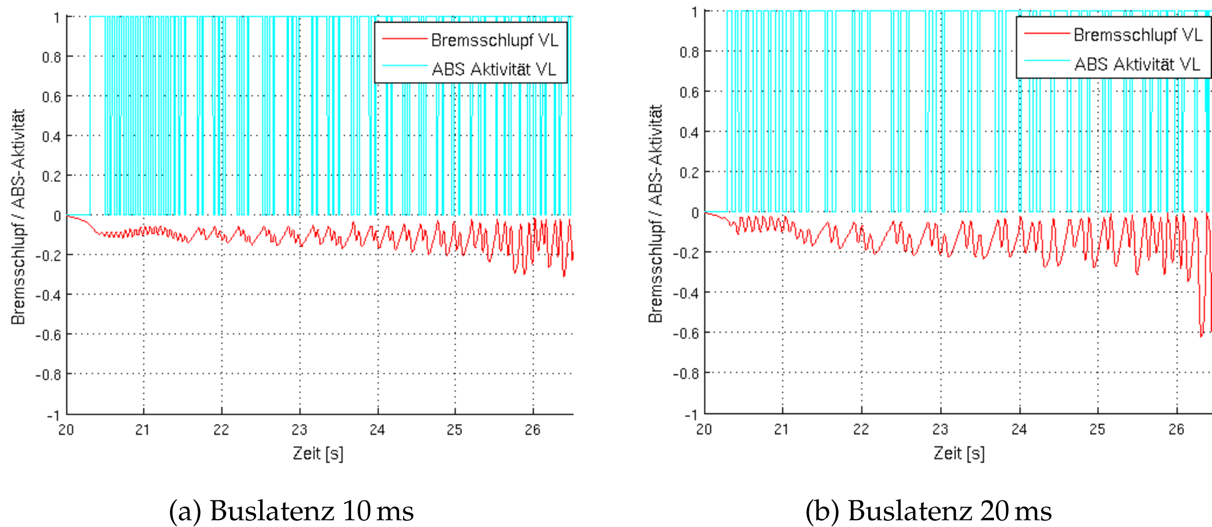
- $S_K < S_{K_u} \rightarrow$ Bremszylinder wird nicht aktiviert
- $S_{K_u} \leq S_K \leq S_{K_o} \rightarrow$ Bremszylinder wird aktiviert
- $S_K > S_{K_o} \rightarrow$ Bremszylinder wird gelöst

Zur Abgrenzung von Bremsschlupf und Antriebsschlupf, also der relativen Differenz zwischen Referenzgeschwindigkeit und Radumfangsgeschwindigkeit während eines Beschleunigungsvorgangs, wird der Bremsschlupf im Folgenden mit negativem Vorzeichen angegeben.

Bei einer Buslatenz von 10 ms und 20 ms ist in Abbildung 7.3 zu erkennen, dass aufgrund der Verzögerung der Regelsignale das ABS häufig regeln muss und deshalb eine hohe Aktivität aufweist. Das ist darin begründet, dass S_{K_u} oder S_{K_o} erreicht wird, aber die Regelung es durch die Buslatenz zu spät erkennt. Dies resultiert in einer eher unsanften Regelung aufgrund der häufigen Betätigung und Lösen des Bremszylinders. Der maximale Bremsschlupf liegt bei ca. -0,3, was einem normalen Wert bei einem starken Abbremsvorgang entspricht, lediglich bei einer Buslatenz von 20 ms ist kurz vor dem Stillstand des Fahrzeugs ein niedrigerer Wert erkennbar.

Wird die Buslast bis zu einer resultierenden Latenz von 60 ms erhöht, ist in Abbildung 7.4 zu sehen, dass der maximale Bremsschlupf bis auf den Wert -0.85 abfällt. Zur

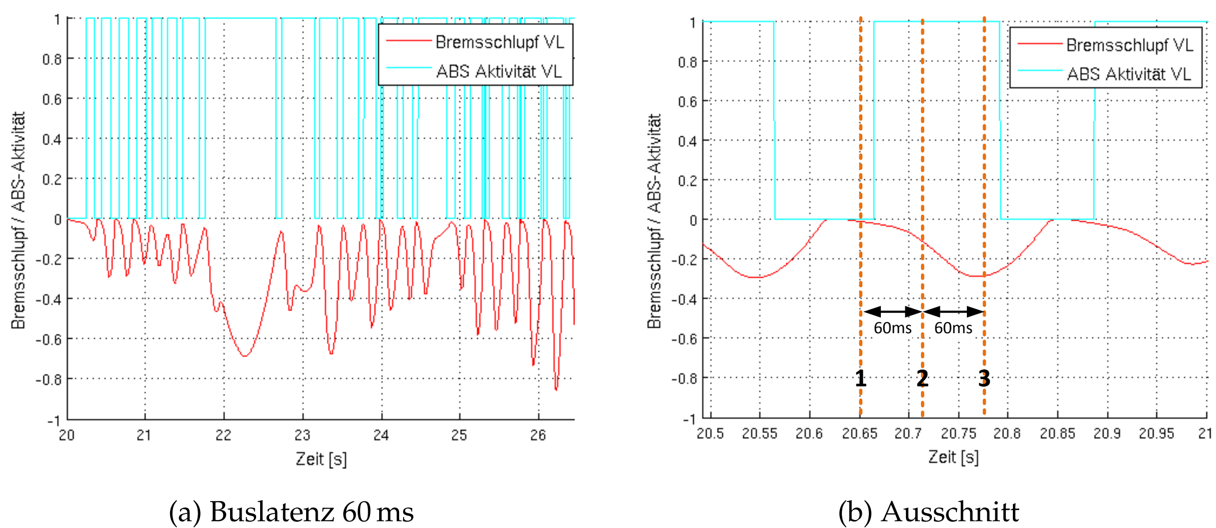
[4]Anti-Blockier-System



(a) Buslatenz 10 ms (b) Buslatenz 20 ms

Abbildung 7.3: Abbremsvorgang mit ABS und unterschiedlichen Buslatenz [73]

genauen Analyse der Ursache ist in Abbildung 7.4b ein Ausschnitt des Abbremsvorgangs dargestellt. Zum Zeitpunkt 1 erkennt das ABS das Überschreiten des unteren Schaltkriteriums S_{K_u}, woraufhin der Bremszylinder aktiviert wird. Bei der nächsten Evaluierung der Daten zum Zeitpunkt 2 wird kein Überschreiten des oberen Schaltkriteriums S_{K_o} festgestellt, sodass keine Aktion vom ABS ausgelöst wird. Zum Zeitpunkt 3 ist jedoch das obere Schaltkriterium schon für eine längere Zeit überschritten, weswegen das ABS den Bremszylinder löst.



(a) Buslatenz 60 ms (b) Ausschnitt

Abbildung 7.4: Abbremsvorgang mit ABS und einer Buslatenz von 60 ms [73]

Die entscheidende Verzögerung zwischen dem Erkennen der Schaltkriterien und der tatsächlichen Betätigung des Aktuators liegt an der Latenz durch den simulierten Kommunikationsbus und lässt das Fahrzeug während des Abbremsvorgangs instabil werden. Die Stabilität des simulierten Fahrzeugs hängt zum Teil jedoch auch von

anderen Faktoren der Fahrzeugdynamik ab, z.B. dem Fahrzeuguntergrund oder der Reifengröße. Nichtsdestotrotz werden die Eigenschaften der simulierten E/E-Architektur in dieser kombinierten Simulation analysierbar, indem sie direkte Auswirkungen auf Eigenschaften des umgebenden Systems hervorrufen. Nachfolgende Schritte zur Lösung, etwa die Modifikation der E/E-Architektur oder die Anpassung der Parameter des ABS-Regelsystems, können somit gezielt durchgeführt werden.

7.2.2.2 Exploration von Konfigurationsparametern

Ausgehend von der Betrachtung der Auswirkungen nicht-funktionaler Eigenschaften auf die Regelsysteme des Fahrzeugs kann ebenfalls eine Exploration der Parameter erfolgen, mit denen die verschiedenen Systeme konfiguriert werden. Unter Verwendung *virtueller Plattformen bzw. virtueller Prototypen* können in kurzer Zeit viele verschiedene Konfigurationen simuliert und bewertet werden. Als Basismodell des eingebetteten Hardware/Software-Systems wird die in Abbildung 7.1 dargestellte E/E-Architektur verwendet, die aus einem *X-By-Wire-System*, also der Übertragung der Steuerinformationen aus dem Cockpit zur elektronisch gesteuerten Aktuatorik für Lenkung, Gas und Bremse, und einem *System zur Erkennung von Verkehrszeichen* besteht. Als Kommunikationsstruktur zur Anbindung dieser Systeme stehen ein FlexRay-Bus und ein MOST[5]-Bus, sowie eine über ein Gateway gekoppelte Kombination aus beiden Bussystemen zur Verfügung. Dadurch lassen sich sowohl die Kommunikationsstrukturen an sich, als auch deren Konfigurationsparameter anhand definierter Metriken bewerten und explorieren.

Tabelle 7.6: Bewertung unterschiedlicher Konfigurationen der Kommunikation

Konfiguration	# Slots	Zykluszeit FlexRay	# Frames/s
# 1 (nur FlexRay-Bus)	343	71,893 ms	13,91
# 2 (nur FlexRay-Bus)	116	24,313 ms	13,71
# 3 (Kombination FlexRay-Bus und MOST-Bus)	2	0,419 ms	98,26

Tabelle 7.6 zeigt einige Bewertungen mit unterschiedlichen Kommunikationsstrukturen und Konfigurationsparametern. In den Konfigurationen 1 und 2 wird jeweils nur ein FlexRay-Bus sowohl zur Übertragung der Daten der Verkehrszeichenerkennung als auch der Daten des X-By-Wire-Systems verwendet. Innerhalb des Systems zur Verkehrszeichenerkennung überträgt die Kamera ein Bild, das eine Auflösung von 320×240 Pixel aufweist und mit einer Pixeltiefe von 8 Bit einer Datenmenge von 76800 Byte entspricht, zum Kreiserkennungsalgorithmus, der wiederum maximal sechs Kreise mit potentiellen runden Verkehrszeichen aus dem Bild extrahiert und diese zu einer Komponente überträgt, die für die Klassifizierung des Inhalts zuständig ist. Werden alle Bilder innerhalb eines FlexRay-Zyklus gesendet, was in der Konfiguration 1 der Fall ist, werden die X-By-Wire-Daten in Abständen von 71,893 ms übertragen, was zu einer zu großen Latenz für die zeitkritischen Daten führt. In Konfiguration 2 wird das Bild der Kamera in drei aufeinanderfolgenden FlexRay-Zyklen übertragen,

[5] Media Oriented System Transport

was die Abstände der Übertragung der X-By-Wire-Daten auf 24,313 ms sinken lässt. Bemerkenswert ist hierbei, dass die mögliche Übertragungsrate der Kamera-Bilder – gemessen in Bildern (engl.: Frames) pro Sekunde – nur sehr geringfügig von 13,91 auf 13,71 sinkt. In Konfiguration 3 werden die Daten-intensiven Kommunikationen des Systems zur Verkehrszeichenerkennung über einen zusätzlichen MOST-Bus übertragen, und nur die klassifizierten Verkehrszeichen über das Gateway und den FlexRay-Bus an das ACC[6]-System des modellierten Fahrzeugs gesendet. Der Zyklus zur Übertragung der X-By-Wire-Daten sinkt dadurch auf 0,419 ms, was sogar die Freiheit gibt, darüber hinausgehend weitere zeitkritische Daten über den FlexRay-Bus übertragen zu können. Die Zahl der pro Sekunde übertragenen Bilder zur Verkehrszeichenerkennung steigt durch den Einsatz der zusätzlichen Kommunikationsstruktur auf nahezu 100 Bilder an.

Dieses Beispiel eines Hardware/Software-Systems, das in die E/E-Architektur eines Fahrzeugmodells eingebettet ist, zeigt die Möglichkeit, anhand der virtuellen Plattform in kurzer Zeit viele verschiedene Konfigurationen zum Zweck der anschließenden Bewertung simulieren zu können. Durch die Interaktion der unterschiedlichen Domänen kann dabei eine Vielzahl von Faktoren berücksichtigt werden, was unter Verwendung eines *realen Prototypen* unmöglich oder nur mit deutlich mehr Aufwand umsetzbar wäre.

7.2.2.3 Temperaturentwicklung auf Steuergeräte/Chip-Ebene

Im Umkehrschluss zur Wirkung des Hardware/Software-Systems auf das umgebende System, was in den Abschnitten 7.2.2.1 und 7.2.2.2 thematisiert wurde, lassen sich durch die vom umgebenden System erzeugte Stimulierung des modellierten Hardware/Software-Systems dessen Eigenschaften evaluieren und optimieren. In diesem Abschnitt wird die Leistungsaufnahme durch die Einbeziehung einer Temperatursimulation auf Steuergeräte- bzw. Chip-Ebene anhand zeitlich bestimmter Leistungsaufnahmen in unterschiedlichen funktionalen Einheiten, wie z.B. Pipeline-Stufen oder Register, verdeutlicht.

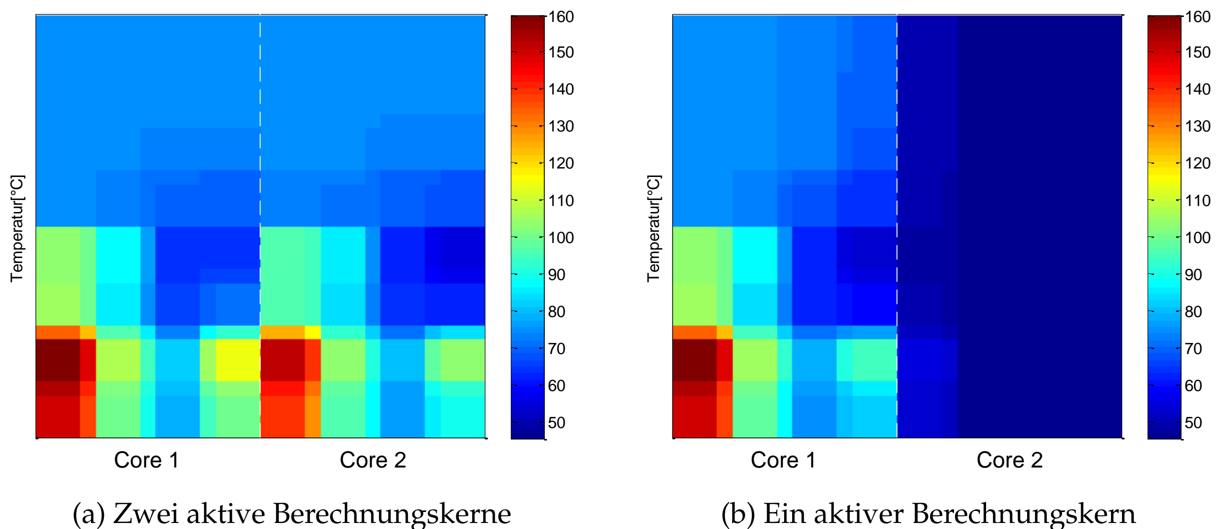
Hierzu wird auf einem virtuellen Hardware/Software-System eine Applikation aus dem Bereich der Kamera-basierten Fahrerassistenzsysteme in Form einer Verkehrszeichenerkennung und abhängig von der Fahrzeugumgebung simuliert. Das Hardware/Software-System wird unter Verwendung der *virtuellen Ausführungsplattform (VEP)* aus Abschnitt 5.2 mit annotierten Ausführungszeiten und der Leistungsaufnahme auf Basisblockebene simuliert. Die Ausführung von dementsprechend instrumentiertem Simulations-Quellcode auf der VEP unter Berücksichtigung einer bestimmten Ablaufstrategie wurde bereits in Abschnitts 5.4.1 vorgestellt. Innerhalb der VEP wird eine Kombination aus instruktionsbasiertem und zustandsbasiertem Leistungsmodell verwendet, um die Leistungsaufnahme abzubilden. Somit ist die Leistungsaufnahme der funktionalen Einheiten der zugrunde liegenden Hardware-Plattform, deren Werte vom Leistungsmodell aus [107] stammen, zwar an den Basisblöcken annotiert, basiert aber auf einem Referenz-Betriebsmodus. Sie werden deshalb abhängig vom aktuellen Betriebsmodus skaliert verwendet (vgl. dazu Abschnitt 4.1.5), um verschie-

[6]Adaptive Cruise Control

dene Betriebsmodi der Hardware-Plattform abbilden zu können. Die Auswahl des aktuellen Betriebsmodus ist abhängig von der Strategie des dynamischen Powermanagements bzw. des Energiemanagements, dessen Auswirkung auf die VEP bereits in Abschnitt 5.4.2 gezeigt wurde. Für die in diesem Abschnitt dargestellte Fallstudie wird ein ARM-basiertes System-on-Chip (SoC) mit 2 Berechnungskernen als ausführende Hardware-Plattform verwendet.

Die Temperaturverteilung auf Chip-Ebene wird mit einem frei verfügbaren Analysewerkzeug [135] durchgeführt, das anhand der Leistungsaufnahme und der Zeitspanne seit der letzten Evaluierung in einem iterativen Analyseprozess die Temperatur basierend auf den Platzierungseigenschaften der funktionalen Einheiten der Hardware-Plattform zu einem oder mehreren spezifizierten Evaluierungszeitpunkten berechnet. Die VEP ruft hierfür das Analysewerkzeug periodisch auf und übergibt die in dieser Periode ermittelte Leistungsaufnahme pro funktionaler Einheit. Als Ergebnis liefert das Analysewerkzeug die zeitliche und räumliche Temperaturverteilung auf der Hardware-Plattform in einer spezifizierbaren Auflösung, wobei ebenfalls Materialparameter und die Konvektion, z.B. durch eine aktive Kühlung, festgelegt werden können.

Bei den nachfolgenden Ergebnissen ist zu beachten, dass die Berechnung der Temperatur in diesem Fall unter Berücksichtigung passiver Konvektion durchgeführt und die Umgebungstemperatur auf 318.5 K (45 °C) festgelegt wird.



(a) Zwei aktive Berechnungskerne

(b) Ein aktiver Berechnungskern

Abbildung 7.5: Temperaturentwicklung auf Chip-Ebene [73]

Abbildung 7.5a zeigt die Temperaturverteilung zu einem bestimmten Zeitpunkt bei Ausführung der Applikation auf beiden Berechnungskernen gleichzeitig und bei ähnlicher Last. Die resultierende Maximaltemperatur beträgt dabei 435 K (162 °C). Aufgrund der gleichmäßigen Belastung kommt es dabei zu keinen nennenswerten Temperaturschwankungen während der Laufzeit. Somit ergibt sich in diesem Szenario eine Temperaturdifferenz zwischen Umgebungstemperatur und Maximaltemperatur von 116 K.

Variiert die Last abhängig vom Szenario, kann die Berechnung auf nur einem Kern

durchgeführt werden. Das Resultat dieser Ausführung wird anhand der Temperaturverteilung in Abbildung 7.5b visualisiert. Da nur ein Berechnungskern aktiv ist, tritt die Maximaltemperatur von 435 K (162 °C) nur auf diesem auf, wobei im Randbereich ebenfalls Auswirkungen der Temperaturentwicklung auf den zweiten Berechnungskern zu sehen sind. Trotz passivem Zustand erfährt dieser eine maximale absolute Temperatur von 328 K (55 °C), was einer Fremderwärmung von 10 K im Vergleich zur Umgebungstemperatur entspricht.

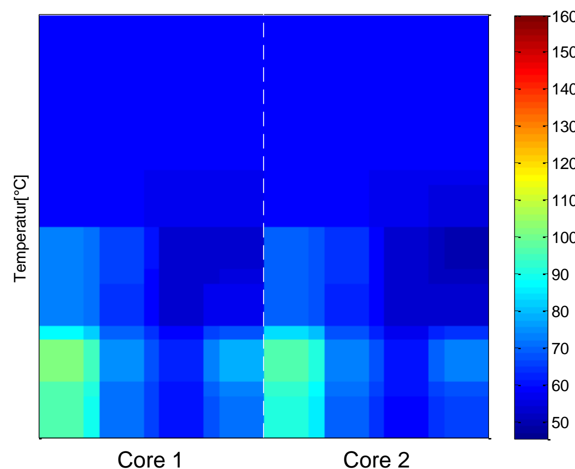


Abbildung 7.6: Temperaturentwicklung auf Chip-Ebene mit Energiemanagement

Wird bei reduzierter Last die Berechnung der Beispiel-Applikation auf beide Berechnungskerne verteilt, kann der Betriebsmodus an die resultierende Durchsatzanforderung pro Berechnungskern angepasst werden. Durch diese Anpassung reduziert sich die Leistungsaufnahme abhängig von Versorgungsspannung und Ausführungsfrequenz, was direkten Einfluss auf die Temperaturverteilung hat. Wie in Abbildung 7.6 dargestellt, reduziert sich die vorkommende maximale Temperatur auf 375 K (101 °C), die resultierende Temperaturdifferenz zur Umgebungstemperatur beträgt somit 56 K.

Anhand der eben beschriebenen Temperaturentwicklung auf Steuergeräte- bzw. Chip-Ebene, die unter Berücksichtigung der Leistungsaufnahme durchgeführt wird, kann gezeigt werden, dass durch die Verwendung der VEP bei der Simulation eingebetteter Hardware/Software-Systeme und die dadurch erzielte Integration einer Ablaufstrategie sowie eines dynamischen Powermanagements dessen Auswirkungen auf die nicht-funktionalen Eigenschaften wie das Zeitverhalten und die Leistungsaufnahme aufgezeigt werden können. Insbesondere können dadurch

- funktionale und nicht-funktionale Eigenschaften analysiert,
- Anforderungen an die nicht-funktionalen Eigenschaften überprüft,
- etwaige Budgets für nicht-funktionale Eigenschaften definiert und
- die Auswirkungen eines Energie- bzw. Powermanagements aufgezeigt werden.

So führt die Anwendung eines Energie- bzw. Powermanagement wie am Simulationsergebnis in Abbildung 7.6 erkennbar zu einer deutlichen Reduzierung der Leistungsaufnahme des virtuellen Modells.

Weiterführend können auf den Temperaturverteilungen basierend zusätzliche applikationsspezifische Analysen hinsichtlich der Zuverlässigkeit und Robustheit der zugrunde liegenden Hardware-Plattform durchgeführt werden, wie z.B. durch Anwendung des Norriz-Landsberg-Modells oder des Coffin-Manson-Modells. Diese sind jedoch nicht Teil dieser Arbeit und werden deshalb hier nicht weiter verfolgt.

7.3 Energiemanagement auf dem Intel SCC

In diesem Abschnitt des Anwendungskapitels sollen die im Kapitel 6 entwickelten Optimierungsalgorithmen auf einer realen Hardware-Architektur mit mehreren verfügbaren Berechnungskernen auf ein komplexes Fahrerassistenzsystem angewandt werden. Als Konsequenz des in Abbildung 4.1 dargestellten Konzepts und des in unterschiedliche Phasen eingeteilten Vorgehens ist dieser Abschnitt in eine Optimierungsphase zur Entwurfszeit und eine anschließende Optimierung während der Laufzeit untergliedert. Ergebnisse der Optimierungsalgorithmen aufgrund synthetischer Experimente und Testanwendungen sind bereits in den jeweiligen Abschnitten 6.1 und 6.2 des Optimierungskapitels enthalten.

Als zugrunde liegende Hardware-Plattform dient die in Abschnitt 2.5.1 eingeführte SCC-Architektur des Herstellers Intel®. Diese Hardware-Plattform bietet sich deshalb an, da sie sowohl über eine große Anzahl an Berechnungskernen verfügt, was Untersuchungen hinsichtlich der Parallelisierung von Anwendungen möglich macht, als auch über ein DVFS-Powermanagement verfügt. Dieses Powermanagement, das die Änderung von Frequenz und Versorgungsspannung unterstützt, kann über eine Programmierschnittstelle aus einer Anwendung heraus geregelt werden. Die Architektur verfügt über insgesamt 48 Kerne gleichen Typs, die über ein gitterförmiges Kommunikationsnetzwerk miteinander verbunden sind. Für die Regelung des Betriebsmodus der Berechnungskerne ist die Architektur in unterschiedliche Bereiche eingeteilt, in denen unabhängig voneinander verschiedenen Betriebsmodi ausgeführt werden können. Die Architektur ist zunächst in sechs Bereiche zu je acht Kernen eingeteilt, in denen die Versorgungsspannung geregelt werden kann, weswegen ein solcher Bereich auch als *Voltage Island* bezeichnet wird. Abbildung 7.7 zeigt die Einteilung in verschiedene Bereiche der SCC-Architektur.

Im Gegensatz zur Versorgungsspannung, die nur in der Granularität der Voltage Islands separat geregelt werden kann, kann die Schaltfrequenz für jeden Tile angepasst werden. Ein Tile bildet somit eine *Frequency Island*. Die maximal mögliche Frequenz hängt technologisch-bedingt von der Versorgungsspannung der Voltage Island ab, in der sich der Tile befindet. Wegen dieser Abhängigkeit und der gröberen Granularität der Einteilung in Voltage Islands werden diese auch als *Power Domains* bezeichnet.

Insgesamt stehen auf der SCC-Architektur 15 verschiedene Betriebsmodi zur Verfügung, wobei einige von diesen jeweils dieselbe Versorgungsspannung verwenden, um einen stabilen Zustand zu garantieren. Die im Folgenden verwendete Benennung der Betriebsmodi richtet sich nach der Einstellung des Frequenzteilers, der die aktuelle Taktfrequenz des SCC relativ zur Referenzfrequenz von 1,6 GHz bestimmt. Tabelle 7.7

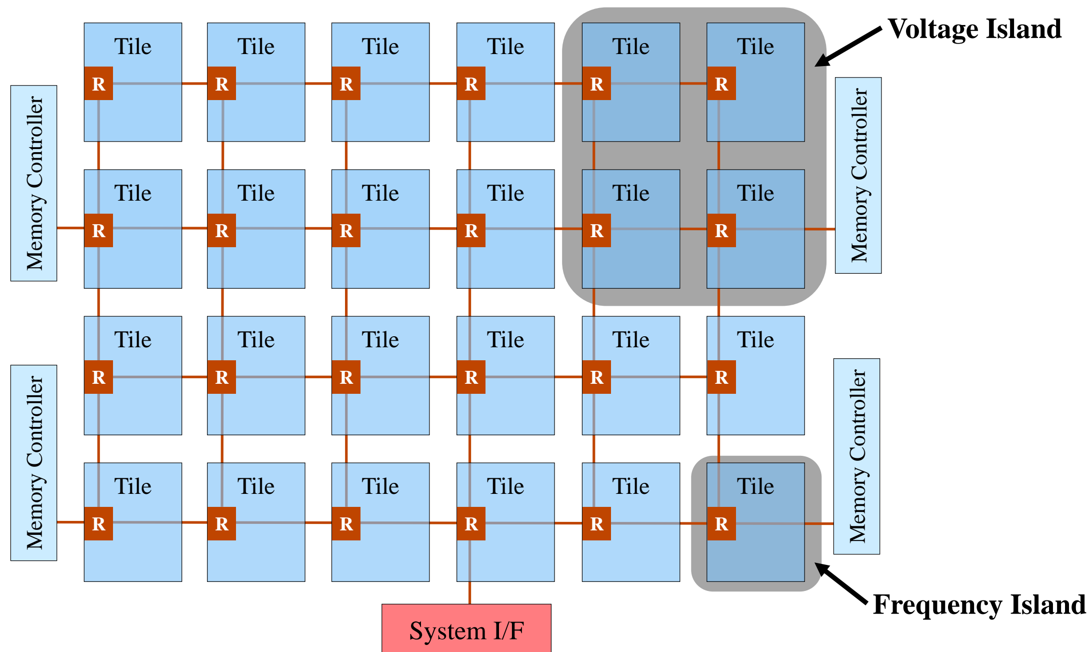


Abbildung 7.7: Einteilung der Power-Domains in SCC-Architektur (Bildquelle: Intel)

zeigt die maximale Frequenz und somit die zur Verfügung stehenden Betriebsmodi in Abhängigkeit der Versorgungsspannung.

Tabelle 7.7: Maximale Frequenz abhängig von Versorgungsspannung

Versorgungsspannung	Maximale Frequenz	Frequenzteiler/Betriebsmodus
1,1 V	800 MHz	2
0,8 V	533 MHz	3
	400 MHz	4
	320 MHz	5
	266 MHz	6
	228 MHz	7
	200 MHz	8
	178 MHz	9
0,7 V	160 MHz	10
	145 MHz	11
	133 MHz	12
	123 MHz	13
	114 MHz	14
	106 MHz	15
	100 MHz	16

Als auszuführende Anwendung aus dem Bereich der Fahrerassistenzsystem in der

Anwendungsdomäne Automobil dient ein System zur Kamera-basierten Erkennung von Verkehrszeichen, das bereits in Abschnitt 7.2.2.2 verwendet und vorgestellt wurde. Diese Anwendung wird auf mehrere Berechnungskerne verteilt, um die an das System gestellten Performanzanforderungen erfüllen zu können. Abbildung 7.8 zeigt die komplette Anwendung in datenflussorientierter Notation. Anhand dieser Anwendung sollen die entwickelten Methoden zur Optimierung der Energieeffizienz in beiden Phasen – zur Entwurfszeit und zur Laufzeit – demonstriert werden.

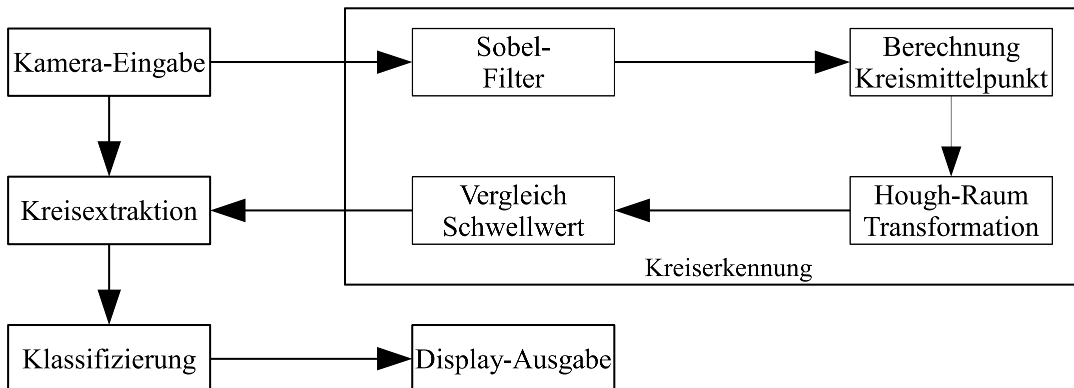


Abbildung 7.8: Anwendung zur Kamera-basierten Erkennung von Verkehrszeichen

Da die Optimierungsalgorithmen neben den Ausführungszeiten der Applikationen bzw. Tasks in den jeweiligen Betriebsmodi auch die Umschaltzeiten zwischen unterschiedlichen Betriebsmodi berücksichtigen, werden zunächst die für die Durchführung eines Wechsels benötigen Umschaltzeiten analysiert. Um dynamische Laufzeiteffekte auszuschließen wird eine spezielle Testanwendungen zum permanenten Wechseln des Betriebsmodus oftmals hintereinander ausgeführt und die entsprechenden Durchschnittswerte ermittelt. Diese Werte sind für die in diesem Abschnitt relevanten Betriebsmodi in Tabelle 7.8 aufgelistet.

Tabelle 7.8: Analysierte Zeiten zum Wechseln des Betriebsmodus auf dem Intel SCC

Aktueller Betriebsmodus	Ziel-Betriebsmodus	Zeitaufwand für Wechsel
2	6	39,729 ms
2	16	40,351 ms
6	2	40,322 ms
6	16	40,360 ms
16	2	40,355 ms
16	6	40,362 ms

Die SCC-Architektur bietet die Möglichkeit, von jedem Betriebsmodus in jeden anderen zu wechseln. Für die Umsetzung dieser Eigenschaft in ein zustandsbasiertes Leistungsmodell bedeutet dies, dass die resultierende *Power State Machine* einen Übergang von jedem Zustand zu jedem anderen aufweist und deswegen einem

vollvermaschten Zustandsgraphen entspricht. Sie ist somit trivial und wird deswegen hier nicht explizit dargestellt.

7.3.1 Optimierung zur Entwurfszeit

Die inhärenten Schwierigkeiten bei der Lösung des nichtlinearen Optimierungsproblems zur Minimierung des Energieverbrauchs wurden bereits in Abschnitt 6.1.3 erörtert. Zum einen ist das Optimierungsergebnis von der Wahl der Startwerte für die Lösungsvariablen abhängig, zum anderen besteht die Möglichkeit, in einem lokalen Minimum zu enden. Es kann also bei der Terminierung des Optimierungsprozesses keine Aussage über die Optimalität des Ergebnisses getroffen werden. Aufgrund dieser experimentellen Ergebnisse wird die nichtlineare Formulierung durch eine geeignete Abstraktion in ein lineares Optimierungsproblem transformiert, wodurch lokale Optima immer auch dem *globalen Optimum* entsprechen.

Dazu werden die mathematischen Formulierungen aus Abschnitt 6.1.1.1 verwendet und wie in Abschnitt 6.1.4.1 beschrieben durch Einführung zusätzlicher Lösungsvariablen zu einem linearen Optimierungsproblems abstrahiert. Dadurch ist es allerdings nicht mehr möglich, die genauen Kosten für etwaige Wechsel der Betriebsmodi in der Kostenfunktion zu berechnen, da keine Möglichkeit besteht, eine direkte Vorgänger-Nachfolger-Beziehung zu bestimmen. Dadurch werden zwar alle potentiellen Wechsel in die Kostenfunktion aufgenommen, diese werden aber gleichzeitig auch minimiert. Das Gesamtergebnis der Kostenfunktion muss deswegen in einem separaten Schritt um die nicht wirklich stattfindenden Wechsel bereinigt werden. Der Einfluss dieser durch Abstraktion eingeführten Ungenauigkeit nimmt mit fallenden Wechselkosten ab.

Zur Lösung des abstrahierten linearen Optimierungsproblems wird das MATLAB-basierte Werkzeug *TomOpt* [127] angewandt. Dieses erlaubt mit der Werkzeugerweiterung *TomSym* die symbolische Spezifikation von Lösungsvektoren, Variablen und Matrizen für Gleichheits- bzw. Ungleichheitsbedingungen, sowie die automatische Auswahl eines geeigneten Lösungsalgorithmus abhängig von einer zuvor durchgeführten Komplexitätsanalyse der Problemformulierung.

Die Formulierung des ganzzahligen linearen Optimierungsproblems in TomSym findet sich in Quellcode 7.1 an einem Beispiel mit 20 Applikationen und acht zur Verfügung stehenden Verarbeitungsressourcen mit jeweils drei Betriebsmodi. Die dynamische Leistungsaufnahme beträgt 1,74 bzw. 0,63 bzw. 0,3 Leistungseinheiten pro Zeiteinheit, abhängig vom jeweiligen Betriebsmodus. Die maximale Periode ist in diesem Beispiel auf 500 Zeiteinheiten beschränkt. Ab Zeile 8 werden die zu lösenden Unbekannten definiert, wobei die einzelnen Dimensionen der Matrizen zur späteren Benutzung benannt werden können. Zeile 21 definiert somit die Zuordnung der Applikationen auf Ressourcen. In Zeile 22 werden Vektoren mit den paarweisen Kombinationen aller Applikationen angelegt. Anhand dieser Vektoren können durch die zu geltenden Nebenbedingungen die unbekannten Variablen festgelegt werden. In Zeile 37 wird z.B. definiert, dass zwischen dem Startzeitpunkt von Applikation j und dem Endzeitpunkt von Applikation i mindestens die Zeit für einen Wechsel des Betriebsmodus zwischen i und j liegen muss, wenn i vor j ausgeführt wird und diese

auf dieselbe Ressource abgebildet sind. In Zeile 24 wird der Energieverbrauch aufgrund dynamischer Verlustleistung berechnet, der als Summand in der in Zeile 29 enthaltenen Zielfunktion verwendet wird. Ab Zeile 32 werden die Nebenbedingungen definiert und somit ungültige Belegungen der unbekannten Variablen verhindert. In Zeile 45 werden schließlich sowohl die Zielfunktion, als auch die Nebenbedingungen an den Lösungsalgorithmus übergeben und dieser ausgeführt.

Nach [13] haben Untersuchungen zur Parallelisierung einzelner Teile des Systems zur Verkehrszeichenerkennung auf der SCC-Architektur ergeben, dass das Potential des enthaltenen Aufgabenparallelismus für eine verteilte Ausführung des Algorithmus zur Kreiserkennung auf drei Berechnungskernen in Abwägung der funktionalen Anforderungen und der Beschleunigung der Ausführung genutzt werden kann. Dazu wurde die Anwendung mehrmals mit unterschiedlicher Konfiguration auf der SCC-Architektur ausgeführt und die resultierende Ausführungszeit in Relation zur Fahrgeschwindigkeit und dem vorgeschriebenen Abstand zwischen Verkehrszeichen gesetzt.

Zur Demonstration der Optimierung der Energieeffizienz zur Entwurfszeit an einem komplexen Beispiel werden neben den Algorithmen des Fahrerassistenzsystems noch weitere beliebige Applikationen auf die Hardware-Plattform abgebildet. Diese werden zur bestehenden Plattform hinzugefügt und verändern dadurch die Zuordnung einzelner Applikationen zu Verarbeitungsressourcen und Betriebsmodi. Die Ergebnisse der Optimierung sind in Abbildung 7.9 für mehrere Beispielszenarien dargestellt. Die Spezifizierung der Architekturparameter, z.B. der Leistungsaufnahme, sind teilweise aus [51] entnommen, die Periode beträgt jeweils 1000 ms.

In Algorithmus 7.1 ist dargestellt, wie die Ausführungszeit der zusätzlich auf die Hardware-Plattform abgebildeten Applikationen berechnet wird.

Algorithmus 7.1 Erstellung von synthetischen Applikationen

function erzeugeApplikationen()
A Applikationen
R Ressourcen
t_{max} Maximale Periode
for all $r \in R$ **do**
  $resFree[r] \leftarrow t_{max}$
end for
for all $a \in A$ **do**
  $[maxFreeResValue, index] \leftarrow max(resFree)$
  $limit \leftarrow min(t_{max}, maxFreeResValue)$
  $t_{upperBound} \leftarrow limit/10 * 8$
  $t_{lowerBound} \leftarrow limit/10 * 2$
  $executionTime[a] \leftarrow t_{lowerBound} + (t_{upperBound} - t_{lowerBound}) * rand()$
  $resFree[index] \leftarrow resFree[index] - executionTime[a]$
end for

Die letztendlich zu erzielenden Ergebnisse bei der Optimierung der Energieeffizienz hängen maßgeblich von den an die Applikationen gestellten Anforderungen ab und kann deswegen nicht unabhängig beurteilt werden. Iterativ werden deshalb synthetische

```
% Spezifikation der Parameter
Na = 20; % Anzahl der Applikationen
Nr = 8; % Anzahl der Ressourcen
Nm = 3; % Anzahl der Betriebsmodi
Tmax = 500; % Maximale Periode
Npairs = Na*(Na-1)/2; % Anzahl an Applikations-Paaren
P_dyn = repmat([1.74 0.63 0.3],Nr,1); % Dynamische Leistungsaufnahme
...

% Definition der zu lösenden Unbekannten
x = tomArray('x',size(timr),{'a','m','r'},'integer'); %Lösungsvektor
tStart = tom('tStart',Na,1); % Startzeiten der Applikationen
tEnd = tom('tEnd',Na,1); % Endzeiten der Applikationen
gDuration = tom('gDuration',Na,1); % Dauer der Lücke zwischen Applikationen
gOffSwitchRes = tom('gOffSwitchRes',Na,Nr,'integer'); % DPM-fähige
    Ressource wird ausgeschaltet
sameResource = tom('sameResource',Npairs,1,'integer'); % Applikation i und
    j sind auf dieselbe Ressource abgebildet
isBefore = tom('isBefore',Npairs,1,'integer'); % Applikation i vor j
isAfter = tom('isAfter',Npairs,1,'integer'); % Applikation i nach j
tSwitch = tom('tswitch',Npairs,1); % Zeit für PM-Wechsel zw. i und j
...
ar = sum(x,'m'); % Zuordnung Applikation zu Ressource
[i,j] = find(tril(ones(Na,Na),-1)); % Kombination aller Applikationen
...
E_dyn = sum(vec(P_dyn.*sum(x.*timr,'t')));
E_stat = ...
...

% Zielfunktion
objective = E_dyn + E_stat + E_modeSwitching + E_dev;

% Nebenbedingungen
constraints = {
  0 <= x <= 1; % Eingrenzung Lösungsvektor
  sum(sum(x,'m'),'r') == 1; % Eindeutigkeit der Zuordnung
  0 <= tstart <= Tmax; % Eingrenzung der Startzeiten
  tEnd == tStart + sum(sum(timr.*x,'r'),'m'); % Berechnung Endzeiten
  tStart(j)-tEnd(i)-tSwitch+Tmax*(1-isBefore) >= 0; % Überlappung i vor j
  tStart(i)-tEnd(j)-tSwitch+Tmax*(1-isAfter) >= 0; % Überlappung i nach j
  ar(i,:)+ar(j,:)-repmat(sameResource,1,Nr) <= 1; % Markierung wenn i und
      j auf dieselbe Ressource abgebildet werden
  isBefore+isAfter >= sameResource; % keine Überlappung von Applikationen
      auf derselben Ressource
  ...
}

% Aufruf des Lösungsalgorithmus
solution = ezsolve(objective,constraints);
```

Quellcode 7.1: Formulierung des linearen Optimierungsproblems in TomSym

Applikationen erzeugt und deren Ausführungszeit nach einer virtuellen Abbildung auf die zur Verfügung stehenden Ressourcen durch ein *Bin-Packing*-Verfahren der *Best-Fit*-Strategie gewählt. Die Ausführungszeiten der Applikationen werden gleichverteilt zwischen 20% und 80% der maximal möglichen Ausführungszeit definiert – gemessen an der absoluten zur Verfügung stehenden Ausführungszeit im Betriebsmodus mit der höchsten Leistungsfähigkeit. Allgemein gilt, dass es weniger Potential zur Optimierung gibt, je länger die Ressourcen im leistungsfähigsten Betriebsmodus ausgelastet wären. In Abbildung 7.9 ist die Reduzierung der benötigten Energie durch Anwendung des in Abschnitt 6.1.4 erklärten Optimierungsverfahrens zur Entwurfszeit dargestellt. Dabei wird jede Konfiguration insgesamt zehn Mal für maximal 600 Sekunden optimiert und das durchschnittlich erzielte Ergebnis aufgetragen. Konfigurationen mit einer größeren Ressourcenanzahl als Applikationen sind ausgenommen, ebenso Konfigurationen mit einer Ressourcenanzahl kleiner als sechs bzw. kleiner als zehn, wenn die Anzahl der Applikationen über 14 bzw. 21 liegt.

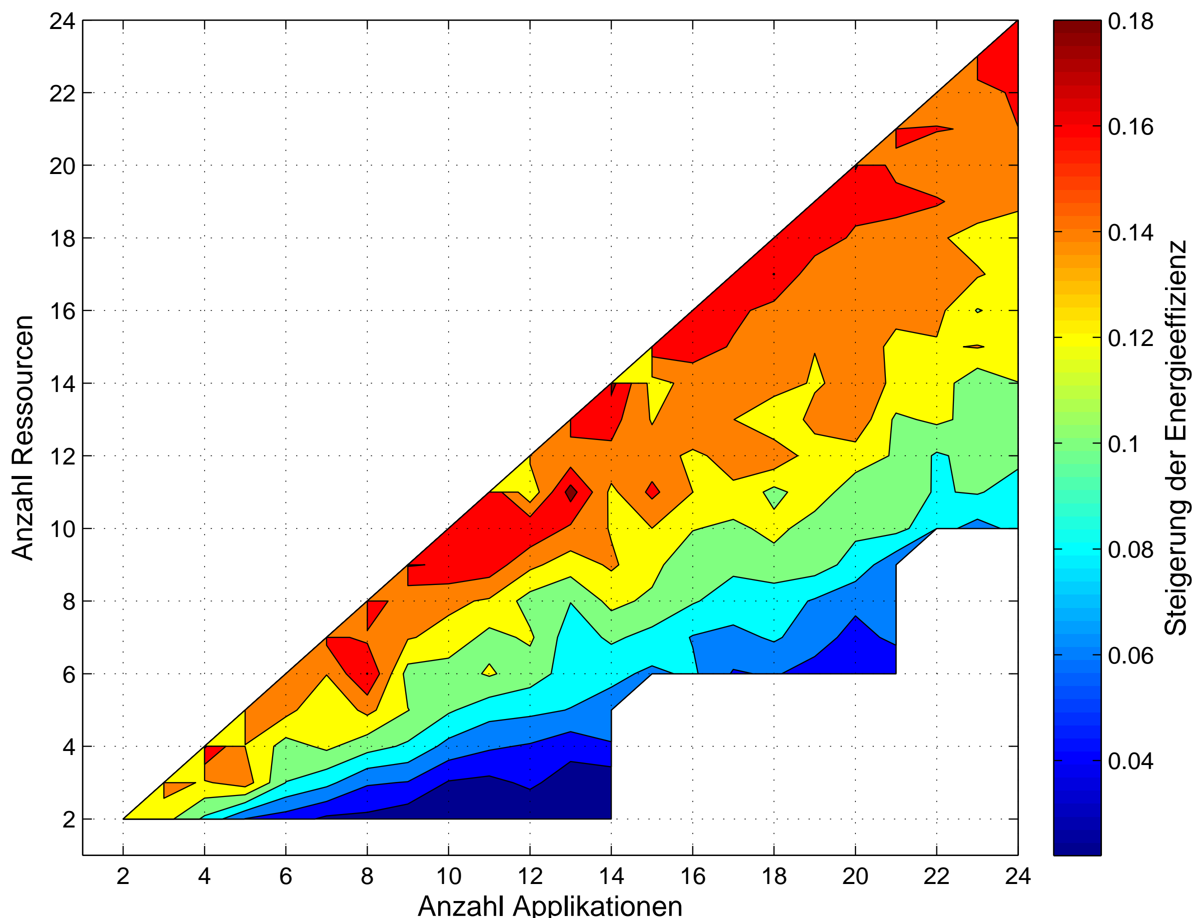


Abbildung 7.9: Reduzierung des Energieverbrauchs in Abhängigkeit verschiedener Konfigurationen von abgebildeten Applikationen und verfügbaren Ressourcen

An den Ergebnissen ist zu erkennen, dass die Effizienzsteigerung dann klein ist, wenn die Anzahl der Applikationen sehr viel größer ist als die Anzahl der Ressourcen. Dies liegt daran, dass durchschnittlich mehr Applikationen auf eine Verarbeitungsressource

abgebildet werden müssen und somit das Potential für Wechsel in einen energieeffizienteren Betriebsmodus abnimmt. Der Grund hierfür liegt sowohl am relativ zur maximalen Periode gesehen großen zeitlichen Aufwand, der für jeden Moduswechsel benötigt wird, als auch an der erheblich reduzierten Frequenz in energieeffizienteren Betriebsmodi der SCC-Architektur. Für Betriebsmodus 6 wird idealisiert die halbe Frequenz, für Betriebsmodus 16 eine um den Faktor 3 reduzierte Frequenz angenommen. Allgemein lässt sich eine nahezu lineare Steigerung der Energieeffizienz mit Zunahme der zur Verfügung stehenden Ressourcen feststellen, die teilweise bis zu 18 % beträgt. Gleichzeitig zeigt sich auch, dass trotz der enormen Komplexität des zu lösenden Optimierungsproblems innerhalb der spezifizierten maximalen Lösungszeit eine beachtliche Steigerung erzielt wird, was in der geeigneten sowie effizienten Formulierung des mathematischen Problems begründet ist.

Die Parametrisierung des Optimierungsalgorithmus zeigt, dass die Klassifizierung als Teil des Algorithmus des Fahrerassistenzsystems, sowie alle synthetischen Applikationen mit der längsten möglichen Ausführungszeit (*WCET*) bewertet werden muss, um auch in diesem Fall alle an das System gestellten Anforderungen erfüllen zu können. Da dies in der Realität zumeist nur in seltenen Situationen der Fall ist, kann die Energieeffizienz zusätzlich während der Laufzeit der Applikationen optimiert werden, indem das Energiemanagement dynamisch an das aktuelle Szenario angepasst wird. Die Anwendung der Optimierung zur Laufzeit wird in Abschnitt 7.3.2 separat anhand der Klassifizierungsapplikation erklärt.

7.3.2 Optimierung zur Laufzeit

Die Optimierung der Energieeffizienz zur Laufzeit soll in diesem Abschnitt anhand deren Anwendung auf den Klassifizierungsalgorithmus, welcher als Teil des Fahrerassistenzsystems in Abbildung 7.8 enthalten ist, demonstriert werden. Die Ausführung dieser Applikation auf der virtuellen Ausführungsplattform legt eine Einteilung in sieben Cluster nahe, welche die unterschiedlichen Ausführungspfade hinsichtlich der benötigten Ausführungszeit kategorisieren. Aus diesen analysierten Ausführungspfaden erfolgt die direkte Ableitung der Applikationsszenarien zur Verwendung im entsprechenden SDF-basierten Berechnungsgraphen. Die Erstellung der Repräsentation dieses ASADF-Graphen wird durch die Verwendung einer domänenspezifischen Sprache (engl.: *Domain-Specific Language (DSL)*) und eines anschließenden Generierungsprozesses automatisiert. Mithilfe einer DSL lassen sich sowohl die abstrakte Syntax und das zugrunde liegende Metamodell als auch die konkrete Syntax definieren, sodass textuelle und teilweise auch grafische Editoren automatisch abgeleitet werden können. Abbildung 7.10 zeigt den resultierenden SDF-Graphen und die Gruppierung in Applikationsszenarien als Zustände – benannt nach der Anzahl der im Bild zu klassifizierenden Kreise – inklusive der möglichen Zustandsübergänge. Ein Zustandswechsel beschreibt also eine Veränderung der Menge enthaltener runder Verkehrszeichen in aufeinanderfolgenden Bildern.

Da es keine Korrelation zwischen der Anzahl der zu klassifizierenden Kreise konsekutiver Bilder der Kamera gibt, sind die Applikationsszenarien unabhängig

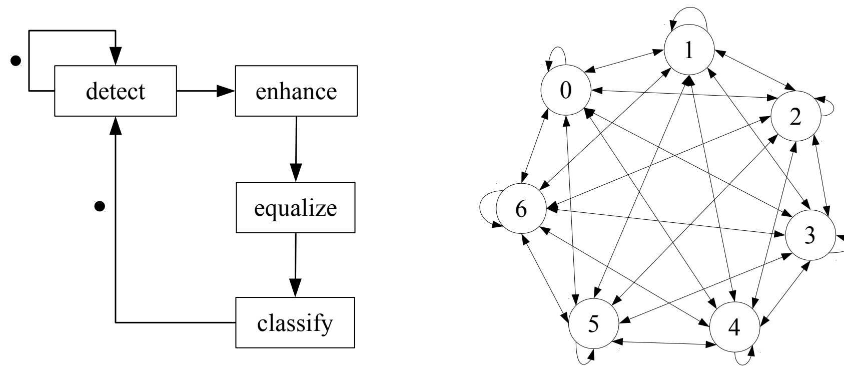


Abbildung 7.10: SDF-Graph und Applikationsszenarien des Klassifizierungsalgorithmus

voneinander, was zu einem vollvermaschten Zustandsgraphen für die Repräsentation der Applikationsszenarien und deren Übergänge führt.

Weiterhin können die Ausführungszeiten der Tasks innerhalb der Klassifizierung entweder durch Simulation bzw. Ausführung dieser Applikation oder durch Spezifikation als WCET ermittelt werden. Diese Ausführungszeiten sind sowohl vom Applikationsszenario, als auch vom Plattformszenario abhängig. Da hier die unterschiedlichen Applikationsszenarien jeweils die Anzahl der zu klassifizierenden Kreise widerspiegeln, kann die Ausführungszeit bezüglich dieser Applikationsszenarien linear zur Kreisanzahl skaliert werden. Die aus dieser Vereinfachung resultierenden Ausführungszeiten pro Kreis sind speziell für die Plattformszenarien bzw. Betriebsmodi der Hardware-Plattform in Tabelle 7.9 dargestellt. Aus Gründen der Komplexität und Übersichtlichkeit sollen in diesem Anwendungsbeispiel nur die Betriebsmodi 2, 6 und 16 verwendet werden. Bei den angegebenen Ausführungszeiten handelt es sich in diesem Fall um Zeitwerte, die auf der Intel SCC-Architektur gemessen wurden. Da sich diese Mehrkern-Architektur noch in einem frühen Entwicklungsstadium befindet, musste die maximale Frequenz in Betriebsmodus 2 zur Verbesserung der Stabilität während der folgenden Experimente von 800 MHz auf 533 MHz gesenkt werden.

Tabelle 7.9: Ausführungszeiten der Klassifizierung pro Kreis abhängig vom Betriebsmodus der Intel SCC-Architektur

Modus	Spannung	Frequenz	Ausführungszeiten Funktionen		
			enhance	equalize	classify
2	1,1 V	533 MHz	5,810 ms	0,362 ms	76,249 ms
6	0,7 V	266 MHz	10,362 ms	0,647 ms	135,968 ms
16	0,7 V	100 MHz	22,121 ms	1,381 ms	290,268 ms

Anhand der Parameter der Hardware-Plattform für den Zeitverbrauch zwischen verschiedenen Betriebsmodi, der Spezifikation des ASADF-Graphen und der Ausführungszeiten der Aktoren in den jeweiligen Szenarien wird der Zustandsraum unter Verwendung des in Abschnitt 6.2 beschriebenen Algorithmus aufgebaut. Mit einer Parametrisierung von λ mit dem Wert 2 und ϵ mit einem Wert von 7 ms ergeben sich nach Anwendung der Regeln zur Zustandsraumlimitierung insgesamt 1664 gültige

Zustände. Dabei existieren 29443 Zustandsübergänge, die einen Zustand in einen anderen überführen. Die Berechnung dieses Zustandsraums dauert auf einem aktuellen Entwicklungsrechner 483 ms. Ungeachtet dieser geringen Ausführungszeit des Algorithmus wird diese Berechnung noch zur Entwurfszeit durchgeführt und besitzt deswegen keine engen zeitlichen Restriktionen. Dieser Zustandsraum wird zur ständigen Auswertung während der Laufzeit in einer speziellen Datenstruktur abgelegt und auf die Zielarchitektur übertragen. Das Metamodell dieser Datenstruktur ist in Abbildung 7.11 als automatisch generiertes Ecore-Metamodell in Eclipse EMF dargestellt.

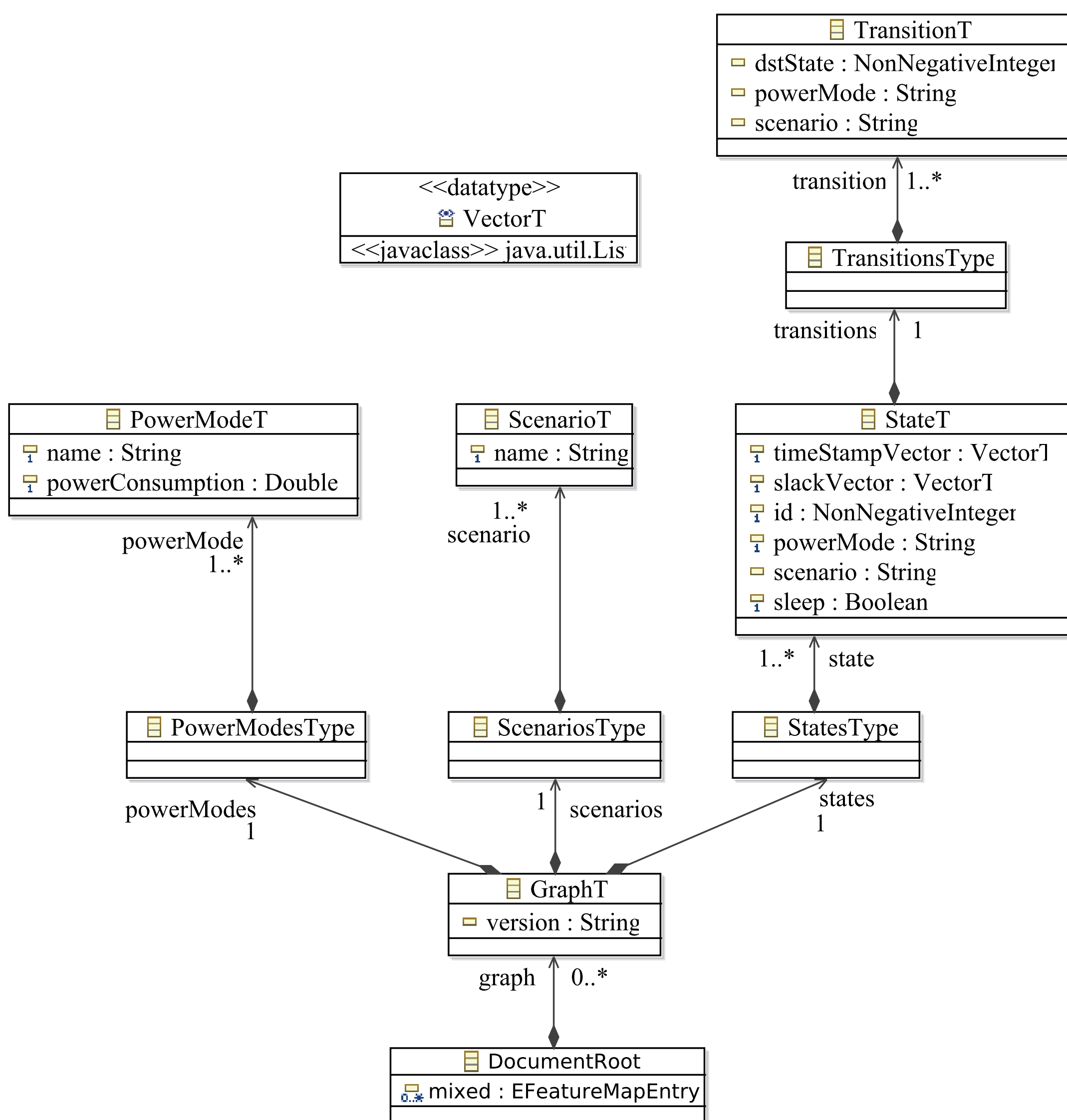


Abbildung 7.11: Metamodell der Datenstruktur zur Speicherung des Zustandsraums

Insgesamt benötigt die gefüllte Datenstruktur auf der SCC-Architektur einen Speicherbedarf von ca. 3 Megabyte. Diese Datenstruktur muss vor dem Ausführen der Applikation einmalig auf der Zielarchitektur geladen werden, um dann zur Laufzeit der Applikation ausgewertet werden zu können. Dieser Ladevorgang hängt von der Größe der Datenstruktur ab, die zur Speicherung des Zustandsraums benutzt

wird, und beträgt bei der Klassifizierungsapplikation 4,58 s. Da der Ladevorgang nur einmal ausgeführt werden muss, wird er als Teil des Bootvorgangs der Zielarchitektur durchgeführt. Die hier angeführten Resultate bezüglich des verwendeten Optimierungsverfahrens sind in Tabelle 7.10 zusammengefasst.

Um den auf der Zielarchitektur abgelegten applikationsspezifischen Zustandsraums zur Laufzeit auswerten zu können, wird eine für die Datenstruktur spezifische Programmschnittstelle verwendet. Diese liefert abhängig vom aktuellen Zustand und dem Applikationsszenario der nächsten Iteration der Applikation den nächsten Zustand, der unter Berücksichtigung der ausgewählten Strategie des Energiemanagements ebenfalls das nächste Plattformszenario und damit den nächsten Betriebsmodus der zugrunde liegenden SCC-Architektur bestimmt. Die Berechnung des nächsten Zustands im geladenen Zustandsraum dauert trotz der Menge der darin enthaltenen Informationen im Durchschnitt lediglich 0,929 ms. Aufgrund dieser kurzen Berechnungszeit impliziert die Auswertung nur einen vernachlässigbaren Mehraufwand während der Laufzeit der Applikation und ermöglicht somit die Anwendung des Optimierungsverfahrens parallel zur eigentlichen Ausführung der Applikation.

Tabelle 7.10: Eigenschaften des Zustandsraums ($\lambda = 2$, $\epsilon = 7\,\text{ms}$, $1/\gamma = 500\,\text{ms}$)

Kriterien	Ergebnisse
Dauer der Berechnung des Zustandsraums	7,727 s
Anzahl der gültigen Zustände im Zustandsraum	1664
Anzahl der Zustandsübergänge im Zustandsraum	29443
Größe des Zustandsraums zur Laufzeitauswertung	3.166.102 Bytes
Zeit zum initialen Laden des Zustandsraums zur Laufzeit	4,58 s
Durchschnittliche Zeit für Zustandsberechnung zur Laufzeit	0,929 ms

Zur Verdeutlichung der Optimierungsergebnisse wird eine Sequenz von 62 Kamerabildern aus einem Beispielszenario 1 herangezogen, das in Abbildung 7.12 dargestellt ist. Die Applikation erlaubt bei der Auswahl der Strategie keine Einbeziehung von Vorkenntnissen, sodass eine Minimierung des Energieverbrauchs durch die Wahl des effizientesten Betriebsmodus in jeder Iteration erreicht wird. Die bei jedem Wechsel des Betriebsmodus benötigte Zeit zur Erreichung eines stabilen Systemzustands wird bei der Effizienzauswertung berücksichtigt. Zu Beginn der Bildsequenz werden keine Verkehrszeichen erkannt, was als Applikationsszenario 0 interpretiert wird. Der durch die Optimierung gewählte Betriebsmodus ist dabei Modus 16, da dessen geringe Leistungsfähigkeit bereits ausreicht. Nach dem sechsten Bild werden zu beiden Seiten der Fahrbahn jeweils zwei Verkehrszeichen erkannt, was zu einen Wechsel zu Applikationsszenario 4 führt. Aufgrund der gestiegenen Belastung wechselt die SCC-Architektur zunächst in Betriebsmodus 6 und anschließend zwischen 2 und 6, da in Betriebsmodus 2 Slack entsteht, der anschließend im energieeffizienteren Modus 6 wieder verbraucht werden kann. Slack kann damit also über Iterationen hinweg propagiert werden, was zur Reduzierung des Gesamtenergieverbrauchs führt.

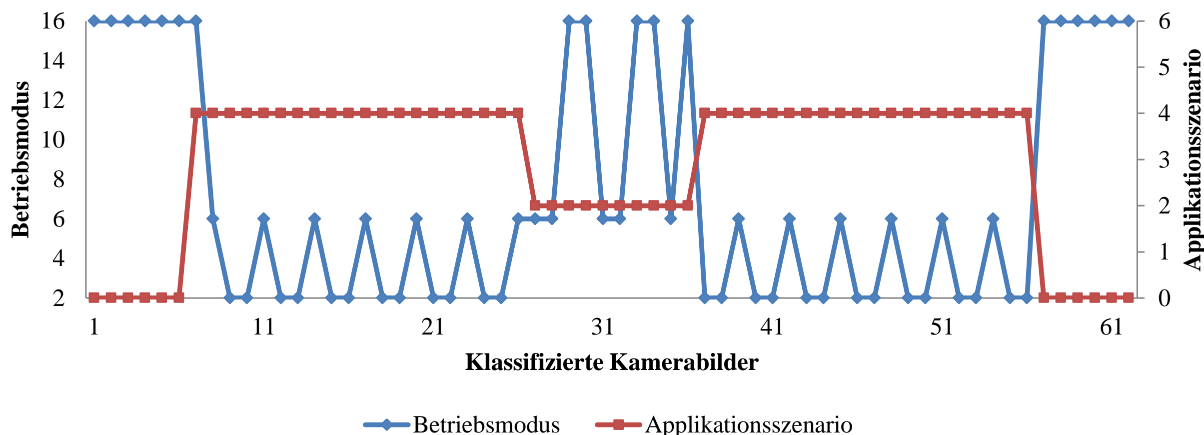


Abbildung 7.12: Resultierende optimierte Wahl der Betriebsmodi in Abhängigkeit der jeweiligen Applikationsszenarien für Beispielszenario 1

Zwischen Bild 27 und Bild 36 sind nur noch zwei Verkehrszeichen sichtbar, sodass in das Applikationsszenario 2 gewechselt wird. Nach einem Verbleib in Betriebsmodus 6 während drei Iterationen wird zur Steigerung der Energieeffizienz anschließend zwischen den Betriebsmodi 6 und 16 gewechselt. Bei erneuter Detektion des Applikationsszenarios 4 muss wie zuvor zwischen den Betriebsmodi 2 und 6 gewechselt werden, um die gestiegenen Anforderungen an die Leistungsfähigkeit des Systems bedienen zu können. Werden zeitweise keine Verkehrszeichen erkannt, wird in den Betriebsmodus mit der niedrigsten Energieverbrauch gewechselt. Die Ausführungszeit für das komplette Beispielszenario 1 beträgt insgesamt 31 Sekunden, was der gewünschten Anforderung von 2 Frames pro Sekunde bzw. $1/\gamma = 500\,\text{ms}$ entspricht.

Die Messung der Leistungsaufnahme erfolgt über das periodische Auslesen spezieller Register auf der SCC-Architektur, wobei die maximal mögliche Periode bei ca. 1 ms liegt. Hier ist zu beachten, dass lediglich die Leistungsaufnahme der kompletten SCC-Architektur gemessen werden kann. Deswegen werden vor allen Messungen alle nicht benötigten Kerne auf den energieeffizientesten Betriebsmodus gesetzt, um deren Einfluss so gering wie möglich zu gestalten.

Weiterhin fließt natürlich die Leistungsaufnahme derjenigen Kerne in die Messung ein, welche die restlichen Teile des Fahrerassistenzsystems ausführen, z.B. den Algorithmus zur Kreiserkennung. Wird dieser auf 3 Kernen parallelisiert ausgeführt, entsteht ein Durchsatz von 2 Frames pro Sekunde, was einer Parametrisierung des Optimierungsalgorithmus mit $1/\gamma = 500\,\text{ms}$ entspricht. In diesem Fall sind insgesamt 8 Kerne gleichzeitig aktiv, wobei in diesem Beispielszenario nur die Energieeffizienz des Klassifizierungsalgorithmus optimiert wird.

Die Ausführung des Fahrerassistenzsystems ohne Powermanagement weißt eine durchschnittliche Leistungsaufnahme von 34,769 W für das oben beschriebene Beispielszenario auf und dient mit diesem Wert als Referenz. Durch die Anwendung des Optimierungsverfahrens zur Laufzeit wird laut Messung die Energieeffizienz um etwas mehr als 11 % erhöht. Tabelle 7.11 enthält neben diesem Wert noch weitere Ergebnisse für unterschiedliche Konfigurationen des ϵ-Werts, welcher den während der Berechnung des Zustandsraums maximal akzeptierten Verlust an Slack repräsentiert. Wie zu erwarten ist, verringert sich die Steigerung der Energieeffizienz mit größerem ϵ-Wert,

da dann tendenziell weniger Slack über mehrere Iterationen propagiert wird und dieser somit nicht für Wechsel in energieeffizientere Betriebszustände zur Verfügung steht. Der Aufbau des Zustandsraums wird dadurch jedoch prinzipiell vereinfacht, da weniger neue Zustände entstehen, weswegen sich auch die Berechnungszeit des entsprechenden Zustandsraums deutlich verringert. Da dieser zur Entwurfszeit berechnet wird, spielt dies aber eine untergeordnete Rolle.

Tabelle 7.11: Ergebnisse für Beispielszenario 1 und $1/\gamma = 500\,\text{ms}$

ϵ-Wert	Berechnung Zustandsraum			∅ Leistungsaufnahme	Energieeffizienz
	# Zustände	# Übergänge	Dauer		
7 ms	1664	29443	7,727 s	30,932 W	+11,04 %
10 ms	1222	21873	4,074 s	31,198 W	+10,27 %
100 ms	91	1624	0,095 s	31,262 W	+10,08 %
200 ms	64	1137	0,047 s	31,664 W	+8,93 %

Tabelle 7.12 zeigt die Ergebnisse der Optimierung zur Laufzeit für eine maximale Periode von 750 ms, wodurch die Ausführungszeit für das komplette Beispielszenario 1 auf insgesamt 46,5 s ansteigt. Der gemessene Referenzwert liegt hier bei 33,971 W für die durchschnittliche Leistungsaufnahme ohne Powermanagement. Durch die Anwendung des Optimierungsverfahrens zur Laufzeit wird bei einem ϵ-Wert von 7 ms die Energieeffizienz um ca. 13,7 % gesteigert. Die im Gegensatz zu Tabelle 7.11 verbesserten Ergebnisse bei der Optimierung sind durch die gesunkenen Anforderungen an den zu erzielenden Durchsatz begründet. Weiterhin fallen die Unterschiede bei der Steigerung der Energieeffizienz in Abhängigkeit des ϵ-Werts geringer aus, da sich die Hardware-Plattform aufgrund der gesunkenen Anforderungen tendenziell öfters in Betriebszuständen mit höherer Energieeffizienz befindet.

Tabelle 7.12: Ergebnisse für Beispielszenario 1 und $1/\gamma = 750\,\text{ms}$

ϵ-Wert	Berechnung Zustandsraum			∅ Leistungsaufnahme	Energieeffizienz
	# Zustände	# Übergänge	Dauer		
7 ms	2789	56021	31,721 s	29,332 W	+13,66 %
10 ms	1963	39240	12,344 s	29,346 W	+13,61 %
100 ms	187	3719	0,253 s	29,408 W	+13,43 %
200 ms	168	3406	0,157 s	29,511 W	+13,12 %

Zum Vergleich ist in Abbildung 7.13 die aus dem Optimierungsschritt resultierende Wahl der Betriebsmodi anhand eines weiteren Beispielszenarios 2 dargestellt. Insgesamt wird hier ebenfalls eine Sequenz von 62 Bildern bearbeitet, die Abfolge der Applikationsszenarien unterscheidet sich jedoch von der in Beispielszenario 1 und kann ebenfalls in der Abbildung abgelesen werden.

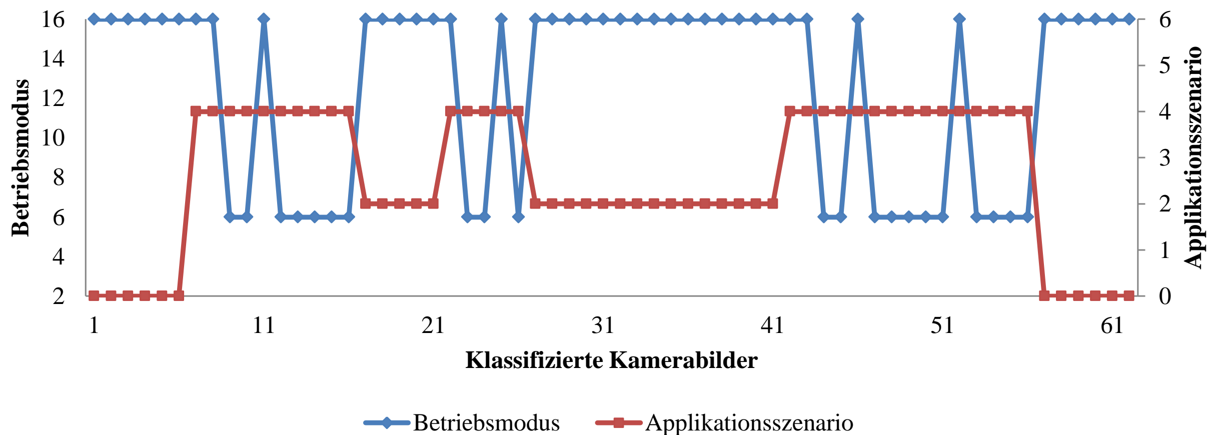


Abbildung 7.13: Resultierende optimierte Wahl der Betriebsmodi in Abhängigkeit der jeweiligen Applikationsszenarien für Beispielszenario 2

Es ist klar erkennbar, dass die SCC-Architektur in Beispielszenario 2 öfter in Betriebsmodus 16 ausgeführt wird. Dies liegt zum einen daran, dass der Klassifizierungsalgorithmus öfter im Applikationsszenario 2 und weniger oft in Applikationsszenario 4 ausgeführt wird, zum anderen an der Erhöhung der maximalen Periode auf 750 ms.

Tabelle 7.13: Ergebnisse für Beispielszenario 2 und $^1/_\gamma = 750\,\text{ms}$

ϵ-Wert	Berechnung Zustandsraum			∅ Leistungsaufnahme	Energieeffizienz
	# Zustände	# Übergänge	Dauer		
7 ms	2789	56021	31,721 s	29,308 W	+14,20 %
10 ms	1963	39240	12,344 s	29,382 W	+13,99 %
100 ms	187	3719	0,253 s	29,446 W	+13,80 %
200 ms	168	3406	0,157 s	29,604 W	+13,34 %

Tabelle 7.13 stellt die gemessene durchschnittliche Leistungsaufnahme während der Ausführung von Beispielszenario 2 für unterschiedliche Konfigurationen des ϵ-Werts dar. Da der Zustandsraum lediglich von der Konfiguration der verwendeten Parameter abhängt, wird derselbe Zustandsraum wie in Tabelle 7.12 verwendet. Der Referenzwert für die Ausführung ohne Powermanagement liegt hier bei 34,160 W.

Durch die Anwendung des entwickelten Optimierungsverfahrens zur Laufzeit wird für einen ϵ-Wert von 7 ms die Energieeffizienz um 14,2 % gesteigert. Selbst für einen ϵ-Wert von 200 ms beträgt die erzielte Steigerung der Energieeffizienz durch eine gezielte Anpassung des aktuellen Betriebsmodus der Hardware-Plattform noch ca. 13,3 % für das dargestellte Beispielszenario.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Beitrag dieser Arbeit

In dieser Arbeit wurde ein Verfahren zur applikationsspezifischen Optimierung der Energieeffizienz in digitalen eingebetteten Hardware/Software-Systemen vorgestellt. Diesem Verfahren liegen modellbasierte Analysen zugrunde, die sowohl das funktionale als auch das nicht-funktionale Verhalten der modellierten Funktionalität, sowie ein Modell der Zielarchitektur und die an das System gestellten Anforderungen berücksichtigen. Die Optimierung ist dazu in verschiedene Phasen aufgeteilt, die sich nach den jeweils zur Verfügung stehenden Eigenschaften richten, und führt zu einer gezielte Anwendung von Mechanismen zur Reduzierung der Leistungsaufnahme auf einer gegebenen Zielarchitektur.

Da während des Systementwurfs zumeist lediglich Abschätzungen bezüglich der Ausführungszeit der Applikationen vorliegen, jedoch ein hoher Freiheitsgrad bezüglich der Abbildung der Applikationen auf Verarbeitungsressourcen und der Auswahl der entsprechenden Betriebszustände der Ressourcen vorliegt, ist eine energieeffiziente Zuordnung der Applikationen zu Ressourcen und Betriebszuständen gesucht. Hierzu wurde der Energieverbrauch des Hardware/Software-Systems in einer Kostenfunktion formuliert, sodass das resultierende Optimierungsproblem unter Berücksichtigung von Randbedingungen durch einen mathematischen Lösungsalgorithmus gelöst werden kann. Da es sich hierbei um ein ganzzahliges nichtlineares Optimierungsproblem handelt, wurde ein darauf angepasster Branch-and-Bound-Algorithmus entwickelt, der iterativ eine Minimierung des Energieverbrauchs durchführt. Das Ergebnis des Optimierungsprozesses ist eine bezüglich des Energieverbrauchs optimierte Abbildung der Applikationen auf Verarbeitungsressourcen, eine auf die Abbildung abgestimmte Ablaufstrategie und eine explizite Zuordnung von Betriebszuständen zu den jeweiligen Applikationen. Weiterhin wurde in dieser Arbeit eine geeignete Abstraktion des Optimierungsproblems präsentiert, durch die das Problem in linearer Form dargestellt werden kann, sodass ein exaktes analytisches Lösungsverfahren angewandt werden kann und somit jedes gefundene Optimum auch dem globalen Optimum entspricht.

Basierend auf diesem Optimierungsergebnis kann eine Konfiguration für ein ausführbares Systemmodell in SystemC abgeleitet werden, welches das funktionale und

nicht-funktionale Verhalten des Hardware/Software-Systems modelliert und frühzeitig simulierbar macht. Hierzu wurde ein Generierungsprozess entwickelt, der mittels Abbildungsregeln eine schnelle und automatische Transformation des modellierten Systems in ein Simulationsmodell über Abstraktionsebenen hinweg realisiert. Zusätzlich wird eine virtuelle Ausführungsplattform zur Simulation einer Systemverwaltungsschicht generiert, die sowohl die Ausführung der abgebildeten Funktionalität gemäß der definierten Ablaufstrategie ermöglicht, als auch ein dynamisches Energie- bzw. Powermanagement integriert. Die aus der Ausführung der Funktionalität auf dem zugrunde liegenden Hardwaremodell resultierende Leistungsaufnahme kann sowohl durch ein zustandsbasiertes Leistungsmodell auf hoher Abstraktionsebene, als auch durch eine Kombination aus zustandsbasiertem und instruktionsbasiertem Leistungsmodell abgebildet werden, falls ein entsprechend mit nicht-funktionalen Eigenschaften instrumentiertes Simulationsmodell verfügbar ist. Zur Integration der Systemumgebung in die Simulation des Hardware/Software-Systems wurde eine Co-Simulationsplattform entwickelt, die exemplarisch mehrere domänenspezifische Simulationswerkzeuge aus dem Automobilbereich verbindet. Durch die ganzheitliche Simulation wird eine frühzeitige Verifikation und Validierung des Systemverhaltens und getroffener Entwurfsentscheidungen, sowie eine damit verbundene Überprüfung der Anforderungen unter Berücksichtigung verschiedener Parameter und der dynamischen Eigenschaften ermöglicht. Weiterhin können typische Anwendungsszenarien während des Betriebs des Hardware/Software-Systems identifiziert werden, die als Eingabe für einen weiteren Optimierungsprozess dienen.

Zur Berücksichtigung des dynamischen Verhaltens von Hardware/Software-Systemen wurde ein Optimierungsverfahren entwickelt, das zur Laufzeit angewandt wird und die Hardware-Plattform bzw. die jeweilige Verarbeitungsressource in einen energieeffizienten Betriebszustand versetzt. In diesem Verfahren werden zunächst die dynamischen Eigenschaften innerhalb der Applikationen, vornehmlich verschiedene Ausführungszeiten der daran beteiligten Tasks aufgrund unterschiedlicher Ausführungspfade, die gegenseitigen Abhängigkeiten der Tasks und Eigenschaften der Hardware-Plattform, wie z.B. verfügbare Betriebszustände und Zustandsübergänge, in einem datenflussorientierten Modell kombiniert. Durch Anwendung und Erweiterung einer (max,+)-Algebra wird dieses Modell in ein mathematisches Modell überführt, mit dem zur Entwurfszeit ein Zustandsraum aufgebaut wird, der alle gültigen Betriebszustände unter Berücksichtigung der möglichen Szenarien der Applikationen enthält. Diese Vorberechnung zur Entwurfszeit ermöglicht die Anpassung der Betriebszustände zur Laufzeit in Abhängigkeit des aktuellen Applikationsszenarios und führt somit zu einer Optimierung der Energieeffizienz währen des Betriebs des Hardware/Software-Systems.

8.2 Ausblick

Der in dieser Arbeit entwickelte Analyse- und Optimierungsansatz basiert auf einem abstrahierten Modell der zugrunde liegenden Hardware-Plattform. So wird implizit davon

ausgegangen, dass jede Verarbeitungsressource bzw. jede Berechnungseinheit unabhängig von allen anderen in einen anderen gültigen Betriebsmodus wechseln kann. Durch die erhöhte Komplexität der Hardware-Implementierung unterschiedlicher Betriebsmodi, z.B. durch die variable Spannungsversorgung, ist das Wechseln der Betriebszustände auf realen Hardware-Plattformen jedoch meist nur in zusammengefassten Bereichen – sogenannten Power Domains – möglich. Wenn diese Power-Domains mehrere Ressourcen umfassen, so wie es beim Intel SCC mit insgesamt 6 Power-Domains zu je 8 Berechnungskernen der Fall ist, betrifft ein Wechsel des Betriebsmodus immer alle Kerne derselben Power-Domain. Diese durch die Zielarchitektur definierte Eigenschaft müsste in einer weiterführenden Ableitung der Strategie des Energiemanagements berücksichtigt werden.

Die in Abschnitt 6.1 vorgestellte Optimierung der Energieeffizienz zur Entwurfszeit kann dadurch erweitert werden, dass bei der Abbildung der Funktionalität eine Berücksichtigung der Speicherhierarchie bzw. eine entsprechende Cache-Metrik im Hinblick auf eine energieeffiziente Lösung erfolgt. Dadurch wird vermieden, dass Applikationen, die z.B. über einen geteilten Speicher kommunizieren, auf Ressourcen abgebildet werden, die eine kostenintensive Kommunikation über entsprechende Kommunikationsstrukturen erfordern und somit die Energieeffizienz verringern, sobald der Energieverbrauch der benötigten Kommunikationsstruktur berücksichtigt wird.

Weiterhin lässt sie die Kommunikationsarchitektur in die Bestimmung der Startzeitpunkte von Applikationen integrieren, sodass diese bei gegenseitigen Abhängigkeiten entweder anhand eines analysierten Kommunikationsverhaltens oder einer Worst-Case-Abschätzung in die Abbildung der Applikationen und Berechnung der Betriebszustände und damit in den Optimierungsschritt integriert wird. Dadurch erfolgt eine zeitliche Berücksichtigung des Kommunikationsverhaltens, das maßgeblichen Einfluss auf die Einhaltung der an das System gestellten Anforderungen haben kann. Insgesamt zeigt diese Thematik einige Schnittpunkte mit Arbeiten aus dem Gebiet der optimierten Abbildung von Prozessen auf Mehrkern-Architekturen, wie sie beispielsweise bereits zur Minimierung der Kommunikation über das Verbindungsnetzwerk existieren.

Basierend auf dem entwickelten Entwurfsablauf erfolgt die Extraktion des dynamischen Verhaltens, das bei der Optimierung zur Laufzeit (siehe Abschnitt 6.2) zur Minimierung des Energieverbrauchs genutzt wird, durch die Generierung eines ausführbaren Modells bzw. einer virtuellen Ausführungsplattform und der Ausführung geeigneter Testfälle, die eine Ableitung der Applikationsszenarien erlaubt. Diese simulative Analyse der Dynamik kann durch eine formal-analytische Extraktion der Applikationsszenarien ersetzt werden, um die Abhängigkeit der Ergebnisse von den durch die Testfälle generierten Stimuli zu reduzieren.

Um die entwickelten Optimierungsschritte auch auf die Erhöhung der Zuverlässigkeit anwenden zu können, sollte eine Anpassung der Analysemodelle zur Verwendung von Leistungsmodellen auf niedrigerer Abstraktionsebene und einer räumlichen Auflösung der Leistungsaufnahme erfolgen. Dadurch würde ein Wechsel des Betriebsmodus nicht nun wie in dieser Arbeit beschrieben zur Steigerung der Energieeffizienz, sondern auch zur Vermeidung von Leistungsspitzen und einer eventuell damit verbundenen lokalen Temperaturentwicklung eingesetzt werden können. Eine Berücksichtigung

von Temperaturschwankungen, die durch ständige Wechsel der Betriebszustände verursacht werden, wäre dann eine ebenfalls mögliche Erweiterung des bestehenden Ansatzes.

Anhang A

Quellcode-Auszüge

A.1 Regelbasierte Modell-zu-Modell-Transformation

Quellcode A.1 zeigt die Transformation *UML2SystemC* in QVT-O, der ein Systemmodell in UML als Eingabemodell und zwei Modelle als Ausgabemodelle übergeben werden. Das erste Ausgabemodell ergibt das SystemC-orientierte Komponentenmodell, das zweite ein Modell, das eine Verknüpfung zwischen dem Komponentenmodell und dem ursprünglichen UML-Modell herstellt und damit in weiteren Transformationsschritten zur Verfügung steht. Dieses *Link-Modell*, das in Abschnitt 5.3.2 detailliert erklärt wird, ist für die nachfolgende Generierung des Simulations-Codes unabdingbar, da es eine eindeutige Verbindung zwischen den Elementen des Eingabemodells und des Komponentenmodells herstellt.

```
1  transformation UML2SystemC(in uml:umlMM, out systemc:systemcMM, out
      link:linkMM);
2  // Einstiegspunkt für die Transformation
3  main() {
4    var umlModel := uml.objects()[uml::Model]->asOrderedSet()->first();
5    scProject := umlModel->map
        umlModel2SystemCModel()->asOrderedSet()->first();
6    ...
7  }
8  // Erstelle SystemC-Modell
9  mapping uml::Model::umlModel2SystemCModel() :
      systemcMM::sc_core::SystemCProject {
10   result.sub_units += uml.objectsOfType(uml::Interface).map
        umlInterface2SystemCUnit();
11   result.sub_units += uml.objectsOfType(uml::Component).map
        umlComponent2SystemCUnit();
12   ...
13 }
14 ...
```

Quellcode A.1: QVT-O-Transformation des Systemmodells in das Komponentenmodell

In Quellcode A.2 ist beispielsweise die Transformation einer UML-Komponente in eine Einheit des Komponentenmodells, in diesem Fall ein SystemC-Modul enthalten. Darin werden gemäß Tabelle 5.1 die Ports der UML-Komponente anhand ihrer Schnittstellenrelation auf unterschiedliche Typen der Ports des Komponentenmodells – also SystemC-Ports – abgebildet.

```
1  mapping uml::Component::umlComponent2SystemModule() :
       systemcMM::sc_core::SC_Module {
2    var portSet : OrderedSet(uml::Port);
3    var requInterfaces : OrderedSet(uml::Interface);
4    var provInterfaces : OrderedSet(uml::Interface);
5    // Bilde Port mit Bedarfsschnittstelle auf SC_Port ab
6    self.findRequiredInterfaces()->forEach(interface) {
7      self.ownedPort->forEach(port) {
8        if (port.type.isDerivedFrom(interface)) then {
9          requInterfaces += interface;
10         portSet += port;
11         result.child_elements += port->map
               compPorts2SC_Ports(port.type.name.getInterfaceName(),result);
12       } endif;
13     };
14   };
15   // Bilde Port mit Angebotsschnittstelle auf SC_Port ab
16   self.findProvidedInterfaces()->forEach(interface) {
17     ...
18   };
19 }
```

Quellcode A.2: Transformation einer UML-Komponente in SystemC-Modul

In Quellcode A.3 ist auszugsweise die Transformation des Komponentenmodells in das AST-Modell enthalten. In diesem Transformationsschritt wird sowohl das Komponentenmodell als auch das entsprechende Link-Modell benötigt, um entsprechende Elemente des AST-Modells zu generieren. Die Transformationsregeln sind in Tabelle 5.2 enthalten.

Quellcode A.4 enthält die die Abbildung einer Schnittstelle des Komponentenmodells, also ein SystemC-Interface, auf eine C++-Klasse. Dabei werden u.a. Elemente für die Sichtbarkeit (ab Zeile 9) angelegt. Ab Zeile 21 werden Vererbungsbeziehungen zwischen den Schnittstellen des Komponentenmodells und den Schnittstellen der entsprechenden Softwarekomponenten generiert.

Quellcode A.5 stellt die Transformation einer UML-Operation, die innerhalb einer UML-Komponente modelliert wurde, in eine abstrakte Repräsentation einer Funktion dar, die wiederum in dem aus der UML-Komponente transformierten Modul bzw. der entsprechenden Klasse enthalten ist. Die Funktion besteht neben dem Funktionsnamen sowohl aus der Definition der Signatur – bestehend aus Rückgabewert und übergebenen Parametern – als auch des Funktionsrumpfs.

```
1  transformation SystemC2AST(in systemc: systemcMM,  in linkmodel : linkMM,
     out ast: astMM);
2  // Einstiegspunkt für die Transformation
3  main() {
4    var sysproj := systemc.objectsOfType(systemcMM::sc_core::SystemCProject)
       ->asOrderedSet();
5    var linkproj :=
       linkmodel.objectsOfType(linkMM::LinkModel)->asOrderedSet();
6    lnModel := linkproj->first();
7    sysproj -> map SystemCProject2ASTProject();
8  }
9  ...
```

Quellcode A.3: QVT-O-Transformation des Komponentenmodells in ein AST-Modell

```
1  mapping
     systemcMM::sc_core::SC_CustomInterface::SystemCIF2SimpleDeclaration
     (filename : String, interfaceLn : linkMM::InterfaceLink) :
     astMM::SimpleDeclaration {
2    result.derivedSyntax := self.name;
3    result.containingFilename := filename;
4    result.declSpecifier := object cppastMM::CompositeTypeSpecifier {
5      derivedSyntax := self.name;
6      name := setName(self.name);
7      containingFilename := filename;
8      // Sichbarkeit
9      members += object VisibilityLabel {
10         visibility := 1;
11         derivedSyntax := self.name + "_label";
12     };
13     // Vererbungsbeziehung zu Basis-Interface in C++
14     baseSpecifiers += object cppastMM::BaseSpecifier {
15       derivedSyntax := interfaceLn.umlInterface.name + "_baseSpecifier";
16       name := setName(interfaceLn.umlInterface.name);
17     };
18   };
19 };
```

Quellcode A.4: Transformation einer SystemC-Schnittstelle in C++-Klasse

A.2 Template-basierte Modell-zu-Text-Transformation

Als Werkzeug für Modell-zu-Text-Transformationen wird *Xpand* verwendet, dass sich in die *Eclipse*-Entwicklungsumgebung integrieren lässt. Xpand liest in einem ersten Schritt das zu transformierende AST-Modell ein und wendet die definierten Transformationsregeln auf diejenigen Elemente an, die für das vorgegebene Muster gültig sind. Schlüsselwörter bzw. Schlüsselkonstrukte der Xpand-Sprache werden durch Anführungszeichen, wie sie in romanischen Sprachen verwendet werden, *Guillemets* genannt, gekennzeichnet. Xpand verfügt über Kontrollstrukturen wie Schleifen, bedingte

```
mapping uml::Operation::umlOperation2FunctionDefinition(filename : String,
    pStatements : OrderedSet(Statement)) : cppastMM::FunctionDefinition {
  result.derivedSyntax := self.name;
  result.containingFilename := filename;
  // Rückgabewert
  declSpecifier := object NamedTypeSpecifier {
    if (self.type != null) then {
      name := setName(self.type.getStringType().
          getStringTypeMultiplicity(self.getUpper()));
    } else {
      name := setName("void");
    } endif;
  };
  // Parameter
  declarator := object cppastMM::FunctionDeclarator {
    name := setName(self.name);
    parameters := createAstParameterSet(self.ownedParameter);
  };
  // Funktionsrumpf
  functionBody := object CompoundStatement {
    scope := object cppMM::BlockScope {
      scopeName := setName(self.name + "_body");
      statements := pStatements;
    };
  };
}
```

Quellcode A.5: Transformation einer UML-Operation in eine Funktionsrepräsentation im AST-Modell

Anweisungen und Verzweigungen, deren Semantik vergleichbar mit denen höherer Programmiersprachen ist.

Quellcode A.6 zeigt in Zeile 1 den Einstieg in die Generierung, indem ein Template definiert wird, das auf das Wurzelelement des AST-Modells *AstProject* angewandt wird. Da dessen Kindelemente in der Listenvariablen *elements* abgelegt sind, wird in Zeile 2 ein weiteres Template expandiert und in einer Schleife auf die Kindelemente angewandt. Ab Zeile 7 ist das Template für den Typ *Declarator* dargestellt, der z.B. für die Deklaration von Variablen oder Methoden angelegt wird. Ist in der Deklaration ein Referenzoperator enthalten, wird vor dem Variablentyp ein „&"-Zeichen generiert, was der Referenzoperation in C/C++ und damit auch SystemC entspricht. Danach wird geprüft, ob es sich um eine Array-Variable oder eine Methodendeklaration handelt und dementsprechend in weiteren Templates verzweigt.

In Quellcode A.7 wird eine Array-Deklaration und die damit verbundenen Generierung der Array-Zeichen „[" bzw. "]" gezeigt. Weiterhin wird ab Zeile 8 die Generierung einer Initialisierung von Klassen oder Konstruktoren dargestellt. In Zeile 16 wird auf das Generierungs-Template der Initialisierungsliste eines Konstruktors verzweigt.

```
1  « DEFINE main FOR AstProject »
2  « EXPAND processNode FOREACH this.elements -»
3  « ENDDEFINE »
4  ...
5  « DEFINE processNode FOR Node -»
6  ...
7  « DEFINE processDeclarator FOR cppast::Declarator -»
8  « FOREACH this.children AS e »« IF e.metaType==cppast::ReferenceOperator
     »&« ENDIF »« ENDFOREACH »
9  « IF this.metaType == ArrayDeclarator -»
10 « EXPAND processArrayDeclarator FOR (cppast::ArrayDeclarator)this -»
11 « ELSEIF this.metaType == FunctionDeclarator »
12 « EXPAND processFunctionDeclarator FOR (FunctionDeclarator)this -»
13 « ELSE -»
14 « EXPAND processPointer FOREACH  this.children.typeSelect(Pointer) -»
15 « EXPAND processName FOR this.name -»
16 « EXPAND processInitializer FOR this.initializer -»
17 « ENDIF -»
18 « ENDDEFINE »
19 ...
```

Quellcode A.6: Template-Definition zur Generierung von Quelltext

```
1  // Array
2  « DEFINE processArrayDeclarator FOR ArrayDeclarator -»
3  « EXPAND processPointer FOREACH  this.children.typeSelect(Pointer) -»
4  « EXPAND processName FOR this.name »« FOREACH this.arrayModifiers AS am
     »[« EXPAND processArrayModifier FOR am »]« ENDFOREACH »
5  « EXPAND processInitializer FOR this.initializer »
6  « ENDDEFINE »
7  // Initialisierungen
8  « DEFINE processInitializer FOR Initializer »
9  « IF this.metaType == ConstructorInitializer »
10 « EXPAND processConstructorInitializer FOR (ConstructorInitializer)this -»
11 ...
12 « ENDDEFINE »
13 // Initialisierung für Konstruktur
14 « DEFINE processConstructorInitializer FOR ConstructorInitializer -»
15 (« FOREACH this.arguments AS e ITERATOR iter -»
16 « EXPAND processInitializerClause FOR e »« IF !iter.lastIteration -»,«
     ENDIF -»
17 « ENDFOREACH »)
18 « ENDDEFINE »
```

Quellcode A.7: Template-Definition für Arrays und Initialisierung von Konstruktoren

Abkürzungen und Symbole

Verwendete Abkürzungen und Akronyme

ACC	Adaptive Cruise Control
ASADF	Application Scenario-Aware Dataflow
AST	Abstract Syntax Tree
BnB	Branch-and-Bound
CP	Communicating Processes
CPS	Cyber-Physical Systems
CPT	Communicating Processes with Time
CPU	Central Processing Unit
DPM	Dynamic Power Management
DSL	Domain-Specific Language
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
ECU	Electronic Control Unit
EDA	Electronic Design Automation
EMF	Eclipse Modeling Framework
ET	Execution Time
FIFO	First In First Out
ILP	Integer Linear Programming
IP	Intellectual Property
ISS	Instruction-Set Simulator

LP	Linear Programming
MDA	Model-driven Architecture
MDE	Model-driven Engineering
MILP	Mixed-Integer Linear Programming
MINLP	Mixed-Integer Nonlinear Programming
MOF	Meta Object Facilities
MPSoC	Multiprocessor System on Chip
NFP	Non-Functional Properties
NLP	Nonlinear Programming
NoC	Network-on-Chip
OMG	Object Management Group
OSCI	Open SystemC Initiative
PMC	Power-Manageable Component
PSM	Power State Machine
QP	Quadratic Programming
QVT	Query View Transformation
RTOS	Real-Time Operating System
SADF	Scenario-Aware Dataflow
SDF	Synchroner Datenfluss
SoC	System-on-Chip
SQP	Sequential Quadratic Programming
SysML	Systems Modeling Language
TDMA	Time Devision Multiple Access
TLM	Transaction-Level Modeling
UML	Unified Modeling Language
VEP	Virtual Execution Platform
WCET	Worst-Case Execution Time

XML Extensible Markup Language

Allgemeine Symbole

$after_{i,j} \in \{0,1\}$ Binärvariable, die anzeigt, ob Applikation i nach Applikation j ausgeführt wird

$before_{i,j} \in \{0,1\}$ Binärvariable, die anzeigt, ob Applikation i vor Applikation j ausgeführt wird

$E^{dev} \in \mathbb{R}_0^+$ Energie für zusätzliche Komponenten

$E^{dyn} \in \mathbb{R}_0^+$ Energie aufgrund dynamischer Leistungsaufnahme

$E^{stat} \in \mathbb{R}_0^+$ Energie aufgrund statischer Leistungsaufnahme

$E^{sw} \in \mathbb{R}_0^+$ Energie für Wechsel zwischen Betriebsmodi

$e_i \in \mathbb{R}_0^+$ Endzeitpunkt von Applikation i

$E^{sw}_{(m,r)\times(n,r)} \in \mathbb{R}_0^+$ Energie für Wechsel von Betriebsmodus m nach Betriebsmodus n auf Ressource r

$g_i^{dur} \in \mathbb{R}_0^+$ Länge der Lücke nach Applikation i

$o_{i,r} \in \{0,1\}$ Binärvariable, die anzeigt, ob Ressource r in Lücke nach Applikation i ausgeschaltet wird

$o_{i,r}^{dur} \in \mathbb{R}_0^+$ Zeit, in der Ressource r in Lücke nach Applikation i eingeschaltet ist

$P_d^{dev} \in \mathbb{R}_0^+$ Leistungsaufnahme von Komponente d

$P_r^{stat} \in \mathbb{R}_0^+$ Statische Leistungsaufnahme von Ressource r

$P_{i,m,r}^{dyn} \in \mathbb{R}_0^+$ Dynamische Leistungsaufnahme bei Ausführung von Applikation i in Betriebsmodus m auf Ressource r

$s_i \in \mathbb{R}_0^+$ Startzeitpunkt von Applikation i

$sr_{i,j} \in \{0,1\}$ Binärvariable, die anzeigt, ob Applikationen i und j auf derselben Ressource ausgeführt werden

$t^H \in \mathbb{R}_0^+$ Maximale (Hyper-)Periode

$t_d^{BE} \in \mathbb{R}_0^+$ Break-Even-Zeit von Komponente d

$t_r^{BE} \in \mathbb{R}_0^+$ Break-Even-Zeit von Ressource r

$t_{i,m,r} \in \mathbb{R}_0^+$	Ausführungszeit von Applikation i in Betriebsmodus m auf Ressource r
$x_{i,m,r} \in \{0, 1\}$	Zuordnung von Applikation i zu Betriebsmodus m und Ressource r
$x_{i,r} \in \{0, 1\}$	Zuordnung von Applikation i zu Ressource r
$z_{i,d} \in \{0, 1\}$	Zuordnung von Applikation i zu Komponente d
$\delta(m, n) \in \mathbb{R}_0^+$	Zeitlicher Aufwand für Wechsel von Betriebsmodus m nach Betriebsmodus n
$\epsilon \in \mathbb{R}_0^+$	Akzeptierter Verlust an Slack
$\lambda \in \mathbb{R}_0^+$	Anforderung an Durchsatz
$\sigma^k \in \mathbb{R}^l$	Slack-Vektor über l Token in Iteration k
$v^k \in \mathbb{R}^l$	Zeitstempel-Vektor über l Token in Iteration k
AS	Menge der Applikationsszenarien
$G_{as,ps} \in \mathbb{R}_0^+$	Ausführungsmatrix von Applikationsszenario as und Plattformszenario ps
$PM \subseteq PS$	Menge der Betriebsmodi bzw. Power-Modi
PS	Menge der Plattformszenarien
$ps^k \in PS$	Plattformszenario in Iteration k
S	Menge der Zustände im Zustandsraum

Abbildungsverzeichnis

1.1 Lösungsansatz zur Analyse und Optimierung der Energieeffizienz . . . 5

2.1 Diagramme der Unified Modeling Language (UML) (Bildquelle: [92]) . . 11
2.2 Erweiterung der UML-Diagrammstruktur durch SysML (Bildquelle: [93]) 12
2.3 Beispiel eines SDF-Graphen . . . 23
2.4 ARM Cortex™ A-9/15 Mehrkern-Architekturen (Bildquelle: ARM) 26
2.5 Tilera®Mehrkern-Architektur (Bildquelle: Tilera) . . . 26
2.6 Intel Single-Chip-Cloud-Computer (Bildquelle: Intel) . . . 27
2.7 ARM big.LITTLE-Architektur (Bildquelle: [44]) . . . 28
2.8 Funktionsweise und Leistungsaufnahme von CMOS-Transistoren 31
2.9 Power State Machine eines Strong-ARM-Prozessors SA-1100 [11] 34
2.10 Schematischer Aufbau eines Power-Managers (nach [11]) . . . 35

4.1 Konzept zur Optimierung der Energieeffizienz in unterschiedlichen Phasen 70
4.2 Aus dem Konzept abgeleiteter Entwurfsfluss . . . 72

5.1 Modellierung der Funktionalität der Software-Komponenten . . . 77
5.2 Modellierung der Hardware-Plattform . . . 79
5.3 Modellierung des Hardware/Software-Abbildung . . . 80
5.4 Modellierung der Ausführung von UML-Operationen . . . 81
5.5 Nicht-funktionale Eigenschaften in der UML-Modellierung mit MARTE 82
5.6 Struktur einer klassischen Softwarearchitektur . . . 84
5.7 Schichtenmodell für die Integration von Softwarekomponenten 86
5.8 Schichtenmodell für Hardware/Software-System . . . 87
5.9 Übersicht der angewandten Modell-Transformationen . . . 93
5.10 Metamodell des Komponentenmodells basierend auf SystemC . . . 94
5.11 Metamodell zur Instanziierung und Verbindung von Komponenten . . . 96
5.12 Metamodell für transaktionsorientierte Kommunikation basierend auf SystemC-TLM . . . 96
5.13 Konzept von regelbasierten Modell-zu-Modell-Transformationen 97
5.14 Verknüpfung zwischen Systemmodell und Komponentenmodell 99
5.15 Ausführung von NFP-annotiertem Simulationsmodell auf der virtuellen Ausführungsplattform (VEP) . . . 104
5.16 Trennung zwischen realer und virtueller Simulationsebene . . . 104
5.17 VEP-Datenstruktur für NFP-Annotationen . . . 105

5.18 Thread-Scheduling und Einbeziehung von (externen) Ereignissen 106
5.19 Flussdiagramm für `consume()`-Methode zur Verarbeitung der NFP-Annotationen im Simulationsmodell [73] 108
5.20 Flussdiagramme für weitere zentrale VEP-Methoden [73] 109
5.21 Metamodellierung für das Energiemanagement auf der virtuellen Ausführungsplattform (VEP) 110
5.22 Zeitlicher Verlauf der Anwendung eines reaktiven Energiemanagements bei 2 Beispielkomponenten 112
5.23 Ableitung von Applikationsszenarien durch Simulation 114

6.1 Zeitlich überlagerte Benutzung einer geteilten Device-Komponente durch mehrere Applikationen 124
6.2 Zeitliche Überlagerung auf Ressource r von vorne bzw. von hinten . . . 126
6.3 Prinzipieller Ablauf des zugrunde liegenden Optimierungsalgorithmus 128
6.4 Separationsphase innerhalb des Branch-and-Bound-Verfahrens 130
6.5 Beispiel mit 5 Applikationen und gültiger Startbelegung [30] 132
6.6 Optimale Ausführung der Beispiel-Applikationen [30] 133
6.7 Abhängigkeit zwischen Ausführungszeit und Ergebnis der Optimierung 134
6.8 Abhängigkeit zwischen Ausführungszeit und Anzahl der durchlaufenen Zweige 135
6.9 Erweiterung des SDF-Modells zur Berücksichtigung der Hardware-Plattform und der Ausführungszeiten abhängig vom Betriebsmodus . . 141
6.10 Orthogonales Konzept der Multi-Domänen-Szenarien 143
6.11 Symbolische Ausführung der Aktoren eines SDF-Modells 147
6.12 Durch Zeitstempel-Vektoren berechnete Produktionszeiten der Token . . 148
6.13 Aufbau des Zustandsraums mit Durchsatzanforderung $1/4$ 152
6.14 Größe des Zustandsraums in Abhängigkeit von Durchsatzanforderung und potentiellem Slack-Verlust 156
6.15 Überführung des Zustandsraums in eine geschlossene Form 157

7.1 Generische Co-Simulationsplattform 170
7.2 Ablauf der Co-Simulation und Synchronisation der Simulationskerne . . 172
7.3 Abbremsvorgang mit ABS und unterschiedlichen Buslatenz [73] 175
7.4 Abbremsvorgang mit ABS und einer Buslatenz von 60 ms [73] 175
7.5 Temperaturentwicklung auf Chip-Ebene [73] 178
7.6 Temperaturentwicklung auf Chip-Ebene mit Energiemanagement 179
7.7 Einteilung der Power-Domains in SCC-Architektur (Bildquelle: Intel) . . 181
7.8 Anwendung zur Kamera-basierten Erkennung von Verkehrszeichen . . 182
7.9 Reduzierung des Energieverbrauchs in Abhängigkeit verschiedener Konfigurationen von abgebildeten Applikationen und verfügbaren Ressourcen186
7.10 SDF-Graph und Applikationsszenarien des Klassifizierungsalgorithmus 188
7.11 Metamodell der Datenstruktur zur Speicherung des Zustandsraums . . 189
7.12 Resultierende optimierte Wahl der Betriebsmodi in Abhängigkeit der jeweiligen Applikationsszenarien für Beispielszenario 1 191

7.13 Resultierende optimierte Wahl der Betriebsmodi in Abhängigkeit der jeweiligen Applikationsszenarien für Beispielszenario 2 193

Tabellenverzeichnis

2.1 Vergleich Performanz und Energieeffizienz von ARM Cortex A-15 und A-7 29

5.1 Transformationsregeln zur Erstellung des Komponentenmodells 98
5.2 Transformationsregeln der Elemente des Komponentenmodells 101

7.1 Ergebnisse generierter statischer Quelltext für X-By-Wire-System 163
7.2 Mehraufwand durch Integration der VEP 164
7.3 Simulationsdauer bei unterschiedlichen VEP-Konfigurationen 165
7.4 Mehraufwand durch Energiemanagement 166
7.5 Vergleich der Ausführung mit bzw. ohne Energiemanagement 167
7.6 Bewertung unterschiedlicher Konfigurationen der Kommunikation . . . 176
7.7 Maximale Frequenz abhängig von Versorgungsspannung 181
7.8 Analysierte Zeiten zum Wechseln des Betriebsmodus auf dem Intel SCC 182
7.9 Ausführungszeiten der Klassifizierung pro Kreis abhängig vom Betriebsmodus der Intel SCC-Architektur 188
7.10 Eigenschaften des Zustandsraums ($\lambda = 2$, $\epsilon = 7\,\text{ms}$, $1/\gamma = 500\,\text{ms}$) 190
7.11 Ergebnisse für Beispielszenario 1 und $1/\gamma = 500\,\text{ms}$ 192
7.12 Ergebnisse für Beispielszenario 1 und $1/\gamma = 750\,\text{ms}$ 192
7.13 Ergebnisse für Beispielszenario 2 und $1/\gamma = 750\,\text{ms}$ 193

Verzeichnis der Definitionen

2.1 Definition (Energieverbrauch) 9
2.2 Definition (Energieeffizienz) 10
2.3 Definition (Optimierung der Energieeffizienz) 10
2.4 Definition (Aufgaben-Parallelismus) 29
2.5 Definition (Daten-Parallelismus) 29

4.1 Definition (Logischer Durchsatz) 65
4.2 Definition (Maximale Periode) 65
4.3 Definition (Applikation) 68
4.4 Definition (Task) 68
4.5 Definition (Verarbeitungsressource) 68
4.6 Definition (Device) 69

6.1 Definition (Hyper-Periode) 118
6.2 Definition (Applikationsszenario) 142
6.3 Definition (ASADF-Graph) 142
6.4 Definition (Plattformszenario) 144
6.5 Definition (PSM) 144
6.6 Definition (Erweiterungen (max,+)-Algebra) 145
6.7 Definition (Slack-Vektor) 149
6.8 Definition (Szenarien-Zustandsraum) 150
6.9 Definition (Szenarien-Transitionsfunktion) 150

Verzeichnis der Algorithmen

6.1 Branch-and-Bound-Algorithmus mit allgemeiner Branching-Heuristik . 131
6.2 Algorithmus zum Aufbau des Zustandsraums 151
6.3 Algorithmus zur Limitierung des Zustandsraums 154
6.4 Überprüfung auf vollständige Abdeckung der Applikationsszenarien . . 155

7.1 Erstellung von synthetischen Applikationen 184

Literaturverzeichnis

[1] *fmincon Active-Set Algorithm*. http://www.mathworks.de/help/toolbox/optim/ug/brnoxzl.html#brnox01. [Online; Stand Februar 2013].

[2] S. Abdi, D. Shin und D. Gajski: *Automatic communication refinement for system level design*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 300–305. IEEE, 2003.

[3] Accellera Systems Initiative: *SystemC*. http://www.systemc.org.

[4] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Amsterdam, 2. Aufl., 2006.

[5] K. Albers, F. Bodmann und F. Slomka: *Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems*. In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2006.

[6] A. Alimonda, S. Carta, A. Acquaviva, A. Pisane und L. Benini: *A Feedback-Based Approach to DVFS in Data-Flow Applications*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(11), 2009.

[7] G. M. Amdahl: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *Proceedings of the spring joint computer conference*, AFIPS '67 (Spring), S. 483–485, New York, NY, USA, 1967. ACM.

[8] A. Andrei, P. Eles, O. Jovanovic, M. Schmitz, J. Ogniewski und Z. Peng: *Quasi-Static Voltage Scaling for Energy Minimization With Time Constraints*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 19(1):10–23, 2011.

[9] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri und L. Benini: *An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Bd. 1, S. 1–6, 2006.

[10] F. Baccelli, G. Cohen, G. J. Olsder und J. P. Quadrat: *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992.

[11] L. Benini, A. Bogliolo und G. De Micheli: *A Survey of Design Techniques for System-Level Dynamic Power Management*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(3):299–316, 2000.

[12] R. A. Bergamaschi und Y. W. Jiang: *State-based power analysis for systems-on-chip*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 638–641, New York, NY, USA, 2003. ACM.

[13] J. Borrmann, A. Viehl, O. Bringmann und W. Rosenstiel: *Parallel video-based traffic sign recognition on the Intel SCC many-core platform*. In: *Conference on Design and Architectures for Signal and Image Processing*, DASIP, S. 1–2, 2012.

[14] J. T. Buck: *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. Doktorarbeit, University of California, Berkeley, 1993.

[15] J. T. Buck: *A dynamic dataflow model suitable for efficient mixed hardware and software implementations of DSP applications*. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, S. 165–172. IEEE Computer Society Press, 1994.

[16] M. Burckhardt, A. Preukschat, J. Reimpell und A. Zomotor: *Fahrwerktechnik: Bremsvorgang, Bremsstabilität, Kräfte, Fahrzeug-, Fahrbahn- und Reifeneinfluß, Bremsanlage: Einzelteile, Auslegung, Berechnung und Volumenhaushalt, Anhängerbetrieb und Anhängerbremsen. Bremsdynamik und Pkw-Bremsanlagen*. Nr. 10 in *Vogel-Fachbuch : KFZ-Technik*. Vogel, 1991.

[17] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey und R. Brodersen: *Optimizing power using transformations*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(1):12 –31, 1995.

[18] J.-J. Chen und C.-F. Kuo: *Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms*. In: *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, S. 28–38, Washington, DC, USA, 2007. IEEE Computer Society.

[19] J.-J. Chen, A. Schranzhofer und L. Thiele: *Energy minimization for periodic real-time tasks on heterogeneous processing units*. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS)*, S. 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[20] ChiasTek: *CosiMate*. `http://www.chiastek.com/products/cosimate.html`. [Online; Stand Februar 2013].

[21] K. Choi, R. Soma und M. Pedram: *Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(1):18–28, 2005.

[22] L. Chouambe, B. Klatt und K. Krogmann: *Reverse Engineering Software-Models of Component-Based Systems*. In: K. Kontogiannis, C. Tjortjis und A. Winter (Hrsg.): *12th European Conference on Software Maintenance and Reengineering*, S. 93–102. IEEE Computer Society, 2008.

[23] P. Derler, E. A. Lee und A. L. Sangiovanni-Vincentelli: *Addressing Modeling Challenges in Cyber-Physical Systems*. Techn. Ber. UCB/EECS-2011-17, EECS Department, University of California, Berkeley, Mar 2011.

[24] V. Devadas und H. Aydin: *On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications*. In: *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT, S. 99–108, New York, NY, USA, 2008. ACM.

[25] G. Dhiman und T. Šimunić Rosing: *Dynamic voltage frequency scaling for multitasking systems using online learning*. In: *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED, S. 207–212, New York, NY, USA, 2007. ACM.

[26] R. Dömer: *System-level Modeling and Design with the SpecC Language*. Doktorarbeit, Universität Dortmund, 2000.

[27] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs und Y. Xiong: *Taming Heterogeneity – the Ptolemy Approach*. Proceedings of the IEEE, 91(2), 2003.

[28] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam und D. Burger: *Power challenges may end the multicore era*. Communications of the ACM, 56(2):93–102, Feb. 2013.

[29] J. Falk, J. Keinert, C. Haubelt, J. Teich und S. S. Bhattacharyya: *A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications*. In: *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT)*. ACM, 2008.

[30] C. Feil: *Modellierung und Entwicklung von Betriebsstrategien zur Steigerung der Energieeffizienz in Elektrofahrzeugen*. Diplomarbeit, Forschungszentrum Informatik, 2012.

[31] H. Frank und U. Schmidt: *Komplexität besser beherrschen*. In: *Elektronik automotive*, S. 49–53, 2007.

[32] F. Fummi, S. Martini, G. Perbellini und M. Poncino: *Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, S. 564–569, Washington, DC, USA, 2004. IEEE Computer Society.

[33] D. Gajski, S. Abdi, A. Gerstlauer und G. Schirner: *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.

[34] D. Gajski und L. Cai: *Transaction Level Modeling: An Overview*. 2003.

[35] M. Geilen: *Reduction techniques for synchronous dataflow graphs*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 911–916, New York, NY, USA, 2009. ACM.

[36] M. Geilen: *Synchronous dataflow scenarios*. ACM Transactions on Embedded Computer Systems, 10:1–31, 2011.

[37] M. Geilen und S. Stuijk: *Worst-case performance analysis of synchronous dataflow scenarios*. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, S. 125–134, New York, NY, USA, 2010. ACM.

[38] A. Gerstlauer, S. Chakravarty, M. Kathuria und P. Razaghi: *Abstract system-level models for early performance and power exploration*. In: *Proceesings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, S. 213–218. IEEE, 2012.

[39] A. Gerstlauer, H. Yu und D. D. Gajski: *RTOS Modeling for System Level Design*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, S. 130–135, Washington, DC, USA, 2003. IEEE Computer Society.

[40] S. V. Gheorghita, T. Basten und H. Corporaal: *Intra-task scenario-aware voltage scheduling*. In: *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, S. 177–184, New York, NY, USA, 2005. ACM.

[41] T. Givargis, F. Vahid und J. Henkel: *Instruction-Based System-Level Power Evaluation of System-on-a-Chip Peripheral Cores*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 10(6), 2002.

[42] J. Gladigau, C. Haubelt und J. Teich: *Model-Based Virtual Prototype Acceleration*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 31(10):1572 –1585, 2012.

[43] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha und F. Zhao: *Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 191–196, New York, NY, USA, 2008. ACM.

[44] P. Greenhalgh: *Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7*. Techn. Ber., ARM, 2011.

[45] T. Groetker, S. Liao, G. Martin und S. Swan: *System Design with SystemC*. Springer, 2002.

[46] A. Grzemba, A. Faschingbauer, M. Fuchs, P. Scheer und T. Liebetrau: *Reduktion des Stromverbrauchs des Kommunikationsnetzes durch Teilnetzbetrieb*. In: *3. Elektronik automotive congress*, 2011.

[47] J. Gustafsson, A. Betts, A. Ermedahl und B. Lisper: *The Mälardalen WCET Benchmarks – Past, Present and Future*. S. 137–147. OCG, 2010.

[48] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter und R. Ernst: *System level performance analysis – the SymTA/S approach*. IEEE Proceedings of Computers and Digital Techniques, 152(2):148–166, 2005.

[49] S. Herbert und D. Marculescu: *Analysis of dynamic voltage/frequency scaling in chip-multiprocessors*. In: *ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED, S. 38 –43. IEEE, 2007.

[50] M. Horowitz, T. Indermaur und R. Gonzalez: *Low-power digital design*. In: *Digest of Technical Papers, IEEE Symposium on Low Power Electronics*, S. 8–11, 1994.

[51] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De und R. Van Der Wijngaart: *A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling*. IEEE Journal of Solid-State Circuits, 46(1):173–183, 2011.

[52] J. Hu, Y. Shin, N. Dhanwada und R. Marculescu: *Architecting voltage islands in core-based system-on-a-chip designs*. In: *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED, S. 180–185, 2004.

[53] IEEE: *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2002.

[54] IEEE: *IEEE Standard for Verilog Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2006.

[55] IEEE Computer Society: *IEEE 1666-2011, IEEE Standard for Standard SystemC Language Reference Manual*, 2012.

[56] T. M. Inc: *Simulink Getting Started Guide*. The MathWorks Inc, 2012.

[57] INTRACOM S.A., Politecnico di Torino, University of Patras, Fraunhofer-Gesellschaft zur Förderung der Angewandten Forschung e.V., STMicroelectronics s.r.l., Interuniversitair Micro-Electronica Centrum, Aristotle University of Thessaloniki, BullDAST s.r.l., Consorzio Ferrara Ricerche: *EASY: Energy-Aware SYstem-on-chip design of the HIPERLAN/2 standard*. `http://easy.intranet.gr`.

[58] S. Irani, R. Gupta und S. Shukla: *Competitive Analysis of Dynamic Power Management Strategies for Systems with Multiple Power Savings States*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 2002.

[59] A. Iranli und M. Pedram: *System-Level Power Management: An Overview*, 2007.

[60] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose und M. Martonosi: *An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget*. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, S. 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[61] ITEA2 Modelisar Projektpartner: *Functional Mock-up Interface (FMI)*. `https://www.fmi-standard.org/`. [Online; Stand Februar 2013].

[62] A. Jantsch: *Modeling Embedded Systems and SoC's*. Morgan Kaufmann, 1. Aufl., 2003.

[63] G. Kahn: *The semantics of a simple language for parallel programming*. In: *Information Processing: Proceedings of the IFIP Congress*, S. 471–475. North Holland Publishing Co., 1974.

[64] R. M. Karp und R. E. Miller: *Parallel program schemata: A mathematical model for parallel computation*. In: *IEEE Conference Record of the Eighth Annual Symposium on Switching and Automata Theory*, SWAT, S. 55 –61, 1967.

[65] K. Knoedler, J. Steinmann, S. Laversanne, S. Jones, A. Huss, E. Kural, D. Sanchez, O. Bringmann und J. Zimmermann: *Optimal energy management and recovery for FEV*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, S. 683–684. IEEE, 2012.

[66] P. M. Kogge: *An exploration of the technology space for multi-core memory/logic chips for highly scalable parallel systems*. Innovative Architecture for Future Generation High-Performance Processors and Systems, Jan. 2005.

[67] M. Krause: *Automatisierte Erstellung und Verfeinerung virtueller Prototypen für den Entwurf verteilter eingebetteter Systeme*. Doktorarbeit, Universität Tübingen, 2011.

[68] K. Kuipers: *BNB20*. `http://www.mathworks.com/matlabcentral/fileexchange/95-bnb`. [Online; Stand Februar 2013].

[69] P. Kutzer, J. Gladigau, C. Haubelt und J. Teich: *Automatic generation of system-level virtual prototypes from streaming application models*. In: *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, S. 128 –134, 2011.

[70] S. Köhler, J. Zimmermann, H. Sahin, O. Bringmann und W. Rosenstiel: *Optimierung der Rekuperation im Elektrofahrzeug durch Co-Simulationstechniken*. In: *Proceedings of the 9. ITG/GI/GMM Workshop Cyber-Physical-Systems - Enabling Multi-Nature Systems (CPMNS)*, 2012.

[71] M. Küster, A. Viehl, A. Burger, O. Bringmann und W. Rosenstiel: *Meta-Modelling the SystemC Standard for Component-based Embedded System Design*. In: *Proceedings of the 1st International Workshop on Metamodelling and Code Generation for Embedded Systems (MeCoES)*, S. 35–40, 2012.

[72] K. Lahiri und A. Raghunathan: *Power analysis of system-level on-chip communication architectures*. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, CODES+ISSS, S. 236–241, New York, NY, USA, 2004. ACM.

[73] M. Lamparter: *Entwicklung einer Co-Simulationsplattform zur ganzheitlichen Simulation und Bewertung von E/E-Architektur und Fahrzeugdynamik bei Elektrofahrzeugen*. Diplomarbeit, Forschungszentrum Informatik, 2012.

[74] E. A. Lee: *Cyber Physical Systems: Design Challenges*. Techn. Ber. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[75] E. A. Lee und D. G. Messerschmitt: *Static scheduling of synchronous data flow programs for digital signal processing*. IEEE Transactions on Computers, 36(1):24–35, 1987.

[76] E. A. Lee und T. M. Parks: *Dataflow Process Networks*. Bd. 83, S. 773 –801. IEEE Press, 1995.

[77] E. A. Lee und H. Zheng: *Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems*. In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded software*, EMSOFT, S. 114–123, New York, NY, USA, 2007. ACM.

[78] S. Leyffer: *Integrating SQP and Branch-and-Bound for Mixed Integer Nonlinear Programming*. Computational Optimization and Applications, 18(3):295–309, 2001.

[79] Y. Li und J. Henkel: *A framework for estimating and minimizing energy dissipation of embedded HW/SW systems*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 188–193, 1998.

[80] S. Lloyd: *Least squares quantization in PCM*. IEEE Transactions on Information Theory, 28(2):129–137, 1982.

[81] E. Macii, M. Pedram und F. Somenzi: *High-level power modeling, estimation, and optimization*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(11):1061 –1079, 1998.

[82] D. Marculescu und S. Garg: *Process-Driven Variability Analysis of Single and Multiple Voltage-Frequency Island Latency-Constrained Systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(5):893–905, 2008.

[83] D. Markovic, V. Stojanovic, B. Nikolic, M. Horowitz und R. Brodersen: *Methods for true energy-performance optimization*. IEEE Journal of Solid-State Circuits, 39(8):1282–1293, 2004.

[84] P. Marwedel: *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer-Verlag Berlin Heidelberg, 2. Aufl., 2011.

[85] H.-U. Michel, A. Barthels und G. Walla: *Jedes Watt zählt - Intelligentes Energie- und Leistungs-Management für die Autos von morgen*. In: *Elektronik automotive*, Bd. 5, S. 24–28. 2012.

[86] K. Neumann und M. Morlock: *Operations Research*. Hanser Fachbuchverlag, 2. Aufl., 2002.

[87] D. Nienhüser, T. Bär, R. Kohlhaas, T. Schamm, J. Zimmermann, T. Gumpp, M. Strand, O. Bringmann und J. M. Zöllner: *Energy Efficient Driving and Operation Strategies Based on Situation Awareness and Reasoning*. it - Information Technology, 54(1):5–16, 2012.

[88] K. Niyogi und D. Marculescu: *Speed and voltage selection for GALS systems based on voltage/frequency islands*. In: *Proceedings of the Asia and South Pacific Design Automation Conference*, ASP-DAC, S. 292–297, New York, NY, USA, 2005. ACM.

[89] J. Nocedal und S. Wright: *Numerical Optimization*. Springer series in operations research and financial engineering. Springer Science+Business Media, LLC., 2. Aufl., 2006.

[90] Object Management Group (OMG): *MOF2 QVT Specification*. Object Management Group, 2011.

[91] Object Management Group (OMG): *Systems Modeling Language*. Object Management Group, 2011.

[92] Object Management Group (OMG): *Unified Modeling Language (UML)*. Object Management Group, 2011.

[93] Object Management Group (OMG): *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Object Management Group, 2012.

[94] Y.-H. Park, S. Pasricha, F. J. Kurdahi und N. Dutt: *Methodology for multi-granularity embedded processor power model generation for an ESL design flow*. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, S. 255–260, New York, NY, USA, 2008. ACM.

[95] T. M. Parks: *Bounded scheduling of process networks*. Doktorarbeit, University of California, Berkeley, CA, USA, 1995.

[96] M. Pehnt: *Energieeffizienz: Ein Lehr- und Handbuch*. Springer, 2011.

[97] C. A. Petri: *Kommunikation mit Automaten*. Doktorarbeit, Technische Hochschule Darmstadt, 1962.

[98] P. Pillai und K. G. Shin: *Real-time dynamic voltage scaling for low-power embedded operating systems*. SIGOPS Oper. Syst. Rev., 35(5):89–102, 2001.

[99] K. Popovici und A. Jerraya: *Flexible and abstract communication and interconnect modeling for MPSoC*. In: *Proceedings of Asia and South Pacific Design Automation Conference*, S. 143–148, 2009.

[100] H. Posadas, J. Adamez, E. Villar, F. Blasco und F. Escuder: *RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model*. Design Automation for Embedded Systems, 10:209–227, 2005.

[101] M. Y. Qadri und K. D. McDonald-Maier: *Data cache-energy and throughput models: Design exploration for embedded processors*. EURASIP Journal on Embedded Systems, 13, 2009.

[102] G. Qu: *What is the limit of energy saving by dynamic voltage scaling?*. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD*, S. 560–563, Piscataway, NJ, USA, 2001. IEEE Press.

[103] A. Raghunathan, N. K. Jha und S. Dey: *High-Level Power Analysis and Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[104] D. Ramanathan, S. Irani und R. Gupta: *An analysis of system level power management algorithms and their effects on latency*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21(3):291 –305, mar 2002.

[105] P. Rong und M. Pedram: *Determining the optimal timeout values for a power-managed system based on the theory of Markovian processes: offline and online algorithms*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, S. 1128–1133. European Design and Automation Association, 2006.

[106] H. Sahin: *Konzeptionierung und Entwicklung einer Co-Simulationsplattform zur Bewertung von energieeffizienten Fahr- und Betriebsstrategien im Automobil*. Diplomarbeit, Forschungszentrum Informatik, 2011.

[107] B. Sander, J. Schnerr und O. Bringmann: *ESL power analysis of embedded processors for temperature and reliability estimations*. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, S. 239–248, New York, NY, USA, 2009. ACM.

[108] P. Sanders und J. Speck: *Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks*. In: *Euro-Par 2012 Parallel Processing*, Bd. 7484 d. Reihe *Lecture Notes in Computer Science*, S. 167–178. Springer Berlin Heidelberg, 2012.

[109] A. Sangiovanni-Vincentelli und G. Martin: *Platform-based design and software design methodology for embedded systems*. Design Test of Computers, IEEE, 18(6):23 –33, 2001.

[110] M. T. Schmitz, B. M. Al-Hashimi und P. Eles: *Iterative schedule optimization for voltage scalable distributed embedded systems*. ACM Transactions on Embedded Computing Systems, 3(1):182–217, 2004.

[111] C. Schmutzler, A. Krüger, F. Schuster und M. Simons: *Energy efficiency in automotive networks: Assessment and concepts*. In: *International Conference on High Performance Computing and Simulation*, HPCS, S. 232–240. IEEE, 2010.

[112] J. Schnerr, O. Bringmann, A. Viehl und W. Rosenstiel: *High-performance timing simulation of embedded software*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 290 –295. ACM, 2008.

[113] T. Schönwald, J. Zimmermann, O. Bringmann und W. Rosenstiel: *Network-on-Chip Architecture Exploration Framework*. In: *Proceedings of the 12th Euromicro Conference on Digital System Design (DSD)*, S. 375–382, 2009.

[114] A. Schranzhofer, J.-J. Chen und L. Thiele: *Power-Aware Mapping of Probabilistic Applications onto Heterogeneous MPSoC Platforms*. In: *Real-Time and Embedded Technology and Applications Symposium*, RTAS, S. 151 –160. IEEE, 2009.

[115] S. Stattelmann, O. Bringmann und W. Rosenstiel: *Fast and accurate source-level simulation of software timing considering complex code optimizations*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 486–491, New York, NY, USA, 2011. ACM.

[116] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann und W. Rosenstiel: *Hybrid source-level simulation of data caches using abstract cache models*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, S. 376 –381. European Design and Automation Association, 2012.

[117] M. Streubuhr, R. Rosales, R. Hasholzner, C. Haubelt und J. Teich: *ESL power and performance estimation for heterogeneous MPSOCS using SystemC*. In: *Forum on Specification and Design Languages (FDL)*, 2011.

[118] S. Stuijk: *Predictable Mapping of Streaming Applications on Multiprocessors*. Doktorarbeit, Technische Universiteit Eindhoven, 2007.

[119] Synopsys: *Virtualizer*. `http://www.synopsys.com/systems/virtualprototyping/pages/virtualizer.aspx`.

[120] C. Szyperski: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[121] J. Teich und C. Haubelt: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Verlag Berlin Heidelberg, 2. Aufl., 2007.

[122] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita und S. Stuijk: *A scenario-aware data flow model for combined long-run average and worst-case performance analysis*. In: *Proceedings of 4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. ACM, 2006.

[123] L. Thiele, S. Chakraborty und M. Naedele: *Real-time calculus for scheduling hard real-time systems*. In: *Proceedings of the IEEE International Symposium on Circuits and Systems*, Bd. 4, 2000.

[124] M. Thoma: *Modellierung und optimierte Abbildung eingebetteter Software auf Multi-/Many-core Systeme*. Diplomarbeit, Forschungszentrum Informatik, 2010.

[125] V. Tiwari, S. Malik und A. Wolfe: *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*. 2(4), 1994.

[126] TLK-Thermo GmbH: *TISC Suite*. `http://www.tlk-thermo.com/`. [Online; Stand Februar 2013].

[127] TOMLAB: *TOMLAB Optimization in MATLAB*. `http://tomopt.com/tomlab/`. [Online; Stand Februar 2013].

[128] A. Tran, D. Truong und B. Baas: *A GALS many-core heterogeneous DSP platform with source-synchronous on-chip interconnection network*. In: *ACM/IEEE International Symposium on Networks-on-Chip*, NoCS, S. 214–223. IEEE, 2009.

[129] G. Varatkar und R. Marculescu: *Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization*. In: *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD, Washington, DC, USA, 2003. IEEE Computer Society.

[130] A. Viehl: *Quantitative Synchronisationsanalyse kommunizierender Prozesse zum Echtzeitnachweis verteilter eingebetteter Systeme*. Doktorarbeit, Universität Tübingen, 2012.

[131] A. Viehl, M. Pressler, O. Bringmann und W. Rosenstiel: *White Box Performance Analysis Considering Static Non-Preemptive Software Scheduling*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 2009.

[132] T. Šimunić, L. Benini, G. De Micheli und M. Hans: *Source code optimization and profiling of energy consumption in embedded systems*. In: *Proceedings of the 13th international symposium on System synthesis*, ISSS, S. 193–198, Washington, DC, USA, 2000. IEEE Computer Society.

[133] R. Xu, R. Melhem und D. Mosse: *Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors*. In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, S. 25–38, Washington, DC, USA, 2007. IEEE Computer Society.

[134] Y. Xu, R. Rosales, B. Wang, M. Streubühr, R. Hasholzner, C. Haubelt und J. Teich: *A very fast and quasi-accurate power-state-based system-level power modeling methodology*. In: *Proceedings of the 25th international conference on Architecture of Computing Systems (ARCS)*, S. 37–49, Berlin, Heidelberg, 2012. Springer-Verlag.

[135] Y. Yang, Z. Gu, C. Zhu, R. P. Dick und L. Shang: *ISAC: Integrated Space-and-Time-Adaptive Chip-Package Thermal Analysis*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(1):86–99, 2007.

[136] W. Ye, N. Vijaykrishnan, M. Kandemir und M. J. Irwin: *The design and use of simplepower: a cycle-accurate energy estimation tool*. In: *Proceedings of the International Design Automation Conference (DAC)*, S. 340–345, New York, NY, USA, 2000. ACM.

[137] J. Zimmermann: *Entwicklung eines fehlerrobusten Routingverfahrens auf SystemC-TLM-Ebene für Network-on-Chip-Architekturen*. Diplomarbeit, Forschungszentrum Informatik, 2006.

[138] J. Zimmermann, O. Bringmann, J. Gerlach, F. Schäfer und U. Nageldinger: *Comprehensive platform and component modeling of heterogeneous interconnected systems*. In: *Proceedings of the 11th Forum on Specification, Verification and Design Languages (FDL)*, S. 227 –232. IEEE, 2008.

[139] J. Zimmermann, O. Bringmann und W. Rosenstiel: *Analysis of Multi-Domain Scenarios for Optimized Dynamic Power Management Strategies*. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 2012.

[140] J. Zimmermann, M. Küster, O. Bringmann und W. Rosenstiel: *Model-Based Generation of a Fast and Accurate Virtual Execution Platform for Software-Intensive Real-Time Embedded Systems*. In: *Proceedings of the 17th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*, 2012.

[141] J. Zimmermann, M. Pressler, A. Viehl, O. Bringmann und W. Rosenstiel: *Model-based virtual prototyping for early automotive software systems evaluation*. In: *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED) at DATE*, 2010.

[142] J. Zimmermann, S. Stattelmann, A. Viehl, O. Bringmann und W. Rosenstiel: *Model-Driven Virtual Prototyping for Real-Time Simulation of Distributed Embedded Systems*. In: *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2012.

Index

A

Abbildungsregeln ... 93
Ablaufplanung ... 81, 105, 112, 146
Abstrakter Syntaxbaum ... 92, 98, 102
Active-Set ... 47, 130
Aktor ... 22, 140
Anforderungen ... 65, 119, 139, 145
Angebotsschnittstelle ... 77, 83, 100
Applikation ... 68
Applikationsszenario ... 110, 114, 142, 187
Artefakt ... 101
Aufgabenparallelismus ... 29, 184
Ausführbares Systemmodell ... 83
Ausführungsmatrix ... 146
Ausführungsmodus ... *siehe* Betriebsmodus
Ausführungspfad ... 113, 143, 187
Ausführungssemantik ... 83

B

Bedarfsschnittstelle ... 78, 83, 100
Berechnungsgraph ... 20
Berechnungsmodell ... 64
Betriebsmodus ... 33, 92, 112, 116, 144
Betriebsmoduswechsel ... 119, 137
Bin-Packing ... 186
Black-Box-Analyse ... 51
Body Biasing ... 33
Bottom-Up-Entwurf ... 102
Branch-and-Bound-Verfahren ... 40, 127, 134
Branching-Heuristik ... 129
Break-Even-Zeit ... 35, 119
Budget ... 112

C

Callback Method ... *siehe* Rückruf-Methode
Cluster-Analyse ... 113
Co-Simulationsplattform ... 168
Communicating Processes ... 87
Compiler ... 18
Cyber-Physische Systeme ... 54, 167

D

Datenparallelismus ... 29
Deployment ... 80, 84
Detektionsschritt ... 156, 187
Device ... 69, 122
Device-Energie ... 121, 137
Device-Komponente ... *siehe* Device
Dispatcher ... 105
Domänenspezifische Sprache ... 187
Dynamic Power Management39, 118, 139, 178
Dynamic Voltage and Frequency Scaling . 37, 118, 139, 180
Dynamische Leistungsaufnahme . 30, 118, 136

E

E/E-Architektur ... 161, 168, 173, 174
Echtzeitanforderung ... 65
Echtzeitsysteme ... 65
Eclipse ... 77, 93, 94, 100, 201
Electronic Control Unit ... 173
Elektrische Energie ... 9
EMF Ecore ... 94, 189
Energieeffizienz ... 10, 63, 115
Energiemanagement . 4, 53, 103, 109, 178, 180
Energieverbrauch ... 9

F

FD-SOI ... 32
Fixed Priority ... 81, 105
FlexRay ... 174
Frequency Island ... 27, 180

G

Ganzheitliche Simulation 75, 92, 173
Ganzzahliges Optimierungsproblem . 40, 127
Gegenseitige Abhängigkeiten 125

H

Hardware-in-the-Loop 19
Hardware/Software-Systeme 1, 75, 167
Heterogene Architektur 28
Heuristik 158
Homogene Architektur 25
Hyper-Periode 118

I

Instruktionsbasiertes Leistungsmodell 36, 66, 107, 177
Instruktionssatz-Simulator 52
Instrumentierung 18
Intel SCC 27, 180
Inter-Task-Optimierung 71, 116
Intra-Task-Optimierung 73, 139
IP-XACT 79
Iteration 23, 146

K

k-means-Algorithmus 114
Kahn-Prozess-Netzwerk 21
Karush-Kuhn-Tucker-Bedingungen . . 43, 131
Komponentenmodell 94, 98, 199
Konkreter Syntaxbaum 102
Kostenfunktion . . . . *siehe* Zielfunktion

L

Leistungsaufnahme 30, 107, 118, 183
Leistungsmodelle 33
Lineares Optimierungsproblem . . 40, 127, 134, 183
Link-Modell 99, 199
Logischer Durchsatz 65, 144

M

Manycore-Architekturen 24
MARTE 11, 81
MATLAB/Simulink 19, 130, 132, 169, 183
Max-Plus-Algebra 23, 145
Maximale Periode 65, 145
Mehrstufiges Schichtenmodell 83
Mixed-Integer Linear Programming 127
Mixed-Integer Non-Linear Programming . 127
Model-driven Architecture 12
Model-driven Engineering 12
Modell-zu-Modell-Transformation 93
Modell-zu-Text-Transformation 100, 201
Modellgenerierung 91, 162
MOST 176
Multicore-Architekturen 24

N

Nebenbedingungen 117, 124
Network-on-Chip 25
Newton-Verfahren 45
NFP-Annotation 18, 103
Nicht-funktionale Eigenschaften . . 12, 81, 142, 168
Nichtlineares Optimierungsproblem . 41, 127
Non-Linear Programming 127

O

Object Management Group 10, 12
Optimierung 10, 40, 41, 63, 69, 116, 138

P

Papyrus 77
Performanz 65, 145
Performanzanforderungen . . . . *siehe* Anforderungen
Petri-Netz 23
Pfadsimulation 18
Plattformszenarien 144
Power Domain 180, 197
Power State Machine 34, 110, 144, 182
Power-Manager 34
Power-Modus . . . . *siehe* Betriebsmodus
Problemaufteilung 128
Provided Interface *siehe* Angebotsschnittstelle

Q

Quadratisches Optimierungsproblem 44
Quasi-Newton-Verfahren 45
Query-View-Transformation 12, 93
QVT-Operational 13, 199

R

Rückruf-Methode 91
Randbedingungen . *siehe* Nebenbedingungen
Reaktives Energiemanagement 109
Real-Time Operating System 53
Realer Prototyp 177
Rekonstruktion 18
Relaxation 129
Repetitionsvektor 22, 146
Required Interface . . *siehe* Bedarfsschnittstelle
Ressource . . . . . *siehe* Verarbeitungsressource
Round-Trip-Engineering 100, 102
RT-Ebene 87

S

S-Function 19, 172
Scheduling . . . . . *siehe* Ablaufplanung
SDF-Graph *siehe* Synchroner Datenflussgraph
Selektion 129
Selektionschritt 156
Self-Timed Execution 23, 146
Separation 129
Sequential Quadratic Programming . . 48, 131
Simplex-Verfahren 40, 134
Simulation 13, 51, 62, 168
Simulationsmodell 91, 174
Slack-Vektor 149
Software-in-the-Loop 19
Special-Ordered-Set 40, 129
Statische Leistungsaufnahme 31, 119, 136
Synchroner Datenfluss 140, 187
Synchroner Datenflussgraph 22, 187
Synchrones Datenflussmodell 22
SysML-Block 11, 78
SystemC 13, 85, 169
SystemC-Interface 14, 85
SystemC-Method 16, 95
SystemC-Modul 14, 85
SystemC-Port 14, 85, 95
SystemC-Simulationskern 15, 171
SystemC-Thread 16, 95
SystemC-TLM 17, 87
Systems Modeling Language 10, 78

T

Task 68, 139
TDMA 81, 105
Teilnetzbetrieb 161, 173
Temperaturentwicklung 177
Testumgebung 113
Timed SDF-Graph 23
Timing-Pfad . . . . . *siehe* Ausführungspfad
TLM-Initiator 88
TLM-Target 88
Token 22, 140
Top-Down-Entwurf 92
Transaction-Level Modeling 17, 95
Transaktion 88

U

UML-Klasse 77, 98
UML-Klassendiagramm 77
UML-Komponente 10, 77, 98
UML-Konnektor 80, 98
UML-Operation 200
UML-Port 79, 98
Unified Modeling Language 10, 76

V

Verarbeitungsressource 68
Verkehrszeichenerkennung 176
Virtuelle Ausführungsplattform . . 83, 100, 103, 163, 177, 187
Virtuelle Hardware 52, 62, 78
Virtuelle Zeitebene 103
Virtueller Prototyp 72, 92, 176
Voltage Island 27, 180

W

White-Box-Analyse 51
Workload-Balancing 24

Worst-Case Execution Time . . 23, 70, 113, 116, 140, 187

X

X-By-Wire 176
Xpand 100, 201

Z

Zeitstempel-Vektor 146, 150
Zielfunktion 116, 117, 136
Zustandsbasiertes Leistungsmodell 33, 66, 177
Zustandsraum 149, 156
Zustandsraumlimitierung 153