Fortschritt-Berichte VDI

VDI

Reihe 8

Mess-, Steuerungs- und Regelungstechnik

Nr. 1257

Dipl.-Inform. Dipl.-Wirt.Inform. Sten Grüner, Ilvesheim

Ressourcenadaptive Anwendungen für die operative Prozessleittechnik

ACPLT AACHENER PROZESSLEITTECHNIK

Lehrstuhl für Prozessleittechnik der RWTH Aachen

Ressourcenadaptive Anwendungen für die operative Prozessleittechnik

Von der Fakultät für Georessourcen und Materialtechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von **Dipl.-Inform. Dipl.-Wirt.Inform.**

Sten Grüner

aus Ekaterinburg, Russland

Berichter:	Univ.-Prof. Dr.-Ing. Ulrich Epple
	Univ.-Prof. Dr.-Ing. Stefan Kowalewski
Tag der mündlichen Prüfung:	23. Mai 2017

Fortschritt-Berichte VDI

Reihe 8

Mess-, Steuerungs- und Regelungstechnik

Dipl.-Inform. Dipl.-Wirt.Inform.
Sten Grüner, Ilvesheim

Nr. 1257

Ressourcenadaptive Anwendungen für die operative Prozessleittechnik

Lehrstuhl für
Prozessleittechnik
der RWTH Aachen

Grüner, Sten
Ressourcenadaptive Anwendungen für die operative Prozessleittechnik
Fortschr.-Ber. VDI Reihe 8 Nr. 1257. Düsseldorf: VDI Verlag 2017.
166 Seiten, 51 Bilder, 2 Tabellen.
ISBN 978-3-18-525708-7, ISSN 0178-9546,
€ 62,00/VDI-Mitgliederpreis € 55,80.

Für die Dokumentation: Prozessleittechnik – Softwarearchitektur – Ressourcenadaptive Anwendungen – IEC 61131 – Slackzeit – Scheduling – Laufzeitumgebung

Die aktuellen Entwicklungen der Prozessleittechnik und der Automatisierungstechnik fordern die Verlagerung zusätzlicher Funktionen auf die Prozessleitebene der Automatisierungspyramide. Die Ausführungszeit der anwenderspezifischen Logik innerhalb der Laufzeitsysteme unterliegt gewissen Fluktuationen. Die überschüssige Zeit, Slackzeit genannt, bleibt wegen der festen Zykluszeit häufig ungenutzt. Der Beitrag dieser Arbeit ist ein Rahmenwerk für die nahtlose Integration von ressourcenadaptiven Anwendungen in solche Laufzeitsysteme. Diese Anwendungen können für die Bereitstellung der zusätzlichen Funktionalität während der Slackzeit genutzt werden, ohne die Echtzeitanforderungen und den Funktionsumfang der existierenden Kernanwendung einzuschränken. Der Mehrwert des Rahmenwerks für die Prozessleittechnik wird an mehreren Use-Cases demonstriert. Dazu zählen Anwendungen mit und ohne Zugriff in den operativen Betrieb des Laufzeitsystems.

Bibliographische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet unter http://dnb.ddb.de abrufbar.

Bibliographic information published by the Deutsche Bibliothek
(German National Library)
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie (German National Bibliography); detailed bibliographic data is available via Internet at http://dnb.ddb.de.

D82 (Diss. RWTH Aachen University, 2017)

ISSN 0178-9546
ISBN 978-3-18-525708-7

Vorwort

Die vorliegende Arbeit wurde während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Prozessleittechnik der RWTH Aachen University angefertigt. Für die Förderung seitens der DFG im Rahmen des Graduiertenkollegs 1298 „AlgoSyn“ (Algorithmische Synthese reaktiver und diskret-kontinuierlicher Systeme) möchte ich mich an dieser Stelle gesondert bedanken.

Mein besonderer Dank gebührt meinem Doktorvater Herrn Univ.-Prof. Dr.-Ing. Ulrich Epple. Sein Gespür für aussichtsreiche Forschungsthemen an der Schnittstelle zwischen Automatisierungstechnik und Informatik, die Bereitschaft zum Ideenaustausch, die vertrauensvoll ermöglichten Freiräume und die durch ihn geschaffene einzigartige interdisziplinäre Atmosphäre am Lehrstuhl hat meine Arbeitsweise maßgeblich geprägt. Dies hat wesentlich zum Gelingen dieser Arbeit beigetragen.

Ebenso danke ich Herrn Univ.-Prof. Dr.-Ing. Stefan Kowalewski, Inhaber des Lehrstuhls für eingebettete Systeme der RWTH Aachen University, für die freundliche Übernahme der Zweitbegutachtung. Herrn Univ.-Prof. Dr.-Ing. Gerhard Hirt, Leiter des Instituts für Bildsame Formgebung der RWTH Aachen University, danke ich für die Übernahme des Prüfungsvorsitzes.

Für das geweckte Interesse am Graduiertenkolleg gilt mein Dank dessen Sprecher, Herrn Univ.-Prof. Dr. rer. nat. Dr. h.c. Wolfgang Thomas, Inhaber des Lehrstuhls für Logik und Theorie diskreter Systeme der RWTH Aachen University. Für die organisatorische Arbeit rund um das Graduiertenkolleg danke ich Frau Helen Bolke-Hermanns und Frau Silke Cormann.

Für viele fruchtbare Diskussionen möchte ich mich bei meinen Kollegen und Kolleginnen am Lehrstuhl und den Mitgliedern des Graduiertenkollegs bedanken. Besonders erwähnen möchte ich (in alphabetischer Reihenfolge) Herrn Lars Evertz, Herrn David Kampert, Herrn Florian Palm, Herrn Andreas Schüller und Herrn Constantin Wagner. Weiterhin danke ich Herrn Dr. techn. Alois Zoitl, Herrn Univ.-Prof. Dr.-Ing. habil. Leon Urbas und Herrn Julius Pfrommer für das frühe Feedback zu den Kernideen dieser Arbeit.

Frau Margarete Milescu-Huber und Frau Ursula Bey danke ich für die Unterstützung bei den diversen organisatorischen Tätigkeiten. Ebenso danke ich Frau Margarete Milescu-Huber und Frau Julia Botov für die Durchsicht dieser Arbeit. Des Weiteren möchte ich mich bei den studentischen Hilfskräften des Lehrstuhls bedanken.

Meiner Ehefrau Natalia und unseren Kindern Konstantin und Alexandra danke ich von ganzem Herzen für die ständige Unterstützung, Geduld und Motivation während der gesamten Promotion.

Schließlich möchte ich meiner Mutter Renate für die liebevolle Unterstützung bei jeder beruflichen und privaten Entscheidung danken.

Ilvesheim, im Mai 2017

Sten Grüner

"Simplicity is prerequisite for reliability."
—Edsger W. Dijkstra, 1975

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABK Anzeige- und Bedienkomponente

AdA Automatisierung der Automatisierung

API Application Programming Interface

BCET Best Case Execution Time

CFC Continuous Function Chart

CPPS Cyber-Physical Production Systems

CPS Cyber-Physical Systems

CPU Central Processing Unit

CTL Computational Tree Logic

DSL Domain-Specific Language

ECC Execution Control Chart

EDF Earliest Deadline First

ERP Enterprise Resource Planning

EWS Engineering Workstation

FBD Funktionsbaustein oder Funktionsbausteinsprache (engl. Function Block Diagram)

FBN Funktionsbausteinnetzwerk

GAMS General Algebraic Modeling System

HMI Human-Machine Interface

HTTP Hypertext Transfer Protocol

I/O Input-Output

IDE Integrated Development Environment

ILP Integer Linear Program

ISSC In-cycle Sequential State Chart

kgV kleinstes gemeinsames Vielfaches

LP Linear Program

MILP Mixed Integer Linear Program

MINLP Mixed Integer Nonlinearly Constrained Program

MIQP Mixed Integer Quadratic Program

OO Objektorientierung

OPC UA Open Platform Communications Unified Architecture

PLC Programmable Logic Controller (deutsch SPS)

PLS Prozessleitsystem

PNK prozessnahe Komponente

POU Program Organization Unit

QoC Quality of Control

R&I Fließschema Rohrleitungs- und Instrumentenfließschema

RA Resource-Aware oder ressourcenadaptiv

RTE Runtime Environment

SDK Software Development Kit

SFC Sequential Function Chart (deutsch Ablaufsprache)

SK selbstständige Komponente

SOA serviceorientierte Architektur

SPS speicherprogrammierbare Steuerung (engl. PLC)

SSC Sequential State Chart

ST Structured Text (deutsch strukturierter Text)

SysML Systems Modeling Language

TA Timed Automata

TCP Transmission Control Protocol

TCTL Timed Computation Tree Logic

UML Unified Modeling Language

WCET Worst Case Execution Time

XML-RPC Extensible Markup Language Remote Procedure Call

Kurzfassung

Die aktuellen Entwicklungen der Prozessleittechnik und der Automatisierungstechnik, die unter den Oberbegriffen „Industrie 4.0“ und „Cyber-Physical Production Systems“ subsumiert werden, fordern die Verlagerung bzw. die Bereitstellung zusätzlicher Funktionen auf die Prozessleitebene der Automatisierungspyramide. Diese Funktionen umfassen beispielsweise Self-X Funktionalitäten wie Selbstoptimierung und Selbstdiagnose einzelner Komponenten sowie die Bereitstellung zusätzlicher nicht-echtzeitrelevanter Daten wie die Beschreibung der Fähigkeiten und Merkmale des Systems. Diese zusätzlichen Funktionen machen den Unterschied zwischen herkömmlichen Systemen und smarten Industrie 4.0-Systemen aus.

Die Bereitstellung der zusätzlichen Funktionalität erfordert überplanmäßige Rechen- und Kommunikationsressourcen, was insbesondere im Hinblick auf die echtzeitkritischen Laufzeitumgebungen nichttrivial ist. Zum einen werden die verfügbaren Ressourcen einzelner Systeme bereits vollständig genutzt bzw. reserviert, zum anderen könnten die Betreiber den Aufwand der notwendigen Rekonfiguration des Systems unter anderem aus Gründen der langen Betreibs- und Lebenszyklen der Systeme scheuen.

Die Laufzeitsysteme der Prozessleitebene werden in den meisten Fällen in einem konstanten Zyklus betrieben, der dem zu kontrollierenden physischen System angepasst ist. Da die Ausführungszeit der anwenderspezifischen Logik gewissen Fluktuationen sowie Überabschätzungen in Bezug auf die maximale Laufzeit unterliegt, variiert die tatsächliche Ausführungszeit innerhalb des Zyklus. Die überschüssige Zeit, Slackzeit genannt, bleibt wegen der festen Zykluszeit häufig ungenutzt.

Die dynamische Anpassung der benötigten (Rechen-)Ressourcen ist eine Dimension der Flexibilität leittechnischer Anwendungen, die in der Domäne der Automatisierungstechnik bislang unbeachtet blieb. In den Bereichen der Echtzeitsysteme und des Schedulings existieren dagegen bereits Konzepte, die den Ausgangspunkt für diese Arbeit darstellen. Die Analyse dieser Ansätze, unter Berücksichtigung der aufgestellten spezifischen Anforderungen der Leittechnik, bildet die Grundlage dieser Dissertation.

Die Zielsetzung dieser Arbeit ist ein Rahmenwerk für die nahtlose Integration von ressourcenadaptiven Anwendungen in die zyklischen Laufzeitsysteme. Diese Anwendungsklasse kann für die Bereitstellung der zusätzlichen Funktionalität während der Slackzeit genutzt werden, ohne die Echtzeitanforderungen und den Funktionsumfang der existierenden Kernanwendung einzuschränken.

Ein Beitrag der Arbeit ist eine Softwarearchitektur, die die Koexistenz unterschiedlicher Ausführungsparadigmen innerhalb eines Laufzeitsystems ermöglicht. Die Paradigmen umfassen die tasklistengesteuerte Ausführung nach IEC 61131-3, die ereignisgesteuerte Ausführung nach IEC 61499 sowie die Einbettung weiterer Ausführungsvorschriften, wie z. B. der eingeführten ressourcenadaptiven Ausführung. Dieses ist durch die konsequente Kapselung der Daten und der Ausführungsvorschrift innerhalb der Komponenten sowie des Prinzips des hierarchischen Schedulings möglich.

Eine mögliche Ausführungsvorschrift wird durch das zusätzlich eingeführte Meta-Modell zur Beschreibung der Ausführungszeitsensitivität für Funktionsbaustein-Anwendungen innerhalb des Laufzeitsystems definiert. Dazu wird die Semantik der Prozedurbeschreibungssprache Sequential State Chart angepasst, um die Auswertung des Charts innerhalb eines Zyklus des Laufzeitsystems zu ermöglichen. Die Syntax der Sprache ist den meisten Nutzern bekannt, was positiv zu der Akzeptanz der Sprache beiträgt. Die Semantik der Prozedur wird formal mithilfe des UPPAAL-Toolkits modelliert, das neben der Eindeutigkeit auch zusätzliche Möglichkeiten für das Engineering, wie z. B. die formale Validierung und Simulation, eröffnet.

Anschließend wird eine Referenzarchitektur für den systemweiten Komponentenscheduler vorgestellt, der die Überwachung und die dynamische Zuteilung der Slackzeit an die einzelnen ressourcenadaptiven Komponenten sicherstellt. Für diesen Zweck wird eine Kombination aus offline und online Scheduling verwendet. Die Berechnung des offline Schedules beinhaltet das Lösen eines NP-harten Problems, das mithilfe eines gemischt-ganzzahligen linearen Programms und eines passenden Solvers aufgestellt bzw. gelöst wird. Die Kombination aus einem offline und online Verfahren ermöglicht die Ausführung der ressourcenadaptiven Anwendungen sowie weitere Möglichkeiten der Flexibilisierung des Schedulings, wie z. B. die Möglichkeit des dynamischen Austauschs des Schedules zur Laufzeit bei gleichzeitiger Sicherstellung der Echtzeitschranken.

Das eingeführte Rahmenwerk inklusive einer Engineering-Umgebung wurde als Erweiterung der quelloffenen Laufzeitumgebung ACPLT/RTE prototypisch implementiert. Der Mehrwert der ressourcenadaptiven Anwendungen für die Prozessleittechnik wird an mehreren Use-Cases demonstriert. Dazu zählen Anwendungen mit und ohne Zugriff in den operativen Betrieb des Laufzeitsystems. Zu der ersten Kategorie gehören die prozessbegleitende Simulation mit variabler Simulationsgenauigkeit und die mehrstufige Messwertvalidierung. In die zweite Kategorie fallen die nicht-echtzeitfähige Kommunikation mittels OPC UA und die Transaktionskontrolle für regelbasiertes Engineering im Rahmen der Automatisierung der Automatisierung.

Abstract

Resource-Aware Applications for Operative Process Control Engineering

Current developments in process control engineering and industrial automation that can be subsumed under the umbrella terms “Industrie 4.0” and “Cyber-Physical Production Systems”, require the provision of additional functionalities to the process control layer of the automatization pyramid. These functionalities include, for example, Self-X functionalities like self-optimization and self-diagnosis of single automation components as well as the provision of additional non real-time information like the description of system capabilities and attributes. These additional functionalities make all the difference between conventional and smart Industrie 4.0 production systems.

The deployment of these additional services requires supplementary computation- and communication-resources. Providing these resources is non-trivial due to the hard real-time requirements of industrial runtime environments. On one hand, the available resources may already be completely utilized or reserved. On the other hand, operators may shy away from the required costs of the system reconfiguration due to the long service- and life-cycles of the utilized equipment.

Industrial runtime systems are usually operated with a fixed cycle time that is fitted to the controlled physical system. The effectively utilized processing time of the whole system varies due to fluctuations in the actual execution times of the application-specific control logic as well as overestimations of its worst-case execution time. The unused processing time at the end of a cycle, the so called slack time, is usually not utilized by current runtime environments.

A dynamic adaptation of required (computational) resources is a dimension of flexibility of industrial automation applications that has not been focused upon in process control engineering research. However, some approaches for resource-awareness of applications exist in the research communities of real-time systems and scheduling theory. A review of these approaches constitutes a starting point for this dissertation. The analysis of the available approaches and frameworks has to be performed under the aspects of derived domain-specific functional and non-functional requirements.

The goal of this work is to develop a framework for a seamless integration of resource-aware applications into cyclic runtime environments. This class of industrial automation applications can be used for the provision of additional functionality during the slack time which per definition cannot violate the real-time requirements and the functionality of the existing runtime’s core-application.

A first contribution of this work is a software architecture which allows the coexistence of different execution control paradigms within one runtime environment. These paradigms comprise a task list-based execution according to IEC 61131-3, an event-based execution of IEC 61499 as well as embedding further execution control rules such as the introduced resource-aware execution mechanisms. This embedment is possible due to a rigorous

encapsulation of dataflow and execution control flows within a program organization unit and the utilization of mechanisms of hierarchical scheduling.

One possibility of realizing the resource-aware execution control is represented by the introduced meta-model for describing the execution-time sensitivity of function block applications inside the runtime environment. The meta-model is built upon a procedure description language called Sequential State Chars of which the semantics are adopted so that they can be evaluated during the cycle of the runtime system. The syntax of the language is familiar to most users in the industrial automation domain thus allowing a higher acceptance of the introduced framework. The semantics of the procedure description language is formalized by using a transformation to timed automata of the UPPAAL-toolkit. This transformation not only allows an unambiguous semantics of the introduced meta-model, but also adds additional possibilities for the engineering, e.g. formal validation and simulation of modelled procedures.

Subsequently, a reference architecture for a resource-aware system level component scheduler is introduced. This architecture allows the monitoring and dynamic assignment of slack time to single resource-aware components at runtime. The presented scheduler uses a combination of offline and online scheduling. The computation of an offline scheduling table requires solving an NP-hard problem. This task is accomplished by modelling the scheduling problem as a mixed-integer program and solving it with available solvers. The combination of offline and online scheduling techniques not only allows the execution of resource-aware applications, but also the use of additional features like the dynamic exchange of offline scheduling tables at runtime and the execution of sporadic tasks.

The introduced resource-aware framework and the appendant engineering environment were prototypically implemented as an extension of an open source industrial runtime environment ACPLT/RTE. The additional value of resource-aware applications is demonstrated in different use cases with and without operative process intervention. The first category includes the process accompanying simulation with variable simulation precision and multistage validation of measured values. The second category contains non real-time communication with OPC UA and transaction control that is used for rule based engineering systems in the domain of automation of automation.

1. Einleitung

1.1. Motivation

Die während der letzten Jahre zu beobachtenden Tendenzen in der Automatisierungstechnik fordern die Ausweitung der Aufgabengebiete der eingesetzten Systeme und deren Komponenten. Die Systeme sollen damit neben deren eigentlichen Kernfunktionen zusätzliche Funktionalität beinhalten, um beispielsweise die vertikale und horizontale Integration zu erleichtern bzw. zu ermöglichen.

Von diesen Entwicklungen sind alle Ebenen der Automatisierungspyramide betroffen. Beispielsweise werden Feldgeräte zu „intelligenten Feldgeräten“ weiterentwickelt, die eine Parametrierung bzw. Integration in ein Leitsystem ohne zusätzliche Programmierschnittstellen ermöglichen. Auf der Prozessleitebene werden immer mehr Laufzeitsysteme mit nicht-echtzeitfähigen Kommunikationsschnittellen ausgestattet, die über hybride Feldbussysteme realisiert sind. Diese Schnittstellen ermöglichen den Austausch von Meta-Daten neben dem operativen Betrieb. Glücklicherweise sind die Kommunikationsprotokolle für einen modellbasierten Zugriff auf solche Daten, wie z. B. Open Platform Communications Unified Architecture (OPC UA) [IEC10], rechtzeitig standardisiert worden und erreichen eine zunehmende Durchdringung des Marktes für die Automatisierungslösungen.

Die Aggregation und die Repräsentanz zusätzlicher Daten und Funktionen ist in dem kürzlich spezifizierten Referenzarchitekturmodell Industrie 4.0 [DIN16] als „Verwaltungsschale“ vorgestellt worden. Laut diesem Modell stellt die Schale die Laufzeitdaten, die dazugehörigen Informationen und die Dienste des Komponenten-Managers für jedes Industrie 4.0-Asset bereit. Die Spezifikation schreibt keine feste Verortung der Verwaltungsschale vor. So können die Schalen eines Assets theoretisch auf einem beliebigen oder sogar verteilten IT-System ausgeführt werden. Nichtsdestotrotz ist die Verortung der Funktionalität der Verwaltungsschale oder deren Teile auf dem Asset-Hardwaresystem ein naheliegender und wegen des Zugriffes auf die Laufzeitinformationen oft unabdingbarer Weg.

Weitere besonders hervorzuhebende Kategorien der zusätzlichen Funktionalität sind zum einen die sogenannten Self-X Funktionen, wie Selbstkonfiguration oder Selbstoptimierung, die häufig zwingend auf dem Zielsystem ausgeführt werden müssen. Zum anderen sind Software-Agentensysteme zu erwähnen, die wegen der Voraussetzung der Mobilität einzelner Agenten einen „Lebensraum“ auf dem ausführenden System benötigen.

Zusammenfassend lässt sich sagen: Ressourcen für zusätzliche Funktionalität werden dringend benötigt. Dieses gilt insbesondere für die Laufzeitsysteme auf der Prozessleitebene der Automatisierungspyramide.

Auf den ersten Blick erscheint die Bereitstellung zusätzlicher Ressourcen (insbesondere der Rechenzeit) auf der Prozessleitebene problematisch. Das liegt vor allem an den harten Echtzeitanforderungen an die Laufzeitsysteme. Diese werden häufig aus Gründen der gleichmäßigen Abtastung in einem festen Zyklus betrieben, dabei wird die verfügbare Rechenzeit oft bereits vollständig genutzt bzw. reserviert. Die faktische Ausführungszeit

der Logik kann jedoch aus mehreren, weiter unten aufgeführten, Gründen Fluktuationen unterliegen. Wegen der festen Zykluszeit kann die, aufgrund dieser Fluktuationen eventuell verfügbare, Zeit nicht für prozessrelevante Berechnungen genutzt werden. Somit sind Szenarien möglich, in denen die verfügbare Rechenkapazität des Laufzeitsystems am Ende eines Zyklus ungenutzt bleibt. Diese ungenutzte Zeit wird in dieser Arbeit als Slackzeit bezeichnet und ist insbesondere für die Ausführung zusätzlicher, nicht-echtzeitkritischer Funktionalität vielversprechend.

Im Folgenden werden Gründe für die Entstehung der Slackzeit als Folge der Schwankung bzw. der Überschätzung der Ausführungszeit der Logik aufgezählt:

- **Überabschätzungen:** Worst Case Execution Time (WCET)-Analyse liefert eine obere Schranke für die Laufzeit eines Programms. Diese Schranke kann bei analytischer Abschätzung über der tatsächlich beobachtbaren maximalen Laufzeit liegen.

- **Summierung der Überabschätzungen:** Falls die WCET-Abschätzungen für die einzelnen Komponenten eines Programms vorliegen (z. B. für jeden Funktionsbaustein), so wird im einfachsten Fall die WCET des gesamten Programms durch eine einfache Addition der einzelnen WCET berechnet. Diese Abschätzung lässt jedoch die Abhängigkeiten einzelner Komponenten voneinander außer Betracht.

- **Fluktuationen durch unterschiedliche Ausführungspfade:** Einfache Standardbausteine der IEC 61131-3, wie z. B. Additions- oder Logik-Bausteine, schwanken kaum in ihrer Ausführungszeit. Die Ausführungszeit komplexerer Funktionsbausteine, die vor allem in imperativen Programmiersprachen wie strukturierter Text oder C implementiert sind, unterliegt Schwankungen, die durch unterschiedliche Ausführungspfade einzelner Programme bedingt sind.

  Als Beispiel gilt ein PID-Regler: auch wenn die Bestandteile des Reglers, z. B. ein P-Glied, als einfachste Blöcke erscheinen, so müssen bei einer tatsächlichen Implementierung viele Randfälle betrachtet werden. Dieses erzeugt Verzweigungen im Ausführungspfad. Tatsächlich beinhaltet die Implementierung eines PID-Reglers in der VDI/VDE 3696 (Bausteinklasse „C“) [VDI95] als strukturierter Text mehr als ein dutzend Verzweigungen des Programmablaufs.

- **Optimierung der Ausführung:** In den einzelnen Laufzeitumgebungen sind Optimierungsansätze zu finden, die über die Spezifikation der IEC 61131-3 hinausgehen. Beispielweise existiert in dem Funktionsbausteinsystem des ACPLT/RTE Laufzeitsystems eine Möglichkeit die Bausteine nur bei Änderung ihrer Eingänge auszuführen. Solche Optimierungen sorgen gerade in den stationären Phasen das Prozesses für eine relativ niedrige Auslastung des Laufzeitsystems.

- **Diskretisierung der Systemauslastung:** Zusätzlich zu der Kontrolllogik in der Funktionsbausteinsprache kann in einer Laufzeitumgebung auch prozedurale Logik enthalten sein. Für die Beschreibung der Prozeduren wird die Ablaufsprache der IEC 61131-3 oder eine Statechart-basierte Sprache verwendet. Da einzelne Zustände der Prozedur unterschiedlichste unterlagerte Logik aktivieren bzw. deaktivieren können, führen diese Abläufe zur diskreten Veränderung der Systemauslastung.

  Die diskreten Sprünge der Auslastung werden von den ereignisorientierten Abläufen verstärkt und tragen zu den dynamischen Lastschwankungen bei. In der IEC 61131-3

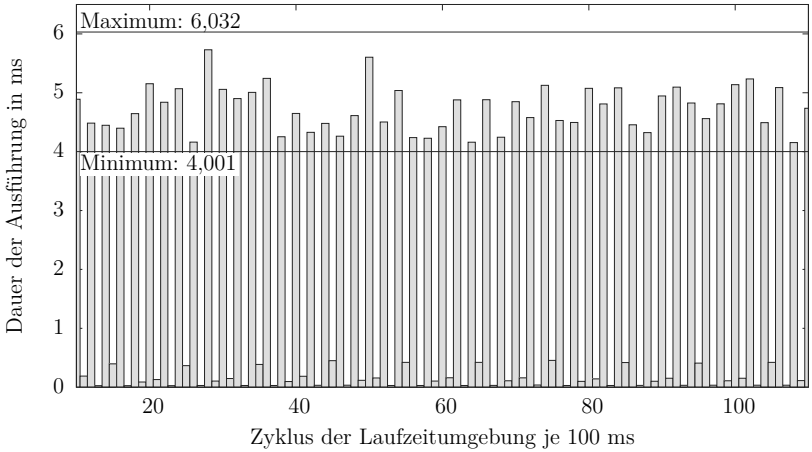


Abbildung 1.1.: Laufzeiten der Kontrolllogik eines Moduls der modularen Anlage M4P.AC inklusive der maximalen und der minimalen Laufzeit des längeren Zyklustyps (gerade Zyklen) innerhalb der kompletten Messreihe.

werden nicht-zyklisch aktivierbare Bausteine erwähnt, die on-demand als Reaktion auf einen physikalischen Eingang ausgeführt werden können.

- **Seltenheit der worst-case Ereignisse:** Falls die WCET sich auf ein seltenes Ereignis des physischen Systems, z. B. auf eine Notsituation, bezieht, lässt sich diese Laufzeit aus Gründen der sicherheitsgerichteten Auslegung der Produktionssysteme praktisch niemals beobachten.

Die Fluktuationen der Ausführungszeit lassen sich empirisch belegen. In Abbildung 1.1 ist die Dauer der Ausführung einer leittechnischen Anwendung in einem der Module der modularen Anlage M4P.AC aufgezeichnet. Das Modul führt während der Aufzeichnung ein einfaches Rezept aus. Das Laufzeitsystem wird dabei von einem Echtzeitbetriebssystem ausgeführt. Es sind zwei Typen von Zyklen erkennbar, die unterschiedliche Laufzeiten aufweisen. Der erste Typ wird in den ungeraden Zyklen der Abbildung dargestellt und terminiert stets innerhalb von 1 ms. Der zweite Typ, der in geraden Zyklen der Abbildung dargestellt ist, weist eine Laufzeit zwischen ca. 4 und 6 ms auf. Diese Laufzeit unterliegt etwas größeren Schwankungen: innerhalb von ca. 18000 aufgezeichneten Zyklen betrug das gemessene Maximum des zweiten Typs ca. 6 ms. Das Minimum hingegen betrug ca. 4 ms. Somit konnten Fluktuationen bis zu 33 % der maximal gemessenen Laufzeit der Logik festgestellt werden.

Die Laufzeitumgebungen verfügen somit über eine Rechenkapazität, die häufig aber nicht garantiert in jedem Zyklus zur Verfügung steht. Diese Kapazität eignet sich zum einen für die Funktionalität, die keinen Echtzeitanforderungen unterliegt. Zum anderen sind auch optional auszuführenden Zusatzfunktionen der Echtzeit-Logik denkbar, deren Ausführung nur in Zyklen mit der ausreichend großer Slackzeit stattfinden darf.

1.2. Zielsetzung

Das Ziel der Arbeit ist die Definition eines neuen Konzepts zur Flexibilisierung des Laufzeitverhaltens der leittechnischen Anwendungen, das eine sichere und effektive Nutzung der Slackzeit der Laufzeitsysteme ermöglicht. Anwendungen mit flexiblem Laufzeitverhalten werden in dieser Dissertation als ressourcenadaptiv bezeichnet.

Die Recherche über die existierenden Konzepte für ressourcenadaptive Systeme aus der Informatik bildet den Ausgangspunkt dieser Arbeit. Die Herausforderung besteht darin, die geeigneten Konzepte und Tools aus den unterschiedlichsten Bereichen der Informatik und der Mathematik, z. B. aus den Bereichen des Software-Engineerings, der Scheduling Theorie, der Optimierung und der formalen Verifikation, zu identifizieren und zu einem an die Randbedingungen der Leittechnik angepassten und realisierbaren Gesamtpaket zu kombinieren. Diese Randbedingungen umfassen nicht nur die notwendigen implementierungstechnischen und konzeptionellen Maßnahmen für die Umsetzung der funktionalen Anforderungen, sondern auch insbesondere die Analyse der domänenspezifischen nichtfunktionalen Anforderungen. Im Bereich der nicht-funktionalen Anforderungen spielen vor allem die minimalen Migrationsaufwände und die Akzeptanz der Nutzer eine besondere Rolle.

Als Ergebnis soll ein Rahmenwerk entstehen, das eine nahtlose Integration von ressourcenadaptiven Konzepten in zyklische, an die IEC 61131-3 angelehnte, Laufzeitumgebungen der Prozessleittechnik ermöglicht. Der Begriff Rahmenwerk suggeriert ein Gerüst aus mehreren Bestandteilen, die die Definition, die Einbettung und das Ausführen ressourcenadaptiver Anwendungen im operativen Kontext ermöglichen.

Für die Integration von ressourcenadaptiven Anwendungen werden folgende Bestandteile bzw. Konzepte benötigt, die das Rahmenwerk zwingend enthalten soll:

- Eine Softwarearchitektur des Laufzeitsystems, die die Koexistenz zwischen den existierenden und ressourcenadaptiven Anwendungen innerhalb einer Laufzeitumgebung ermöglicht.

- Eine domänenspezifische Sprache für die Beschreibung des ressourcenadaptiven Verhaltens von Anwendungen. Da Funktionsbausteinnetzwerke speziell in der Domäne der Automatisierungstechnik stark verbreitet sind, soll die zu erstellende Sprache insbesondere die Ausführung solcher Netzwerke steuern können. Die eingeführte Sprache soll möglichst stark an die etablierten Prozedurbeschreibungssprachen der Automatisierungstechnik angelehnt werden, um eine maximale Akzeptanz bei den Nutzern zu erreichen.

- Ein Scheduling-Modell, das die Ausführung einzelner Anwendungskomponenten zur Laufzeit überwachen und die Slackzeit an die adaptiven Anwendungen dynamisch zuteilen kann. Das Modell soll mit den existierenden zyklischen Laufzeitumgebungen kompatibel sein.

Die Umsetzbarkeit und die Anwendbarkeit des Rahmenwerks soll anhand einer prototypischen Implementierung und Einbettung des Rahmenwerks in ein existierendes Laufzeitsystem bestätigt werden. Dieser Prototyp soll auf industrieller Hardware und anhand synthetischer Tests sowie unterschiedlicher Anwendungsszenarien aus dem Umfeld der Prozessleittechnik evaluiert werden.

1.3. Aufbau der Arbeit

Der Rest dieser Dissertation ist wie folgt aufgebaut:

- Kapitel 2 beleuchtet die für diese Arbeit relevanten Grundlagen. Dazu gehört eine Vorstellung der Aufgaben der Automatisierungs- und der Prozessleittechnik, eine Übersicht über die unterschiedlichen Verfahren für das Scheduling von Echtzeitsystemen sowie eine kurze Einführung in Timed Automata und die gemischt-ganzzahlige Optimierung. Anschließend werden die Begriffe aus dem Umfeld der Laufzeitsysteme und deren Softwarearchitekturen nach IEC 61131-3 und IEC 61499 eingeführt.

- Kapitel 3 stellt den Stand der Wissenschaft und Technik auf dem Gebiet der Flexibilisierung leittechnischer Anwendungen zusammen. Neben den eigenen Vorarbeiten stehen dabei die aktuellen Arbeiten zu Themen der losen Kopplung der Systeme durch Serviceorientierung, die Agentensysteme, die modellgetriebenen Ansätze und die Ansätze zur Laufzeit-Rekonfiguration der Laufzeitsysteme im Mittelpunkt. Abgeschlossen wird das Kapitel mit einer Gegenüberstellung ausgewählter Laufzeitsysteme und Sprachen für die Prozedurbeschreibung. Die Untersuchungen zum Stand der Technik zeigen eine Forschungslücke bezüglich der adaptiven Ausführungszeit der leittechnischen Anwendungen auf.

- In Kapitel 4 werden funktionale und nicht-funktionale Anforderungen an das zu erstellende Rahmenwerk aufgestellt, die aus den domänenspezifischen Anforderungen der Leittechnik abgeleitet werden.

- Kapitel 5 analysiert vielversprechende Kandidaten für das Schließen der aufgezeigten Forschungslücke aus dem Bereich der Informatik unter den Gesichtspunkten der aufgestellten funktionalen und nicht-funktionalen Anforderungen.

- Kapitel 6 beschreibt das entwickelte Rahmenwerk als den Hauptbeitrag dieser Arbeit:
  - Abschnitt 6.1 stellt eine Softwarearchitektur vor, die die gemeinsame Verwaltung der Logik und deren Ausführungsvorschrift innerhalb einzelner Komponenten vorschreibt. Damit wird eine Kapslung erreicht, die die Grundlage für die Integration weiterer Typen der Ausführungssteuerung schafft. Als Beispiel dient die Koexistenz von IEC 61131-3 und IEC 61499 Anwendungen.
  - In Abschnitt 6.2 wird ein Meta-Modell für die Beschreibung des ressourcenadaptiven Verhaltens vorgestellt, das an eine bekannte Prozedurbeschreibungssprache angelehnt ist. Die Semantik des Modells wird formal durch eine Überführung auf Automatennetzwerke des UPPAAL-Toolkits beschrieben.
  - In Abschnitt 6.3 wird ein Referenzmodell eines Schedulers für ressourcenadaptive Anwendungen vorgestellt. Dieser nutzt eine Kombination aus offline und online Scheduling-Verfahren. Das offline Problem wird mithilfe der Methoden der gemischt-ganzzahligen Optimierung aufgestellt und gelöst.

- Kapitel 7 stellt die prototypische Implementierung des Rahmenwerks vor. Das Rahmenwerk wird anhand von vier Use-Cases mit und ohne Prozesseingriff validiert.

- Kapitel 8 schließt die Arbeit mit einer Diskussion der erzielten Ergebnisse und einem Ausblick ab.

2. Grundlagen

In diesem Kapitel werden die benötigten Grundlagen vorgestellt. Die Zielsetzung der Darstellung ist somit eine möglichst kompakte und konsistente Definition der verwendeten Begriffe und Zusammenhänge. Für eine umfassende Einführung in die angesprochenen Themengebiete wird auf die zitierten Quellen verwiesen.

2.1. Allgemeine Grundlagen

2.1.1. Automatisierungstechnik und Prozessleittechnik

Der Begriff „Automatisierung“ bezeichnet laut [LG99] den Prozess „Maschinen, Geräte oder technische Anlagen mit Hilfe von elektrischen, mechanischen, pneumatischen oder hydraulischen Einrichtungen in die Lage zu versetzen, mehr oder weniger selbsttätig zu arbeiten“. Je nach der zu automatisierenden Anlage wird der Begriff der Automatisierung weiter spezialisiert.

Im Bereich der industriellen Produktion haben sich historisch zwei weitgehend getrennte Domänen herauskristallisiert [MBS+11]: die diskrete Fertigung und die Prozesstechnik. Die daraus abgeleiteten spezialisierten Unterdisziplinen der Automatisierung heißen Fertigungsautomatisierung bzw. Prozessautomatisierung. Neben den historisch geprägten Unterschieden der beiden Disziplinen, wie z. B. den Normen und den Begriffssystemen, existiert eine Reihe technischer Differenzierungsmerkmale, die in Tabelle 2.1 zusammengefasst sind.

Als Prozessleittechnik bezeichnet man den Oberbegriff für die Kernaufgaben der Automatisierung (wie z. B. Steuern und Regeln) im Bereich der Prozessautomatisierung [Pol94, LG99, Mey02]. Die Leittechnik bindet den Menschen „in angemessener Art und Weise in seiner Verantwortlichkeit und seinen Werten in das Prozessgeschehen“ ein [ASS97]. Dabei spielt vor allem die Informationsorientierung den entscheidenden Unterschied, denn sie erlaubt die ganzheitliche Betrachtung aller operativen Aufgaben unter den Gesichtspunkten der Integration in die Informationssysteme eines produzierenden Unternehmens. Unter den „operativen Aufgaben“ werden dabei Maßnahmen mit unmittelbaren Auswirkungen im physischen System verstanden [Mey02]. Die informationsorientierte Betrachtungsweise ermöglicht den Zugang zu komplexeren Aufgaben, wie z. B. der Instandhaltung, der Produktverfolgung und dem Engineering. Der Begriff der Informationsorientierung beschreibt auch die Tatsache, dass die Methoden der Informatik und der Informationsverarbeitung immer mehr an Bedeutung für die Leittechnik gewinnen.

Die Systeme und Funktionen bzw. die Aufgaben der Automatisierungstechnik lassen sich in einem Ebenenmodell, der sogenannten Automatisierungspyramide, einordnen. Diese Pyramide ist in Abbildung 2.1 dargestellt. Die Breite der Ebenen stellt sowohl die Hierarchie der Führung, als auch die Anzahl der eingesetzten Systeme dar – ein Unternehmensleitsystem kann eine Anlage mit mehreren tausenden Feldgeräten leiten. Die Übersicht über die

Tabelle 2.1.: Fertigungs- und Prozessautomatisierung im Vergleich (nach [MBS+11, Fel01]).

Merkmal	Fertigungsautomatisierung	Prozessautomatisierung
Aktoren	Motoren	Ventile, Pumpen
Transport	Förderbänder	Pumpen, Kompressoren
Produkt	Festkörper	ungeordnete Menge
Prozesse	bewirkt durch Aktoren	laufen selbstständig
Reaktionszeiten	kurz	mittel bis lang
Bedienen und Beobachten	dezentral	zentral

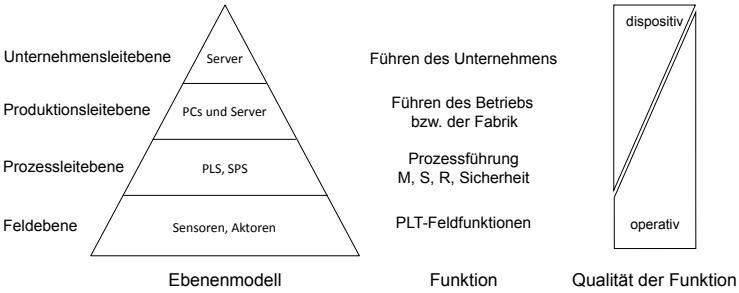


Abbildung 2.1.: Automatisierungspyramide mit eingesetzten Hardware-Plattformen, Funktionen und deren qualitativer Bewertung (angelehnt an [Pol94, Pol85]).

Qualität der Funktion sagt aus, dass die Systeme auf den unteren Schichten zunehmend operativ agieren. Damit werden auch Zeitschranken für den Echtzeitbetrieb zunehmend kleiner.

Die Prozessleitebene ist für die Ausrichtung dieser Arbeit von besonderem Interesse. Die Systeme auf dieser Ebene setzen die Produktionsaufträge übergeordneter Ebenen in operative Realisierungsprozesse um [Pol94], die mittels der angebundenen Sensorik und Aktorik der Feldebene physisch bewirkt bzw. überwacht werden.

Der Einsatz der Prozessleitsysteme (PLS) auf der Prozessleitebene gehört zum heutigen Stand der Technik in der Prozessautomatisierung. Der Haupteinsatzzweck ist unbestritten das Messen-Steuern-Regeln Aufgabenfeld. Das Leitsystem besteht aus den Komponenten die in Abbildung 2.2 dargestellt sind. Das PLS ist ein verteiltes System, dessen Mindestausstattung aus folgenden Komponenten besteht: Die Anzeige- und Bedienkomponente (ABK) ist die Schnittstelle zwischen dem Leitsystem und dem Bediener und ist normalerweise in der Leitwarte installiert. Sie ermöglicht das Überwachen der Anlage und kann für manuelle Eingriffe in die Prozessführung genutzt werden. Die Engineering Workstation (EWS) ist eine dedizierte Software- oder Hardwareeinheit, die zur Konfiguration des PLS genutzt wird. Die prozessnahe Komponente (PNK) bildet die Schnittstelle des Leitsystems zu der Feldebene. Die Komponente enthält in der Regel Input-Output (I/O)-Karten, die konven-

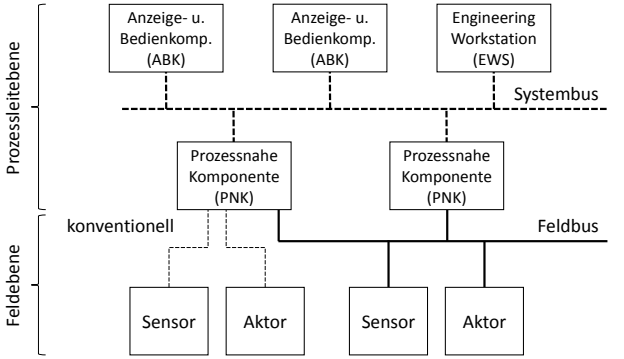


Abbildung 2.2.: Grundstruktur eines Prozessleitsystems (angelehnt an [TM09a, Pol94]).

tionell oder mittels eines Feldbuses mit den Sensoren und Aktoren verbunden sind. Die Kommunikation zwischen den PNKs und anderen Komponenten des PLSs erfolgt mittels eines echtzeitfähigen Systembuses. Die Ausführung der operativen Logik ist das definierende Merkmal einer PNK.

Die PNK stellt die Hardware-Plattform für die ausgeführte Software bereit. Diese besteht aus der Laufzeitumgebung und der darin eingebetteten benutzer- und anlagenspezifischen Anwendung. Eine verbreitete Plattform für die PNK ist die speicherprogrammierbare Steuerung (SPS), die in Abschnitt 2.2.3 detailliert vorgestellt wird. Die primäre Aufgabe einer PNK ist das Ausführen der Steuer-, Regel-, sowie Interlocklogik. Aus diesen Aufgaben lassen sich auch die nicht-funktionalen Anforderungen an die PNKs und die eingebetteten Laufzeitumgebungen ableiten, wie z. B. Echtzeitanforderungen, Anforderungen an die Programmierung durch den typischen Nutzer, Stabilität und Anforderungen an die Kommunikation. Einer genauen Anforderungsanalyse ist Kapitel 4 dieser Arbeit gewidmet.

2.1.2. Echtzeitsysteme und Scheduling

Echtzeitsysteme

Da die Hauptaufgabe der Automatisierungstechnik in der Beherrschung eines physikalischen Systems besteht, spielen die temporalen Eigenschaften des Systems eine kritische Rolle im gesamten Lebenszyklus einer leittechnischen Anwendung. Das korrekte Verhalten des gesteuerten Systems hängt somit nicht nur von der korrekten Ausgabe des steuernden Systems, z. B. von dem richtigen Schaltsignal oder Stellwert, sondern auch von dem Zeitpunkt der Wirksamkeit dieses Signals ab. Solche Systeme werden Echtzeitsysteme genannt. Das kanonische Beispiel für ein Echtzeitsystem ist die Funktionalität der Airbag-Auslösung in einem Kraftfahrzeug. Für die Sicherheit der Insassen ist nicht nur die Tatsache der Auslösung im Falle eines Unfalls, sondern auch der richtige Zeitpunkt dieser von immenser Bedeutung. Es ist ersichtlich, dass sowohl Cyber-Physical Systems (CPS), als auch Cyber-Physical Production Systems (CPPS) aufgrund der Kopplung mit der physikalischen Welt einen Sonderfall der Echtzeitsysteme darstellen (die Definition beider Begriffe folgt im Abschnitt 2.1.3).

Für diese Arbeit wird die Definition eines Echtzeitsystems von Kopetz [Kop11a] genutzt: „Ein Echtzeitsystem (...) muss auf die Stimuli der Umgebung in den, von der Umgebung vorgegebenen, Zeitintervallen reagieren“. Das einfachste Modell eines Echtzeitsystems fordert somit die Existenz einer garantierten oberen Zeitschranke für die Dauer der Reaktion des Systems auf einen Stimulus. Die tatsächliche Dauer der garantierten Antwortzeit spielt dabei keine Rolle und hängt alleine von der Umgebung des Systems und somit von der zu lösenden Aufgabe ab. Es ist hinzuzufügen, dass die Definition keinen Prozessor oder Computer innerhalb eines Echtzeitsystems fordert und somit auch weitere, z. B. mechanische oder elektrische Systeme, abdeckt.

Die Umgebung des Echtzeitsystems besteht notwendigerweise aus dem gesteuerten physischen System. Neben diesem können weitere Echtzeitsysteme Teil der Umgebung sein, z. B. ein Human-Machine Interface (HMI), das mit den Bedienern interagiert und deren Befehle aufnimmt.

Eine Dimension für die Klassifikation der Echtzeitsysteme ist der abstrakte Nutzen der gelieferten Antwort nach dem Ablauf der garantierten Echtzeitschranke (oder Deadline). Hat die Antwort keinen Nutzen, so wird die Schranke eine feste Schranke genannt. Im anderen Fall wird die Schranke als weich bezeichnet.

Die zweite Dimension der Klassifikation sind die Konsequenzen der Nichteinhaltung der garantierten Schranke. Die Konsequenzen können entweder sicherheitskritisch oder nicht sicherheitskritisch sein. Welche Umstände sicherheitskritisch sind, hängt von der spezifischen Aufgabe ab. Typischerweise werden Schäden der eingesetzten Hardware, der Umgebung und, vor allem der Menschen als sicherheitskritisch bewertet.

Sicherheitskritische Systeme mit festen Schranken werden Systeme mit harten Echtzeitbedingungen genannt. Nicht sicherheitskritische Systeme mit weichen Schranken werden als Systeme mit weichen Echtzeitbedingungen bezeichnet. Die Begriffe „hart“ und „weich“ werden somit in Abhängigkeit vom Kontext unterschiedlich gedeutet. Der Entwurf der Systeme mit harten Echtzeitbedingungen unterscheidet sich grundlegend von dem der weichen Echtzeitsysteme [Kop11a]. Die harte Echtzeit erfordert ein garantiertes Systemverhalten des Systems für alle möglichen Zustände des Systems und der Umgebung.

Neben der Erfüllung der charakteristischen Anforderung, der Zusicherung der Echtzeitschranken, können Echtzeitsysteme in Bezug auf weitere Eigenschaften bewertet werden. Die zwei wichtigsten Kriterien für diese Arbeit sind die Vorhersehbarkeit und die Flexibilität. Diese werden wie folgt definiert [BH09]: „Vorhersehbarkeit ist der Grad des Vertrauens darin, dass korrekte, qualitative oder quantitative Prognosen über den Zustand eines Systems gemacht werden können“ und „Flexibilität stellt den Grad der Anpassungsfähigkeit eines Systems an eine neue Umgebung dar“. Weitere Bewertungskriterien für Echtzeitsysteme sind in [BH09] zu finden. Der Begriff Adaptivität wird mit dem Begriff der Flexibilität synonym verwendet. Ein ausführlicher Vergleich der Begriffe Adaptivität und Flexibilität ist in [VMSS07] zu finden.

Aktivierungsparadigmen der Echtzeitsysteme

Im Allgemeinen wird zwischen den zwei grundlegenden Paradigmen der Aktivierung der Echtzeitsysteme unterschieden: den zeit- und den ereignisgesteuerten Systemen [Kop11a, Foh12]. Der Trigger ist in diesem Sinne der Auslöser für eine bestimmte Systemaktion [Kop11a]. Das Aktivierungsparadigma beantwortet die folgenden Fragen: wann werden die Ereignisse erkannt, wer initiiert Aktivitäten des Systems und wann werden die Entschei-

dungen getroffen [Foh12].

In den zeitgesteuerten Systemen hängen alle Aktivitäten von dem Fortschritt der physischen Zeit (wall-clock time) ab. Genauer genommen werden die Aktivitäten an den vordefinierten, periodischen Punkten der Zeit gestartet (den sogenannten Uhr-Ticks). Die äußeren Stimuli des Echtzeitsystems können somit nur während dieser Zeitpunkte verarbeitet werden. Das klassische Beispiel für ein zeitgesteuertes System ist das Softwaremodell einer SPS (vgl. Abschnitt 2.2.3). Die Implementierung eines zeitgesteuerten Systems beinhaltet nur einen CPU-Interrupt – den Timer für die Abstände zwischen den Zyklen.

In den ereignisgesteuerten Systemen werden die Aktivitäten durch „signifikante Ereignisse, die keine Uhr-Ticks sind", [Kop11a] ausgelöst. Signifikante Ereignisse kennzeichnen die Veränderungen der Umgebung des Echtzeitsystems, auf die eine Systemreaktion erfolgen muss [Kop93]. Dieses Verhalten wird durch unterschiedliche CPU-Interrupts des Systems umgesetzt. Als Beispiel für ein ereignisgesteuertes System dienen viele der eingebetteten Systeme die auf, die unterschiedlichen externen Stimuli, z. B. eine Eingangsänderung, direkt über einen Interrupt reagieren können.

Die Diskussionen der Vor- und Nachteile der beiden Ansätze auf der generellen Ebene der Echtzeitsysteme werden seit Jahren geführt [Foh12]. In der Domäne der Automatisierungstechnik finden sich diese zwei unterschiedlichen Paradigmen in den zwei Softwarearchitekturen der Standards IEC 61131 und IEC 61499 wieder, die in Abschnitten 2.2.5 bzw. 2.2.6 genauer vorgestellt werden. Die Vor- und Nachteile der beiden Ansätze sind im Kontext der domänenspezifischen Anforderungen ein Gegenstand der aktiven Diskussion.

Scheduling der Echtzeitsysteme

Aus der Perspektive des Schedulings kann ein Echtzeitsystem als eine Menge der Prozessoren (CPUs), einer Menge der Tasks und der Ressourcen definiert werden [But11]. Zum Zweck dieser Arbeit kann die Existenz eines einzelnen Prozessors (uniprozessor System) angenommen werden. Darüber hinaus ist die Modellierung der Ressourcen nicht erforderlich. Für die Modellierung der Zeit wird ein diskretes Modell mit positiven Zeitpunkten aus $\mathbb{N}_0$ angenommen.

Die Aufgabe des Schedulers ist somit, die Ausführung der Tasks $T_i = T_1, ..., T_n$ auf einem Prozessor unter der Einhaltung bestimmter Bedingungen zu garantieren. Da ein Task auch mehrfach ausgeführt werden kann (z. B. bei zyklischen Tasks), wird zwischen den Tasks und deren Instanzen (Jobs) unterschieden.

Eine grundlegende Bedingung ist das Einhalten der Ausführungsdeadlines für zyklische Tasks. Dazu werden die Tasks als Tupel $T_i = (w_i, per_i, dead_i)$ modelliert. Die k-te Instanz des Tasks T_i, $k \geq 1$, kann ab dem Zeitpunkt $(k-1) \cdot per_i$ ausgeführt werden und muss vor der Deadline $(k-1) \cdot per_i + dead_i$ beendet werden. Der Parameter $dead_i$ bezeichnet die relative Deadline des Jobs bezüglich der Periode. Der Parameter w_i ist die WCET des Jobs. Diese Zusammenhänge sind beispielhaft in Abbildung 2.3 dargestellt.

Ein Schedule ist somit eine Funktion, die die Jobs an die CPU in Abhängigkeit von dem jeweiligen Zeitpunkt zuordnet. Ein Schedule ist zulässig, wenn er die gewünschten Bedingungen erfüllt. Eine Menge der Tasks heißt schedulbar, falls ein zulässiger Schedule existiert. Das zugehörige Entscheidungsproblem wird als Scheduling-Problem bezeichnet.

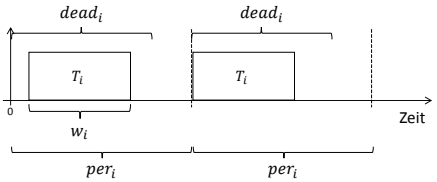


Abbildung 2.3.: Korrekte Ausführung eines zyklischen Tasks T_i über zwei Perioden.

Preemptive und Cooperative Scheduling

Unter „Preemption“ versteht man die technische Möglichkeit die Ausführung eines Tasks zu unterbrechen und die Ressourcen der CPU an einen anderen Task zu übertragen. Dieser Vorgang wird als „Kontextwechsel“ bezeichnet. Die Unterbrechung der Ausführung erfordert keine Kooperation seitens des Tasks und geschieht aus Sicht des unterbrochenen Programms transparent. Scheduler, die Mechanismen der Preemption einsetzen, werden als preemptive oder unterbrechende Scheduler bezeichnet.

Scheduling-Verfahren, die die laufenden Tasks nicht unterbrechen, werden cooperative oder nicht-unterbrechende Scheduler genannt. In diesem Fall kann der Scheduler nur nach der Terminierung des aktuell ausgeführten Tasks aktiv werden.

Die Vorteile der nicht-unterbrechenden Verfahren umfassen [BBY13]:

- **Laufzeit-Effizienz und einfache Vorhersagbarkeit:** Das nicht-unterbrechende Scheduling vermeidet den Aufwand und die Unsicherheiten, die mit dem Kontextwechsel verbunden sind, wie z. B. die Auffrischung der Caches oder das Zurücksetzen der CPU-Pipeline. Die Vergrößerung der gemessenen WCET, alleine wegen der Cache-Auffrischung nach dem Kontextwechsel, kann bis zu 33% betragen [BBY13].

- **Kleinerer Jitter:** Bei einem cooperativen Scheduler gleicht die Ausführungszeit eines Tasks der Differenz des Start- und Endzeitpunktes der Ausführung. Das hat vor allem Vorteile bei dem Entwurf regelungstechnischer Systeme.

- **Keine Mechanismen zur Synchronisation der Prozesse notwendig:** Der gegenseitige Ausschluss der Zugriffe einzelner Tasks auf Ressourcen ist garantiert, somit entfällt die Implementierung der Synchronisationsmechanismen und das Lösen der damit verbundenen Probleme, z. B. der Prioritätsinversion.

Aus der Perspektive der Leittechnik sind alle der oben genannten Aspekte relevant. Systeme mit hoher Vorhersagbarkeit und Anwendbarkeit für die Aufgaben der Regelungstechnik sind für die Aufgaben der Leittechnik bestens geeignet. Der letzte Punkt erleichtert die Programmierung der PNKs, da es zu keinen unerwünschten Effekten durch die Kontextwechsel kommen kann.

Zu den Nachteilen des nicht-unterbrechenden Ansatzes zählen:

- **Fragilität:** Da die Kontrolle über die CPU des Systems komplett an ein Task bis zu seiner Terminierung bzw. die Rückgabe der Kontrolle übergeben wird, kann ein Task nicht nur die anderen Tasks, sondern auch den Scheduler von dem Zugriff auf die

CPU ausschließen und somit das gesamte System monopolisieren. Bestimmte Eigenschaften der Tasks, wie z. B. eine garantierte Terminierung, müssen somit während der Entwicklung bzw. vor der Ausführung sichergestellt sein.

- **Zusicherung der Eigenschaften aller Tasks notwendig:** Als zusätzlicher Aspekt des vorherigen Punktes, muss jeder Task auch die Einhaltung seiner Ausführungszeit garantieren. Somit müssen alle Tasks mit gleicher Sorgfalt analysiert bzw. verifiziert werden. Im Falle von preemptive Scheduling kann ein „fehlerhafter“ Task, z. B. einer, der eine obere Laufzeitschranke überschreitet, durch den Scheduler terminiert werden.

- **Lange Antwortzeit:** Das System ist während der Ausführung eines Tasks „blockiert“ und kann auf keine externe Stimuli reagieren.

- **Komplexität des Scheduling-Problems:** Die Komplexität der nicht-unterbrechenden Scheduling-Probleme, z. B. die Entscheidung, ob für eine Menge der Tasks ein gültiger Schedule existiert, ist im Gegensatz zu unterbrechendem Scheduling für die meisten Problemklassen nicht effizient lösbar [But11]. Die Schedules werden aus diesem Grund entweder durch extensive Suche oder heuristisch erstellt.

Die notwendige Verifikation der ausgeführten Programme macht das nicht-unterbrechende Scheduling für allgemeine Zwecke unattraktiv. Im Falle einer gesicherten Entwicklungs- und Ausführungsumgebung, wie z. B. im Bereich der industriellen Produktion, können aber die Vorteile des Ansatzes die Nachteile übersteigen. Darüber hinaus kommen die Beispiele der Anwendung nicht-unterbrechender Scheduler aus den Bereichen der Sensornetzwerke (z. B. TinyOS [LMP+05]) und der Agentensysteme (z. B. JADE [BPR01]).

Die Kombination der Vorteile beider Ansätze ist durch die Einschränkung der Möglichkeiten für Kontextwechsel möglich, z. B. durch die Angabe erlaubter Zeitpunkte für einen Wechsel. Diese Scheduling-Verfahren werden unter dem Begriff des „Limited Preemptive Schedulings“ zusammengefasst [BBY13].

Scheduling zeitgesteuerter Systeme

In diesem und im nächsten Abschnitt werden die Vor- und Nachteile der Scheduling-Ansätze für beide Aktivierungsparadigmen diskutiert. Die Grundlage für diese Auflistungen bietet die Veröffentlichung von Fohler [Foh12]. Beide Ansätze können sowohl im unterbrechenden, als auch im nicht-unterbrechenden Modus eingesetzt werden.

Zeitgesteuerte Systeme werden durch das Abarbeiten einer Taskliste (oder Tabelle), die vor der Laufzeit des Echtzeitsystems erstellt wurde, gescheduled. Dieser Vorgang wird in den Quellen auch als offline, static oder pre-runtime Scheduling bezeichnet. Die Taskliste beinhaltet die zu dem Zyklusanfang relativen Zeitpunkte oder Intervalle der Ausführung einzelner Tasks bzw. Jobs. Im Falle einfacher zyklischer Systeme mit einem Prozessabbild, wie z. B. einer SPS, wird nur die Reihenfolge der Jobs benötigt, da es keine blockierenden Ressourceninteraktionen innerhalb des Systems gibt. Das gesamte Wissen über die Systemarchitektur, z. B. die Anzahl und Parameter der ausgeführten Tasks, sowie die Eigenschaften des kontrollierten physischen Systems, wie z. B. für die Wahl der Zykluszeit, wird für die Erstellung der Tabelle benötigt.

Die Vorteile des Schedulings der zeitgesteuerten Systeme beinhalten [Foh12, Kop11a]:

- **Determinismus:** Durch die zyklische Abarbeitung der Taskliste werden die einzelnen Jobs in unveränderter Reihenfolge in jedem Zyklus ausgeführt. Auch der genauere Zeitpunkt der physischen Wirksamkeit der geschriebenen Ausgänge (das „Erscheinen“ der Werte am physischen Ausgang des Systems) lässt sich bestimmen. Diese Eigenschaften hängen nicht von den äußeren Einflüssen auf das System ab und definieren somit ein vorhersagbares Verhalten. Dieser Vorteil erlaubt einfache Berechnungen der nächsten Ausführungszeit jedes Tasks und somit der maximalen Wartezeit zwischen den Ausführungen. Deshalb sind auch einfache Abschätzungen der Reaktionszeit des gesamten Systems auf äußere Stimuli möglich.

- **Konstruktive Erstellung der Taskliste:** Die Taskliste kann nach deren Erstellung in Bezug auf die Einhaltung der geforderten Eigenschaften überprüft werden. Die Komplexität der Validierung ist in der Regel gering und umfasst im einfachsten Fall nur die Sicherstellung der Zeitschranken (vgl. Abschnitt 6.3.3). Der Algorithmus zur Erstellung einer Taskliste muss somit nur eine gültige Lösung finden, die den geforderten Bedingungen genügt.

- **Komplexe Randbedingungen möglich:** Da die Taskliste in einer Phase vor der Laufzeit erstellt wird, können komplexe Bedingungen in Bezug auf die Eigenschaften der Liste bei deren Erstellung berücksichtigt werden. Diese Eigenschaften können Reihenfolgebeziehungen der Tasks, deren Phase, sowie die Regelung der Ressourcenzugriffe umfassen. Die Algorithmen zur Konstruktion der Tasklisten können in der Regel auch weitere Randbedingungen problemlos aufnehmen, da diese normalerweise auf der Lösung eines Optimierungsproblems basieren, die eine globale Suche im Lösungsraum impliziert (die möglichen Randbedingungen für den entwickelten Komponentenscheduler werden in Abschnitt 6.3.2 vorgestellt).

- **Minimaler Laufzeitaufwand:** Durch die Tatsache, dass die Scheduling Taskliste vor der Laufzeit erstellt werden kann, beschränkt sich der Laufzeit-Aufwand auf wenige einfache Operationen wie das Nachschlagen des nächsten Eintrages oder das Ausrechnen einiger Differenzen zwischen bestimmten Zeitpunkten. Die Anzahl dieser Operationen und deren Komplexität hängen nicht von der Komplexität der Randbedingungen, die bei der Erstellung die Taskliste berücksichtigt wurden, ab.

- **Einfachere Implementierung:** Dieser Vorteil folgt direkt aus dem letzten Punkt, denn die Implementierung des Scheduler muss nur die oben erwähnten Operationen umfassen. Durch die einfachere Implementierung ist zu erwarten, dass Scheduler für zeitgesteuerte Systeme auch weniger anfällig für Implementierungsfehler sind und eine einfache Fehlersuche ermöglichen.

- **Einfacheres Testen:** Die zyklischen Systeme sind einfacher zu testen, da nur alle Szenarien innerhalb eines Zyklus getestet werden müssen [Kop11a].

Die Nachteile eines solchen Ansatzes umfassen:

- **Berechnungskomplexität und Komplexität des benötigten Wissens:** Für das Aufstellen der Taskliste wird das komplette Wissen über das System benötigt. Dazu gehören die Auflistung der ausgeführten Tasks, deren Parameter, wie z. B. die Periode, und vor Allem eine Abschätzung der Laufzeit der abgeleiteten Jobs. Diese

Parameter sind nicht immer zu dem Zeitpunkt des Entwurfs des Systems bekannt oder mit hohem Aufwand ermittelbar.

Darüber hinaus ist allein das Aufstellen einer Probleminstanz zur Erstellung einer Taskliste komplex, da alle Randbedingungen berücksichtigt werden müssen. Das Problem der konstruktiven Erstellung einer Taskliste ist typischerweise nicht effizient (d. h. in polynomieller Zeit) lösbar. In der Praxis werden die Tasklisten über eine Überführung des Problems auf ein Standardproblem der Optimierung gelöst, z. B. auf ein Integer Linear Program (ILP). Falls aber die Taskliste selten (im Extremfall nur einmal) erstellt bzw. angepasst wird, ist die für die Erstellung benötigte Rechenzeit vernachlässigbar.

- **Größe der Taskliste:** Die Taskliste enthält Information über die Ausführung einzelner Tasks bzw. der abgeleiteten Jobs. Die Menge der Information hängt vom gewählten Algorithmus ab und variiert von den genauen Zeitpunkten der Ausführung einzelner Jobs, bis zu einer groben Zuordnung der Jobs zu dem jeweiligen Systemzyklus. Eine Taskliste für zyklische Tasks muss über die Länge einer Hyperperiode aufgestellt werden. Diese ist das kleinste gemeinsame Vielfache (kgV) der Perioden einzelner Tasks. Die Ausführung der Jobs innerhalb der Hyperperiode bleibt unverändert, d. h. die Abarbeitung der Taskliste erfolgt zyklisch.

  Die Größe der Taskliste hängt somit von der Länge der Hyperperiode und der benötigten Information zur Ausführung einzelner Jobs ab. Gerade bei Systemen mit eingeschränkten Speicherressourcen kann diese Größe zum kritischen Punkt werden. Um das Anwachsen der Länge der Hyperperiode bei der ungünstigen Wahl der Perioden einzelner Tasks (z. B. als Primzahlen) vorzubeugen, können harmonische Perioden der Tasks eingesetzt werden. Dies bedeutet, dass die Periode jedes Tasks ein ganzes Vielfaches jeder kürzeren Periode eines anderen Tasks ist. Die Information zur Ausführung des Jobs kann durch zusätzliche Annahmen über das Laufzeitsystem reduziert werden. Bei der nicht-unterbrechenden Ausführung der Logik innerhalb einer SPS ist die Information über die Zuordnung eines Jobs zu dem jeweiligen Systemzyklus und innerhalb der Hyperperiode sowie deren Reihenfolge bereits ausreichend.

- **Fehlende Flexibilität:** Ein Hauptnachteil des offline Schedulings ist die fehlende Flexibilität. Das Hinzufügen neuer Tasks oder eine Änderung der Parameter bestehender Tasks erfordert die (teilweise komplexe) Neuberechnung der Taskliste und deren Austausch auf dem Laufzeitsystem. Es existieren allerdings Ansätze, wie man trotz rigider Struktur der Taskliste zusätzliche Aktivitäten durchführen kann (vgl. Abschnitt 5.1.6). Dazu zählt z. B. das Ausführen sporadischer Tasks zur Laufzeit.

- **Konservative Abschätzung:** Die Berechnung der Taskliste erfolgt typischerweise unter der Berücksichtigung der WCETs der einzelnen Tasks. Bei einer früheren Terminierung eines Jobs kann das offline System die verfügbare Rechenzeit nicht verwerten. Dieser Nachteil kann teilweise behoben werden, indem zusätzliche Aktivitäten des Schedulers zur Laufzeit des Systems stattfinden (vgl. Abschnitt 5.1.6 bzw. Abschnitt 5.1.5).

Scheduling ereignisgesteuerter Systeme

Für ereignisgesteuerte Systeme ist das Aufstellen einer offline Taskliste praktisch unmöglich, da die Ereignisse in beliebiger Reihenfolge und zu beliebigen Zeitpunkten Aktivitäten des Echtzeitsystems auslösen können. Aus diesem Grund werden im Bereich der ereignisgesteuerten Systeme online Scheduler eingesetzt, also solche, die die Entscheidung über das nächste auszuführende Task während der Systemlaufzeit treffen.

Bei dem Vergleich der beiden Scheduling-Ansätze steht das Schedulen von zyklischen Tasks im Mittelpunkt. Die Nutzung des ereignisgesteuertes Schedulings für aperiodische Tasks ist zunächst nicht im Fokus, da das einhalten der Echtzeitanforderungen dafür zusätzliche Annahmen über Tasks benötigen wie z. B. über deren Auftrittshäufigkeit.

Es kann offline überprüft werden, ob eine bestimmte Menge der Tasks von einem online Scheduler fehlerfrei, d. h. z. B. unter der Einhaltung der Deadlines, für jeden Job gescheduled werden kann. Eine solche Analyse wird „schedulability Analysis“ genannt. Im Gegensatz zu einer vorberechneten Taskliste ist diese nicht konstruktiv. Stattdessen wird das Verhalten des Schedulers anhand bestimmter Kriterien der Tasks, z. B. der Utilization-Faktoren, abgeschätzt.

Die Vorteile des ereignisgesteuerten Schedulers können wie folgt zusammengefasst werden [Foh12]:

- **Flexibilität:** Da die Entscheidungen zur Laufzeit getroffen werden, kann der Scheduler auf die Änderungen der Menge der Tasks bzw. der Parameter einzelner Tasks entsprechend reagieren. Der Algorithmus kann insbesondere auf die variierenden Laufzeiten einzelner Jobs reagieren und die eventuell verfügbaren Ressourcen nutzen.

- **Marktdurchdringung:** Die meisten Echtzeitsysteme folgen dem ereignisgesteuerten Ansatz [Kop11a]. Die vorhandenen Algorithmen und deren Implementierungen sind über Jahre gereift und werden in unzähligen Systemen eingesetzt.

- **Vorteile bei der Speichernutzung bei großer Anzahl von Tasks:** Der für die Implementierung des Schedulers und dessen Datenstrukturen verwendete Speicher ist bei einer genügend großen Menge der Tasks kleiner als die explizite Speicherung der Taskliste.

Zu den Nachteilen des ereignisgesteuerten Schedulers zählen:

- **Komplexität der Implementierung und höherer Laufzeitaufwand:** Die Implementierung eines online Schedulers ist komplexer als die des offline Ansatzes. Da ein online Algorithmus alle gültigen Taskmengen unter allen Bedingungen korrekt schedulen muss, ist die Verifikation der Korrektheit und Analyse eines solchen Verfahrens nichttrivial. Ferner steigt die Menge der Operationen, die zur Laufzeit durchgeführt werden müssen – das ist der Tradeoff zwischen der Laufzeit und dem Speicherbedarf im Vergleich zu einer vorberechneten Taskliste.

- **Nur einfache Randbedingungen für die Taskmenge:** Online Verfahren können nur einfache Bedingungen für die Tasks berücksichtigen, z. B. gegenseitiger Ausschluss. Darüber hinaus erfordert das Hinzufügen neuer Bedingungen die Entwicklung neuer Algorithmen und der dazugehörigen schedulability Tests.

- **Beschränkte Vorhersagbarkeit:** Auch wenn die Einhaltung der Deadlines durch den schedulability Test garantiert werden kann, kann der genaue Ausführungszeitpunkt eines einzelnen Jobs im Voraus nicht bestimmt werden.

Ein Beispiel für ein Scheduling Verfahren ist das Earliest Deadline First (EDF) Verfahren [LL73]. Der Algorithmus führt bei jeder Aktivierung (z. B. das Erreichen eines neuen Tasks) den Job aus, dessen Deadline am nächsten ist. Das Verfahren ist im unterbrechenden Modus auf einem uniprozessor System optimal. Das heißt, es findet für jede schedulbare Taskmenge einen Schedule, der die Deadline-Bedingungen der Tasks nicht verletzt. Für periodische Tasks mit $per_i = dead_i$ ist der schedulability Test für EDF recht einfach. Die Menge der Tasks kann gescheduled werden, genau dann wenn:

$$U = \sum_{i=1}^{|T|} \frac{per_i}{dead_i} \leq 1.$$

Der Bruch wird der Utilization-Faktor eines Prozesses, die Summe U der Utilization-Faktor der Taskmenge genannt. Da die Priorität eines Tasks vom Abstand zur nächsten Deadline und somit vom Zeitpunkt der Aktivierung des Schedulers abhängt, wird EDF der Klasse der Verfahren mit dynamischen Prioritäten zugeordnet.

2.1.3. CPS und CPPS

In den letzten Jahren erhalten die Begriffe der CPS [KRS12] und deren Anwendung in der industriellen Produktion unter dem Sammelbegriff CPPS [Mon14] Einzug in die Domäne der Automatisierungstechnik.

Der Begriff der CPS in der Definition von Lee [Lee06] beinhaltete zunächst nur die bidirektionale Interaktion eines Rechners oder eines eingebetteten Systems (cyber system) mit einem physischen System (physical system). Aus der kybernetischen Sicht erfüllt dabei in der Regel das cyber System die Rolle des steuernden und das physische System die Rolle des gesteuerten Systems. Diese Aspekte beinhalten faktisch das Selbstverständnis klassischer eingebetteter und automatisierungstechnischer Systeme der letzten Dekaden.

Für einen qualitativen Sprung sind jedoch weitere Eigenschaften des Systems notwendig [KRS12]. Dazu zählen die Kopplung über geschlossene und offene Netze und eine domänenübergreifende Funktionalität.

Die Zielsetzung dieser Arbeit für die Gestaltung eines flexiblen Rahmenwerks zur besseren Nutzung verfügbarer Rechenressourcen spricht diese Punkte explizit an. Durch ein adaptives System können z. B. echtzeitfähige und nicht-echtzeitfähige Anwendungen bzw. Kommunikationsprotokolle kombiniert werden. Darüber hinaus versteht sich das Rahmenwerk als ein Ergebnis der Integration vielfältiger Konzepte aus den Bereichen der Informatik und der Automatisierungstechnik.

2.1.4. Timed Automata und Model-Checking

Die Zeit kann auf zwei fundamental unterschiedliche Arten modelliert werden. Die erste Möglichkeit – die diskrete Zeitmodellierung – stellt die Zeit als Folge ganzzahliger Zeitpunkte dar. Die Ereignisse in diesem Modell können nur zu diesen Zeitpunkten auftreten. Ein Vorteil dieser Vorgehensweise ist die Möglichkeit des Rückgriffes auf andere diskrete

formale und semi-formale Modelle und somit deren einfache Erweiterung auf das Zeitverhalten des zu untersuchenden Systems.

Die zweite Alternative ist die kontinuierliche Zeitmodellierung. Diese Vorgehensweise betrachtet die Zeit als einen kontinuierlichen, linear wachsenden Wert. Das kontinuierliche Modell ist näher an den Systemen der physischen Welt, die Ereignisse zu beliebigen Zeitpunkten auslösen können. Da die ganzen Zahlen Teilmenge der reellen Zahlen sind, kann jedes diskrete Zeitmodell als ein Sonderfall eines kontinuierlichen Zeitmodells mit entsprechender Diskretisierung, z. B. mittels Abtastung, betrachtet werden.

Timed Automata (TA) [AD94] sind ein formales Modell für die Abbildung der Systeme mit einer endlichen Anzahl diskreter Zustände unter der Berücksichtigung eines kontinuierlichen Zeitmodells. Dabei wird das klassische Model eines endlichen Automaten durch eine Reihe reellwertiger, linear wachsender Variablen – der Uhren – erweitert. Eine triviale Rücktransformation eines solchen Modells zu einem endlichen Automaten ist nicht möglich, da in der Zeitdimension unendlich viele Zeitpunkte existieren.

Die praktische Handhabbarkeit der Timed Automata wird durch Modellierungsplattformen erreicht, die auf dem Formalismus der TA basieren. Das zugrunde liegende Modell wird dabei durch zusätzliche syntaktische Elemente erweitert. Diese tragen zur Erhöhung der Nutzerfreundlichkeit und einer kompakten Darstellung der Modelle bei.

Die in dieser Arbeit eingesetzte Werkzeug-Plattform UPPAAL verwendet Netzwerke von TA als Modellierungssprache [BDL04]. Diese Netzwerke bestehen aus mehreren TA, die über sogenannte binäre Kommunikationskanäle kommunizieren bzw. synchronisiert werden können. Durch die Kanäle kann ein Automat in mehrere kleinere orthogonale Automaten unterteilt und somit kompakt dargestellt werden. Darüber hinaus wurden die Automaten um weitere Elemente, wie z. B. Konstanten, ergänzt.

Neben der Möglichkeit Modelle mithilfe von TA aufzustellen, bietet UPPAAL die Option den integrierten symbolischen Model Checker zu nutzen, um bestimmte Eigenschaften des Modells zu beweisen bzw. zu widerlegen. Die Erfüllung der in einer Abfragesprache formulierten Eigenschaften durch das TA wird mithilfe formaler Methoden zugesichert bzw. durch die Generierung eines Gegenbeispiels widerlegt. Die in UPPAAL genutzte Abfragesprache ist eine Untermenge der Timed Computation Tree Logic (TCTL) [BDL04].

2.1.5. Methoden der gemischt-ganzzahligen Optimierung

Ziel der mathematischen Optimierung ist das effiziente Bestimmen eines Extrempunktes einer vorgegebenen Güterfunktion unter der Berücksichtigung gegebener Nebenbedingungen. Methoden der mathematischen Optimierung werden in vielen Disziplinen als Grundbausteine für die Abbildung und Lösung domänenspezifischer Probleme eingesetzt. Als Beispiel dafür dienen die Anwendungen in der Logistik und Produktionsplanung oder der Berechnung von Tasklisten für das Scheduling der Echtzeitsysteme.

Die Grundform eines gemischt-ganzzahligen Optimierungsproblems ist wie folgt definiert [Kal12]: Die Entscheidungsvariablen werden durch zwei Vektoren $x^T = (x_1, \cdots, x_{n_c})$ und $y^T = (y_1, \cdots, y_{n_d})$ repräsentiert. Vektor x enthält die reellen, y die ganzzahligen Entscheidungsvariablen. Darüber hinaus enthält das Problem eine Menge der Gleichungsnebenbedingungen $h(x, y) = 0$ und der Ungleichheitsnebenbedingungen $g(x, y) \geq 0$. Die Vektorgleichungen $h(x, y) = 0$ und $g(x, y) \geq 0$ werden dabei komponentenweise gelesen.

Das Optimierungsproblem besteht darin, ein Minimum (x', y') einer Gütefunktion $J(x, y)$ (jedes Maximierungsproblem lässt sich in ein Minimierungsproblem überführen) durch die

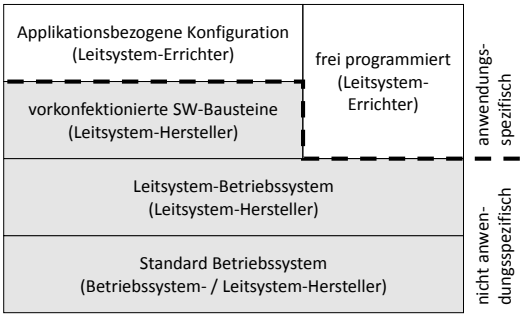


Abbildung 2.4.: Softwaremodell eines PLS mit Angabe des Verantwortlichen [KM09, NAM02].

Belegung der Entscheidungsvariablen zu finden, sodass $h(x', y') = 0$ und die Ungleichheitsnebenbedingungen $g(x', y') \geq 0$ erfüllt sind.

Optimierungsprobleme können nach der Form der Gütefunktion, der Gleichungen der Nebenbedingungen und den verfügbaren Entscheidungsvariablen klassifiziert werden. Zum Beispiel sind die linearen Probleme eine für die Praxis relevante Unterklasse der Optimierungsprobleme. In diesem Fall sind sowohl die Gütefunktion als auch die Nebenbedingungen linear. Die Gütefunktion kann in diesem Fall als Vektormultiplikation und die Vektorgleichungen der Nebenbedingungen als Matrixmultiplikation dargestellt werden.

Das Optimierungsproblem heißt ganzzahlig, falls es ausschließlich ganzzahlige Entscheidungsvariablen enthält, d. h. $n_c = 0$. Sind Entscheidungsvariablen beider Klassen vorhanden, d. h. $n_c > 0$ und $n_d > 0$, heißt das Problem gemischt-ganzzahlig.

Ganzzahlige und gemischt-ganzzahlige Probleme haben generell eine höhere Komplexität als entsprechende Probleme mit ausschließlich reellen Variablen. So ist z. B. die Klasse der linearen Probleme mit reellen Variablen (LP) in polynomieller Zeit lösbar, während die entsprechenden ganzzahligen Probleme (ILP) und somit auch die gemischt-ganzzahligen Probleme (MILP) NP-hart (d. h. mit hoher Wahrscheinlichkeit nur mit exponentiellem Aufwand lösbar) sind.

2.2. Laufzeitsysteme der Prozessleittechnik

2.2.1. Laufzeitsysteme

Eine allgemeine Definition des Begriffes „Laufzeitsystem“ (alternativ Laufzeitumgebung oder runtime environment) ist in [App90] zu finden. Ein Laufzeitsystem implementiert demnach grundlegende Konzepte der für die Programmierung verwendeten Programmiersprache. Die Definition umfasst die Rolle des Laufzeitsystems als Schicht zwischen dem Betriebssystem und der von dem Anwender geschriebenen Anwendungen.

Eine Einordnung des Laufzeitsystems bietet die Abbildung 2.4 (in der Abbildung wird das Laufzeitsystem als „Leitsystem-Betriebssystem“ bezeichnet). Das Laufzeitsystem baut normalerweise auf einem Standard-Betriebssystem auf, das normalerweise bestimmte Echtzeitanforderungen erfüllt. Es existieren auch Anwendungen in denen Laufzeitsysteme die

Aufgaben des Betriebssystems übernehmen und direkt auf der Hardware residieren (baremetal Anwendungen). „Nach oben“ bieten Laufzeitsysteme somit eine Schnittstelle für vorkonfigurierte und anwendungsspezifische Programme. Aus der Sicht der Automatisierungstechnik wird das Softwaresystem einer PNK als das Laufzeitsystem bezeichnet. Das wichtigste Merkmal einer PNK ist der *operative Eingriff* in das kontrollierte System. Vorgänge innerhalb einer PNK haben unmittelbare Folgen im physischen System. Dabei spielt die verwendete Hardwareplattform eine zweitrangige Rolle.

In [App90] werden die Standardbibliotheken einer Programmiersprache (z. B. die Menge der standardisierten Funktionsbausteine aus der IEC 61131-3 [IEC11] oder die Funktionen der C Standard-Bibliothek) nicht zum Laufzeitsystem dazu gezählt, da diese meistens in der Programmiersprache selbst implementiert sind. In dieser Arbeit wird auf diese Unterscheidung verzichtet und die Implementierung der Standardfunktionen einer Programmiersprache zum Funktionsumfang eines Laufzeitsystems hinzugezählt.

Die typischen Funktionen eines allgemeinen Laufzeitsystems (eines, das nicht auf die Leittechnik zugeschnitten ist) umfassen:

- **Speicherallokation und -Freigabe:** Die Mechanismen der Speicherverwaltung des Betriebssystems werden auf die Abstraktion der jeweiligen Programmiersprache abgebildet, z. B. durch einen Garbage Collector für Programmiersprachen ohne explizite Speicherverwaltung wie Java. Der Speichermanagement erfolgt in enger Zusammenarbeit mit dem unterliegenden Betriebssystem.

- **Abstraktion weiterer Betriebssystemfunktionen:** Beispielsweise für den Zugriff auf eine Echtzeituhr.

- **Abstraktion und Unifikation der Hardware:** Zum Beispiel I/O Handling oder Anbindung an einen Feldbus.

- **Speicherpersistenz:** Die Speicherung des Systemzustands, insbesondere der Werte der Variablen sowie der erstellten Objektinstanzen über den Neustart des Laufzeitsystems hinweg. Dieses kann beispielsweise durch eine Synchronisation des Arbeitsspeichers mit einem nichtflüchtigem Speicher, z. B. einer Festplatte, realisiert werden.

- **Profiling und Debuggen:** Das Laufzeitsystem bietet Schnittstellen für den Zugriff auf die aktuellen Performance-Indikatoren sowie Möglichkeiten der Feineinstellung der Systemparameter zwecks Fehlersuche oder Optimierung des Systemverhaltens.

- **Mechanismen der Introspektion und der Reflexion:** Das Konzept der Introspektion bezeichnet die Selbstauskunft über bestimmte Aspekte des Aufbaus und des aktuellen Zustands des Systems durch das System selbst. Die Reflexion erweitert die Introspektion durch die Möglichkeiten der dynamischen Strukturänderung des Systems. Das Auslösen der Selbstauskunft kann entweder lokal oder über eine Kommunikationsschnittstelle erfolgen. Ein Meta-Modell wird dabei für die Modellierung der Systemaspekte eingesetzt und macht das Softwaresystem „self-aware“ [BMR+96].

Eine Laufzeitumgebung muss nicht zwingend alle der aufgelisteten Funktionalitäten anbieten. Beispielsweise gehören im Bereich der Leittechnik die Mechanismen der Introspektion noch nicht zum Stand der Technik. Die Notwendigkeit der Selbstbeschreibung des Systems für die Umsetzung von Meta-Modell-basierten Kommunikationsprotokollen, wie OPC UA,

erhöht allerdings den Druck auf die Hersteller der Laufzeitsysteme in Bezug auf diese Funktionalität.

2.2.2. Models@run.time

Die am Ende des Abschnitts 2.2.1 erwähnten Mechanismen der Introspektion und der Reflexion eines Laufzeitsystems können zu dem Konzept des Models@run.time (alternativ auch als Models@runtime bezeichnet) erweitert werden. Die Reflexion agiert im Kontext einer Programmiersprache und ist damit im großen Maße an die Implementierung der ausgeführten Anwendung gebunden. Im Gegensatz dazu existiert der Models@run.time Ansatz im Problemraum. Ein Beispiel dafür ist ein Regelkreis auf einem Rohrleitungs- und Instrumentenfließschema (R&I Fließschema) und nicht seine Umsetzung mittels Bausteintechnik. Somit sind Models@run.time weitgehend unabhängig von der Implementierung der Anwendung [BBF09]. Durch eine höhere Abstraktion des Modells und somit der Objekte der Selbstbeschreibung, weicht der Ansatz zunehmend die Grenzen zwischen den Engineering- und den Laufzeitmodellen auf.

2.2.3. Speicherprogrammierbare Steuerungen

Die speicherprogrammierbaren Steuerungen (SPSen) oder Programmable Logic Controllers (PLC) gehören zu einer der wahrscheinlich wichtigsten Klasse der PNK-Hardware-Plattformen für den Einsatz in der industriellen Automatisierungstechnik. Historisch gesehen entstanden die SPS als „überschaubarer Ersatz für einfache verbindungsprogrammierte Steuerungen, die vorher mit Schütz- und Relaistechnik (...) realisiert worden waren“ [TM09b]. Heute sind SPSen in unterschiedlichsten Leistungsklassen am Markt verfügbar und decken unterschiedlichste Aufgaben sowohl im Bereich der Fertigungs- als auch der Prozessautomatisierung ab.

Die speicherprogrammierbaren Steuerungen werden hauptsächlich auf der zweiten Ebene der Automatisierungspyramide eingesetzt, in der die einfachen Automatisierungsfunktionen für die gesteuerten Prozesse bzw. die gesteuerten Maschinen mittels der SPSen implementiert sind (die sogenannte Basisautomatisierung).

Die Grundstruktur einer SPS ist in der Norm IEC 61131-1 [IEC03a] beschrieben und kann vereinfacht in Abbildung 2.5 dargestellt werden. Die wichtigsten drei Bereiche der Grundstruktur umfassen die Schnittstellen für die Aktoren bzw. Sensoren, welche die Verbindung zum gesteuerten System ermöglichen. Diese I/O-Anbindung kann entweder modular über spezielle I/O-Karten oder über Remote-I/O-Geräte realisiert werden. Remote-I/Os werden normalerweise über ein Bussystem mit der SPS verbunden, z. B. PROFIBUS PA.

Die Signalverarbeitungsfunktionen beinhalten die internen Funktionen der SPS, die Funktionen des Betriebssystems und die vom Anwendungsprogrammierer definierten anwendungsspezifischen Programme, die in den Sprachen der IEC 61131-3 beschrieben sind.

Die dritte Funktionsgruppe in Abbildung 2.5 sind die Kommunikationsfunktionen. Diese ermöglichen zum einen die Kommunikation mit weiteren Systemen, z. B. SPSen, zum anderen werden die Schnittstellen zu dem Bediener bzw. dem Anwendungsprogrammierer bereitgestellt. Die Kommunikation zu den weiteren Systemen ist in der Norm IEC 61131-5 spezifiziert [IEC01] und bildet die Grundlage für eine dezentrale Automation.

Die physische Ausführung der meisten SPSen erfüllt die Anforderungen an die PNKs, z. B. in Bezug auf Elektromagnetische Verträglichkeit oder den lüfterlosen Betrieb. Somit

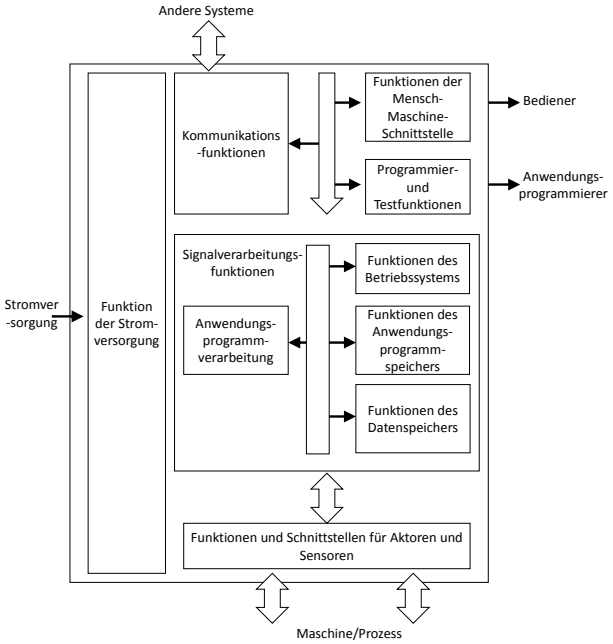


Abbildung 2.5.: Funktionale Grundstruktur eines SPS-Systems nach IEC 61131-1 [IEC03a].

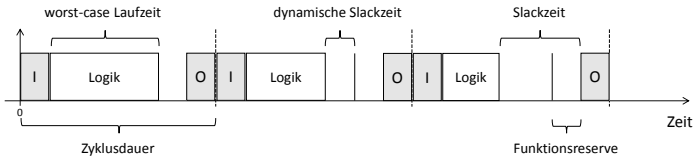


Abbildung 2.6.: Zyklischer Betrieb eines SPS-ähnlichen Systems.

können SPSen auch nah am physischen Prozess eingesetzt werden.

Ein weiteres wesentliches Merkmal einer SPS ist der zyklische Betrieb (engl. cyclc scan mode). Die in der SPS beinhaltete Signalverarbeitungslogik wird ununterbrochen in einem Zyklus ausgeführt. Das heißt, dass jeder Zyklus aus folgenden Aktivitäten besteht [TM09b]:

- **Einlesen der Eingänge:** Die Eingänge werden in einen Puffer, das Prozessabbild, kopiert und bleiben während des Zyklus unverändert.
- **Abarbeitung der Programmlogik:** Die Programmlogik operiert während der Ausführung auf dem konstanten Prozessabbild sowohl für die Eingabe- als auch für die Ausgabeparameter.
- **Schreiben der Ausgänge:** Erst am Ende des Zyklus werden die Ausgangsvariablen des Prozessabbildes auf die physischen Ausgänge der SPS übertragen.

Die worst-case Reaktionszeit der SPS, d. h. die Zeit bis eine Änderung des Eingangs eine Änderung des Ausgangs bewirkt, kann sich somit auf die doppelte Zykluszeit belaufen.

Der zyklische Betrieb einer SPS ist in Abbildung 2.6 dargestellt. Aus Gründen der Gleichmäßigkeit der Abtastung wird vor allem in der Prozessautomatisierung die Dauer des Grundzyklus konstant gehalten. In den meisten Fällen wird die CPU der SPS dabei zwischen dem Ende der Ausführung der Logik und dem Beginn der Ausgabe nicht genutzt (unter der Annahme eines single-task Systems). Diese Zeit wird die Slackzeit (engl. slack time) genannt und ist für die Zielsetzung dieser Arbeit von besonderer Bedeutung (vgl. Abschnitt 1.1). Es wird zusätzlich zwischen der Funktionsreserve und der dynamischen Slackzeit unterschieden. Die Funktionsreserve oder statische Slackzeit entsteht aufgrund der konservativen Auslegung des Systems und ist grundsätzlich in jedem Zyklus vorhanden. Die dynamische Slackzeit entsteht aufgrund der Schwankungen der Laufzeit der Programmlogik, somit variiert deren Ausmaß zwischen den einzelnen Grundzyklen des Systems. Die Slackzeit im Allgemeinen ist daher die Summe aus der Funktionsreserve und der dynamischen Slackzeit in jedem Zyklus.

2.2.4. IEC 61131-3 Sprachen – die Linguae francae der Automatisierung

Die Norm IEC 61131-3 [IEC11] definiert fünf Programmiersprachen, die für die Definition der Logik innerhalb der Program Organization Units (POUs) genutzt werden.

Die Sprachen der IEC 61131-3 (insbesondere die graphischen) sind universelle, weit akzeptierte Programmier- bzw. Modellierungssprachen für die vielfältigen Anwendungen der Automatisierungs- bzw. der Leittechnik. Aus diesem Grund werden neue Konzepte, wenn

möglich, auf diese Sprachen abgebildet, um eine leichte Zugänglichkeit für die Endnutzer zu gewährleisten. Als Beispiel dient die Darstellung von Regeln zur Modelltransformation als ein Funktionsbausteinnetzwerk (FBN) [KQE11]. Die IEC 61131-3 beschreibt die fünf folgenden Programmiersprachen:

Strukturierter Text (ST) Die textuelle Sprache strukturierter Text (ST) ist von Pascal abgeleitet und ermöglicht die Programmierung der SPS in einer höheren Programmiersprache. Im Jahr 2013 wurde ST um die Paradigmen der Objektorientierung (OO) erweitert. Es ist seitdem möglich Klassen und Interfaces in ST zu definieren. Es werden die Konzepte der OO wie Einfachvererbung, Polymorphismus und unterschiedliche Zugriffsarten der Klassenelemente, wie z. B. Methoden oder Variablen, unterstützt.

Anweisungsliste (AWL) Im Gegensatz zu ST ist die textuelle Programmiersprache AWL eine maschinennahe Sprache und ist mit Assembler vergleichbar. Die Sprache wird häufig als Zwischensprache verwendet, auf die die übrigen Sprachen der Norm abgebildet werden [JT09]. Die dritte Ausgabe der Norm hat AWL als veraltet erklärt. Es bedeutet, dass diese Sprache in einer zukünftigen Ausgabe der Norm nicht mehr enthalten sein wird.

Kontaktplan (KOP) Die graphische Programmiersprache KOP ist in ihrer Darstellung an Stromlaufpläne angelehnt. Die Sprache beinhaltet die sogenannten Kontakte und Spulen, die zwischen den Stromschienen geschaltet und mit booleschen Variablen verknüpft werden. Beide Elemente drücken je nach Art der Verschaltung, z. B. sequentiell oder parallel, boolesche Ausdrücke aus, z. B. eine logische UND- oder eine logische ODER-Verknüpfung.

Ablaufsprache (AS oder engl. SFC) Sequential Function Chart (deutsch Ablaufsprache) (SFC) ist ein Beschreibungsmittel zur Ablaufsteuerung innerhalb einer SPS. Eine detaillierte Beschreibung dieser Sprache ist in Abschnitt 3.4.1 zu finden.

Funktionsbausteinsprache (FBS oder engl. FBD) Der Einfachheit halber werden in dieser Arbeit unter dem Begriff FBD die Funktionsbausteinsprache im engeren Sinne nach der IEC 61131-3 und deren Erweiterung, die als Continuous Function Chart (CFC) bekannt ist, zusammengefasst. Obwohl CFC nicht normiert ist, ist es eine gängige Programmiersprache, die von vielen Programmierumgebungen unterstützt wird [SVH13].

Die Kernelemente von FBD sind Funktionsbausteine, die zur Strukturierung der Programmlogik genutzt werden. Ein Funktionsbaustein kapselt einen ausführbaren Algorithmus und ggf. den Speicherbereich, auf dem der Algorithmus operiert, nach innen und stellt wohldefinierte Ein- bzw. Ausgänge nach außen zur Verfügung. Dieser Ansatz trägt zur komponentenbasierten Entwicklung der leittechnischen Anwendung bei. Man unterscheidet zwischen dem Bausteintyp, einer „Vorlage“ für den Baustein und seiner Logik, und den Bausteininstanzen, die jeweils einen eigenen Speicherbereich besitzen. Die Typisierung der Bausteine ermöglicht eine Wiederverwendung bewährter Algorithmen oder die Nutzung von normierten Bausteintypen.

FBD dient zur signalorientierten Verschaltung von Eingängen bzw. Ausgängen der Funktionsbausteininstanzen und beschreibt so den Datenfluss zwischen diesen bzw. den eingebetteten Algorithmen. Signalorientierung bedeutet in diesem Kontext einen (aus der Sy-

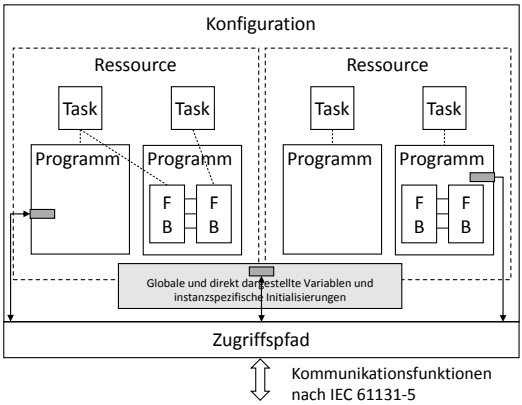


Abbildung 2.7.: Softwaremodell eines SPS-Systems nach IEC 61131-3 [IEC11]. Graue Rechtecke stellen Variablen dar, Pfeile stellen die Pfade des Variablenzugriffs dar.

stemsicht) quasi-kontinuierlichen Informationsaustausch zwischen den Bausteinports. Die graphische Darstellung ist nicht nur für die intuitive Programmierung, sondern auch für die Dokumentation [TM09a] des SPS-Programms von hoher Bedeutung. Diese Struktur kann nicht nur für die Darstellung, sondern auch z. B. für Modelltransformationen genutzt werden [GWE14].

Die Norm sieht Möglichkeiten zur Kombination der Programmiersprachen vor. So werden typischerweise die Algorithmen innerhalb der Funktionsbausteine mittels ST beschrieben und die Bausteininstanzen mittels FBD miteinander verschaltet.

Zusätzlich zu den Programmiersprachen definiert die Norm eine Menge von Funktionsbausteintypen, die von den kompatiblen Systemen implementiert werden. Diese beinhalten z. B. einfache Bausteintypen für arithmetische Operationen oder für die Zwischenspeicherung boolescher Werte (Flipflops).

2.2.5. Softwarearchitektur eines Laufzeitsystems nach IEC 61131-3

Der dritte Teil des Standards IEC 61131 [IEC11] beschreibt neben den Programmiersprachen eine Software-Sicht auf eine SPS. Zusätzlich zu der Norm existiert der technische Bericht IEC 61131-8 [IEC03b], der die Richtlinien für die Implementierung einer IEC 61131-3 kompatiblen Umgebung enthält. Das Softwaremodell ist in Abbildung 2.7 dargestellt. Auf Grundlage dieses Modells basieren die meisten IEC 61131-3 kompatiblen oder an die Norm angelehnten Laufzeitumgebungen. In der Abbildung entspricht die Konfiguration der in Abschnitt 2.2.3 vorgestellten SPS. Die Ressourcen decken die Funktionen der Signalverarbeitung, der Mensch-Maschinen-Schnittstelle und der I/O-Schnittstellen einer SPS ab. Typischerweise bieten die Ressourcen eine Abstraktion für die CPUs einer SPS.

Die anwenderspezifische Anwendung wird laut der Norm in sogenannte POUs unterteilt, die die kleinsten voneinander unabhängigen Einheiten der Software-Anwendung bilden [JT09]. Die POUs werden hierarchisch aufgebaut: Ein Programm kann z. B. aus einem

FBN bestehen, das aus weiteren Funktionsbausteinen oder Funktionen bestehen kann. Der Unterschied zwischen den Funktionsbausteinen und den Funktionen liegt in der Abwesenheit eines internen Speichers bei den letzteren. Jede POU besteht aus einem Deklarationsteil mit der Beschreibung lokaler Variablen und der externen Schnittstellen sowie einem Anweisungsteil mit der Programmlogik.

Die unterschiedlichen Aspekte der Anwendung werden auf verschiedenen Ebenen der POU-Hierarchie abgebildet. So hat z. B. nur das Programm die Möglichkeit des Zugriffs auf die Peripherie einer SPS und stellt diese den enthaltenen POUs zur Verfügung.

Ressourcen enthalten POUs und einen oder mehrere Tasks. Die Tasks sind in der Lage, die Ausführung einer Menge von Programmen oder untergeordneten POUs, z. B. Funktionsbausteinen, anzustoßen. In der Praxis werden die Tasks periodisch (auch zyklisch genannt) ausgeführt. Die Norm sieht jedoch auch eine Möglichkeit der Aktivierung durch Ereignisse vor, z. B. durch Änderung eines I/O-Eingangs. Die Zuordnung der POUs zu Tasks erfolgt in der Definition der Ressource.

Die Kommunikation zwischen den POUs kann auf unterschiedliche Art und Weise erfolgen. Bei POUs innerhalb eines Programms ist der direkte Datenfluss mittels Datenflussverbindungen möglich. Der Datenaustausch zwischen den unterschiedlichen Programmen in einer Konfiguration kann mittels globaler Variablen realisiert werden. Der Datenaustausch zwischen unterschiedlichen Konfigurationen oder einer SPS und einer nicht-SPS ist mittels der Kommunikationsbausteine nach IEC 61131-5 [IEC01] oder Zugriffspfade umsetzbar.

2.2.6. Softwarearchitektur eines Laufzeitsystems nach IEC 61499

Die Norm IEC 61499 [IEC12] für verteilte Automatisierungssysteme erweitert bzw. ersetzt die folgenden Aspekte der IEC 61131-3:

- **Softwarearchitektur des verteilten Automatisierungssystems:** Das Modell der IEC 61131-3 bezieht sich auf eine Konfiguration, d. h. auf ein einzelnes SPS-System. Fragestellungen der Verteilung einer Anwendung auf unterschiedliche Systeme werden nicht explizit angesprochen, sondern dem Nutzer überlassen bzw. über Kommunikationsbausteine der oder Zugriffspfade realisiert. Die IEC 61499 definiert dagegen ein Systemmodell für verteilte Anwendungen, das in Abbildung 2.8 zu finden ist. Die meisten Elemente der Systemarchitektur, wie z. B. die Ressource oder das Gerät, werden von der in dieser Arbeit vorgestellten einheitlichen Laufzeitsystemarchitektur wiederverwendet und daher detailliert in Abschnitt 6.1.1 behandelt.

- **Ereignisgesteuerte Ausführungssemantik der Funktionsbausteine:** Ähnlich wie IEC 61131-3 nutzt die IEC 61499 FBNs als grundsätzliches Mittel zur Anwendungsstrukturierung. Im Gegensatz zu einer Tasklisten-basierten Abarbeitung der Bausteine, führt die Norm jedoch zusätzliche Ports bzw. Datenverbindungen zwischen den Bausteinen ein. Diese ermöglichen den Austausch der Ereignisse zwischen den Bausteinen. Die von den Bausteinen gekapselte Logik wird mittels ST und einem Execution Control Chart (ECC) für die Ablaufsteuerung beschrieben. Die Ablaufsteuerung durch den Ereignisfluss ist eine Voraussetzung für die verteilbare Architektur der Anwendung. Die Ereignisse können genau wie Daten über das Netzwerk übertragen werden und sichern somit eine unveränderte Semantik der Anwendung (abgesehen von der mit der Übertragung verbundenen Latenz).

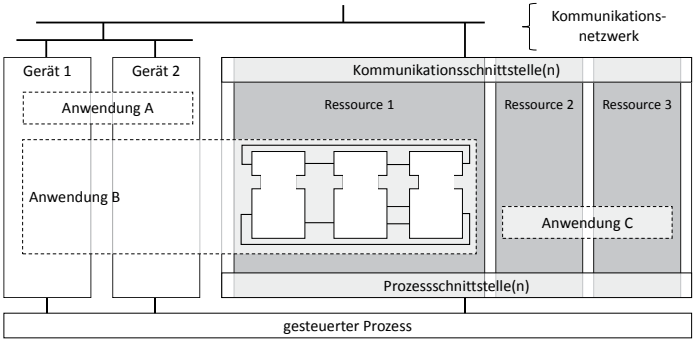


Abbildung 2.8.: Softwaremodell verteilter Automatisierungssysteme nach IEC 61499 [IEC12].

Die Verbreitung und die Akzeptanz der Norm variieren je nach dem Anwendungsgebiet und scheinen in der Fertigung größer als im Bereich der Prozesstechnik zu sein. Der Vergleich zwischen den Konzepten der beiden Normen wird zusätzlich häufig durch Missverständnisse erschwert [Thr13]. In den letzten Jahren wurde an Konzepten zur Koexistenz beider Normen, z. B. in [ZSSB09], und an der automatischen, semantisch-korrekten Übersetzung von IEC 61131-3 Programmen auf das Modell der IEC 61499, z. B. in [DDV14], gearbeitet. Eine Übersicht über die Norm und die ihre Anwendung ist in [Vya11] zu finden.

2.2.7. Entwicklungsphasen leittechnischer Anwendungen

In der Domäne der Prozessleittechnik wird die Gesamtheit der Aktivitäten für die „Projektierung und Konfiguration des Prozessleitsystems“ [TM09a] als das Engineering bezeichnet. Die meisten Anwendungen werden dabei aus vorgefertigten Typen kombiniert und parametriert (vgl. Abbildung 2.4). Auch die Programmierung in den Sprachen der IEC 61131-3 kann, abhängig von der Betrachtungsweise, unter Aktivitäten der Engineering-Phase fallen.

Das Engineering kann abhängig von der „Transparenz“ des konfigurierten Objekts in Black- und White-Box unterteilt werden. Bei einer Black-Box sind nur die äußeren Schnittstellen eines Objekts sichtbar. Bei dem White-Box Ansatz ist hingegen die Betrachtung der Elemente innerhalb der Systemgrenze möglich [Mey02, Alb03]. Der Kompromiss zwischen den beiden Ansätzen, bei dem nur bestimmte Aspekte des Systems sichtbar sind, wird als Gray-Box Engineering bezeichnet.

In [GE13b] wird neben dem Engineering zusätzlich die Aktivität der Typenentwicklung der anwendungsspezifischen Bausteintypen aus Abbildung 2.4 hervorgehoben. Am anderen Ende des Spektrums befindet sich die Tätigkeit der Orchestrierung komplexer Subsysteme bzw. Dienste, die als Systems Engineering bezeichnet wird.

2.2.8. Akteure im Entwicklungsprozess leittechnischer Anwendungen

Das PLS unterliegen dem generischen Lebenszyklusmodell für technische Assets [VDI15b] und hat somit unter anderem eine Errichtungs-, eine Inbetriebsetzungs- und eine Betriebsphase [NAM03]. Die während des Lebenszyklus anfallenden Aufgaben sind im Entwick-

lungsprozess der leittechnischen Anwendungen auf viele Akteure verteilt. Diese haben oft nicht nur unterschiedliche fachliche Hintergründe, sondern sind auf mehre Unternehmen verteilt. Typischerweise lassen sich die beteiligten Akteure in drei Gruppen unterteilen.

Hersteller Die Hersteller leittechnischer Komponenten können nach [VhKW09, NAM02] in Leitsystem- und Feldgeräte-Hersteller sowie Tool-Lieferanten unterteilt werden. Die Tool-Lieferanten entwickeln Software-Tools, die für das Engineering oder den Betrieb des Leitsystems notwendig sind, z. B. eine Entwicklungsumgebung. Große Technologiehersteller können die Hardware- und die Softwareplattform aus „einer Hand“ anbieten und somit einen hohen Grad der Integration erreichen. Hersteller bieten unterschiedliche Produktlinien für die unterschiedlichen Anwenderbranchen an, die oft auf einem Kernprodukt basieren, jedoch im Funktionsumfang erheblich variieren können.

Engineering-Dienstleister Die Dienstleister, auch Automation Contractor genannt, suchen die für die Anlagenautomatisierung benötigten Komponenten gemäß der vorgegebenen Spezifikationen aus und begleiten die Anlage angefangen mit der Planung bis zu ihrer Inbetriebnahme. In den Begriffen der NE 58 [NAM02] werden die Dienstleister als „Errichter“ bezeichnet. Sowohl die Hersteller als auch die Anwender bzw. deren speziellen Abteilungen können diese Rolle übernehmen. Die Erstellung der Software-Anwendung gehört auch zu den Pflichten des Engineering-Dienstleisters.

Betreiber Die Anwender des PLS Systems betreiben die Anwendung und die Anlage ohne Änderungen in die Hardware- oder Softwarestruktur des PLS-Systems vorzunehmen. Da aber typischerweise Änderungen während des gesamten Anlagenlebenszyklus notwendig sind, übernehmen Betreiber bzw. spezielle Abteilungen oder beauftragte Firmen die Aufgaben des Engineerings. Die Anpassungen des Systems während der Betriebsphase wird auch als Reengineering [LHFL14a, LHFL14b] bezeichnet. Insbesondere im Bereich der Anlagenautomatisierung sind die vom Betreiber bzw. während des Betriebs eingebrachtes Know-How äußerst anlagenspezifisch. Aus diesem Grund werden viele Engineering-Aufgaben vom Betreiber der Anlage durchgeführt.

Eine grobe Übersicht über den mehrschichtigen Aufbau des Laufzeitsystems aus Sicht der Anwender ist in Abbildung 2.4 zu finden. Der Aufbau des Laufzeitsystems ist demnach in der Verantwortung des Herstellers eines solchen Systems. Der Errichter bzw. der Benutzer setzen in den meisten Fällen die vorkonfigurierten „Bausteine“ des Laufzeitsystems ein. Dazu zählen vordefinierte Funktionsbausteintypen für Funktionsbeschreibung oder Faceplates für das HMI. Die Entwicklung eigener Bausteintypen passiert nur in Spezialfällen bzw. für die vom Hersteller nicht abgedeckte Funktionalität.

3. Stand der Wissenschaft und Technik

Der Fokus dieses Kapitels liegt auf den gegenwärtigen Ansätzen der Flexibilisierung leittechnischer Anwendungen und den verfügbaren Implementierungen industrieller Laufzeitsysteme. Dem Leser wird ein Überblick über die Forschungsaktivitäten der letzten Jahre und die Vorarbeiten des Autors geboten.

Die Forschungsaktivitäten werden in Kategorien der Beiträge zur Flexibilisierung leittechnischer Anwendungen, der Arbeiten im Bereich der Laufzeitumgebungen in der Automatisierungstechnik und der Prozedurbeschreibungssprachen für leittechnische Anwendungen unterteilt.

3.1. Eigene Vorarbeiten

Die für diese Arbeit relevanten Beiträge wurden im Rahmen der Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Prozessleittechnik der RWTH Aachen University und der Mitgliedschaft im DFG Graduiertenkolleg „AlgoSyn“ (Algorithmische Synthese reaktiver und diskret-kontinuierlicher Systeme) angefertigt. Diese Beiträge lassen sich in drei Themengebiete unterteilen, die für die Ausrichtung dieser Arbeit von Bedeutung sind: die Architektur der Laufzeitumgebung, die Beschreibungsmittel der von der Laufzeitumgebung ausgeführten Anwendungen und die leittechnischen Use-Cases, die nur mit rekonfigurierbaren Laufzeitsystemen bzw. mit ressourcenadaptiven Algorithmen realisierbar sind.

Folgende relevante Ergebnisse wurden auf nationalen und internationalen Konferenzen und Workshops veröffentlicht (nach Themengebieten und in chronologischer Reihenfolge sortiert).

Softwarearchitektur der Laufzeitumgebung In [GKE12] wurde der Fortschritt des Lehrstuhls auf dem Gebiet modellbasierter Implementierung der auf der IEC 61131-3 basierender graphischen Funktionsbausteinsprache (FBD) vorgestellt. Die Bausteinsprache ist eine der sechs grundlegenden Beschreibungssprachen für Anwendungen in der Prozessleittechnik. Der Fokus der Arbeit liegt auf der Beschreibung des zugrunde liegenden objektorientierten Meta-Modells für die Funktionsbausteine sowie der groben Strukturierung des Entwicklungsprozesses in eine Entwicklungs- und eine Engineeringphase (vgl. Abschnitt 2.2.7). Darüber hinaus wurden Gesichtspunkte des Baustein- und Bibliothekslebenszyklus diskutiert.

Der als [GE13b] veröffentlichte Beitrag listet Anforderungen und Lösungsparadigmen für ebenenübergreifende Laufzeitsysteme auf. Ebenso legt er einen Grundstein für die Strukturierung der Anwendungen durch die Einführung des Begriffs der „selbstständigen Komponente“. Diese nehmen eine Stellung zwischen den vorhandenen Organisationseinheiten der Software aus der IEC 61131-3 und der IEC 61499 ein und stellen deswegen eine Möglichkeit zur Homogenisierung der unterschiedlichen Ausführungssemantik beider Standards

dar. Die Komponenten schließen die Lücke zwischen dem verteilten Automatisierungssystem und den (in den meisten Fällen) auf eine CPU bzw. Ressource zugeschnittenen Anwendungen der IEC 61131-3. Diese Vorgehensweise erlaubt eine schrittweise Migration der bestehenden Anwendungen auf die neu vorgeschlagene Architektur des Laufzeitsystems. Eine detaillierte Beschreibung der selbständigen Komponenten und deren Einsatz im Kontext des Schedulings ist in Abschnitt 6.1.2 zu finden. Darüber hinaus wird das aus [GKE12] bekannte Entwicklungsmodell unter den Aspekten der Black- und White-Box Modellierung und Verifikation erläutert. Die überarbeitete Fassung des Beitrags erschien in deutscher Sprache als [GE13a].

In [GE15] wurden die Möglichkeiten für den Einsatz von Echtzeiteigenschaften für den Entwurf heterogener Systeme in der industriellen Automation aufgezeigt. Das Echtzeitsystem wird dabei in eine Menge von End-to-End Flüssen unterteilt, die die gewünschten End-to-End Systemverhalten beschreiben und implementierungsunabhängig sind. Unter dem End-to-End Verhalten werden die erwünschten physischen Konsequenzen einer physischen Systemanregung verstanden. Dieses Verhalten aggregiert somit alle Aspekte der Sensorik (z. B. die Signalverarbeitung), der Aktorik (z. B. das Losreißmoment) und der tatsächlich ausgeführten Kontrolllogik (z. B. die Verzögerungen aufgrund der zyklischen Abtastung). Die temporalen Eigenschaften der Flüsse werden mithilfe der angenommenen und der worst-case Ausführungszeit sowie der Deadline-Charakteristik definiert. Der dimensionslose Quotient beider Ausführungszeiten wird als der Kritikalitätsfaktor bezeichnet und kann für die Auslegung des Systems verwendet werden. Diese Ansätze finden für das Scheduling der selbstständigen Komponenten in Kapitel 6 Verwendung.

Prozedurbeschreibungsmittel für leittechnische Anwendungen und deren Semantik
In [YQGE12] wurde die Rolle der SFCs, einer zustandsorientierten graphischen Programmiersprache aus der IEC 61131-3, als universelles Beschreibungsmittel für Anwendungen in der Leittechnik diskutiert. Es wurde eine Reihe von Erweiterungen und Veränderungen identifiziert, die benötigt werden, um diese Sprache für einen implementierungs- und domänenübergreifenden Einsatz verallgemeinern zu können. Diese Erweiterungen umfassen eine serviceorientierte Schnittstelle für SFC, die die Voraussetzungen für lose gekoppelte Anwendungen schafft sowie eine auf den Funktionsaufrufen basierende Variation der Ausführungssemantik der Charts, welche eine effiziente und intuitive Ausführung der SFC-Schritte und der enthaltenen Aktionen ermöglicht.

Die nachfolgende Publikation [YGE13] auf diesem Themengebiet führt eine Modifikation der Syntax von SFC ein, die an UML Statecharts angelehnt ist. Die Sprache wird als Sequential State Charts (SSC) bezeichnet und ist durch eine Kombination aus einem universellen Beschreibungsmittel für Prozeduren in unterschiedlichen Bereichen der Leittechnik mit einer eindeutigen Ausführungssemantik gekennzeichnet. Der Fokus der Arbeit liegt auf der Einbettung der mithilfe von SSCs beschriebenen Prozeduren in ein auf IEC 61131-3 basiertes zyklisches Laufzeitsystem. Diese Aufgabe wird durch einen definierten Ausführungsrahmen gelöst, der die Interaktion zwischen der Prozedur und den signal- sowie nachrichtenbasierten Kommunikationspartnern reglementiert. Die Semantik von SSCs wurde mithilfe der UPPAAL Modellierungsumgebung für Timed Automata beschrieben. Dieser pragmatische Ansatz ermöglicht eine knappe Beschreibung des Systemverhaltens, die bei Bedarf in eine formale Form überführt werden kann. Darüber hinaus bietet das Tool die Möglichkeiten der Simulation und der formalen Verifikation von Prozedureigenschaften.

Die Sprache SSC wird im Rahmen dieser Arbeit mit zeitbehafteten Übergangstransitionen und durch eine in-cycle Semantik erweitert. Die Formalisierung der Semantik erfolgt ebenfalls unter Verwendung von UPPAAL (vgl. Abschnitt 6.2.4).

Leittechnische Anwendungen für rekonfigurierbare Laufzeitumgebungen In [GWE14] wurde ein neuartiges regelbasiertes System vorgestellt, das für zahlreiche Aufgaben im Kontext der Automatisierung der Automatisierung (AdA) eingesetzt werden kann. Die Regeln für das System werden mithilfe von Graphabfragen formuliert, die auf einer graphischen NoSQL Datenbank ausgeführt werden. Die Datenbank kann verschiedenste Daten beinhalten, die als attributierte Multigraphen dargestellt werden könnten. Eine solche Darstellung ist für die meisten relevanten Modelle der Leittechnik in der Regel bereits vorhanden (vgl. Sprachen der IEC 61131-3 in Abschnitt 2.2.4). Diese generische Vorgehensweise ermöglicht das Formulieren von Regeln für ein breites Spektrum der Anwendungen. Da die Abfragen Aussagen über partielle Graphen treffen, können diese jedenfalls graphisch formuliert und dargestellt werden, was maßgeblich zur Akzeptanz der Lösung unter den Anwendern beiträgt. In der Publikation wurde ein R&I Fließschema als Datenquelle für die Regelanwendung verwendet, das regelbasiert in eine funktionsbausteinbasierte Implementierung der Basisautomatisierungsfunktionen, wie z. B. Einzelkontrollbausteinen und einfachen Regelkreisen, überführt wurde. Das Erzeugen dieser Kontrolllogik im laufenden Betrieb birgt allerdings auch Gefahren, wie jeder Engineering-Eingriff. Aus diesem Grund wird das vorgestellte AdA-System als ein Anwendungsfall für die Transaktionskontrolle in Abschnitt 7.5 verwendet. Das regelbasierte System wurde zusätzlich im Rahmen eines Workshops vorgestellt [GE14].

Ein weiterer Anwendungsfall für das adaptive Laufzeitsystem ist die nicht-echtzeitfähige Kommunikation am Beispiel des OPC UA Kommunikationsprotokolls. Im Rahmen des Projekts „open62541“ [PGP+15] wird am Lehrstuhl für Prozessleittechnik eine Open-Source Implementierung eines OPC UA Kommunikationsstacks entwickelt, welche ausschließlich der standardisierten Spezifikation (IEC 62541 [IEC10]) des Protokolls folgt. Ein wichtiger Aspekt des Projekts sind die Möglichkeiten der Einbettung des neuen Kommunikationsstacks in vorhandene Anwendungen bzw. das Koppeln des OPC UA Meta-Modells mit den vorhandenen Modellen. In dieser Arbeit werden die Möglichkeiten der Einbettung eines open62541-Servers in die Laufzeitumgebung (vgl. Abschnitt 7.2) untersucht. Um den Durchsatz des Servers zu erhöhen, wurde zusätzlich eine Reihe von Erweiterungen des Standards für zustandslose OPC UA Kommunikation vorgestellt und deren Auswirkungen untersucht [GPP15, GPP16].

Ein Ausschnitt der Anforderungsanalyse und der Use-Cases dieser Arbeit wurde vorab auf einer Konferenz vorgestellt [GE16].

3.2. Ansätze zur Flexibilisierung leittechnischer Anwendungen

3.2.1. Lose Kopplung der Komponenten durch Serviceorientierung

Die Anwendung einer serviceorientierten Architektur (SOA) im Umfeld der industriellen Produktion wurde bereits 2005 in [JS05] vorgeschlagen. Das Ziel dieses Ansatzes ist es

verteilte Systeme aufzubauen, die aus autonomen und gleichzeitig interoperablen Komponenten bestehen. Ein Augenmerk liegt dabei insbesondere auf Komponenten, die in unterschiedlichen Verantwortungsbereichen z. B. Organisationen, Einheiten oder Anwendungsdomänen liegen [OAS06].

Eine abstrakte Funktionalität wird als Dienst (engl. service) bezeichnet, der von einem Dienstanbieter angeboten wird und von einem Dienstnutzer in Anspruch genommen werden kann. Dazu muss jede Komponente eine wohldefinierte Schnittstelle besitzen, über die der Dienst angeboten bzw. genutzt werden kann. Die Beschreibung dieser Schnittstelle wird als Dienstbeschreibung bezeichnet und erfolgt unabhängig von der Implementierung des Dienstes. Diese Abstraktion ermöglicht es die Schnittstellen eines Dienstes von dessen interner Logik zu trennen. Das Kernmodell der Dienste [DIN14] sieht eine Möglichkeit vor, Dienste zu typisieren sowie Anforderungen bzw. Zusicherung bestimmter Qualitätsmerkmale an Dienste anzuknüpfen.

Während die Ideen eines standardisierten Interfaces der Software-Komponenten [TVK01] bereits durch die Anwendung der Bausteinmuster aus der IEC 61131-3 eine weite Präsenz in der Domäne der Automatisierungstechnik findet, bleiben die einzelnen Bausteine durch die Signalverbindungen eng aneinander gekoppelt, was die Portabilität und die Flexibilität der Anwendungen beeinträchtigt. Enge Kopplung bedeutet in diesem Kontext eine statische Verbindung zwischen zwei Komponenten, die in der Engineering-Phase definiert wurde und zur Laufzeit des Systems unveränderbar ist [EE13]. Aus dieser Perspektive liegt der Vorteil von SOA in einer losen Kopplung der Komponenten, d. h. die einzelnen Komponenten der Anwendung können flexibel und meist zur Laufzeit ihren Kommunikationspartner aussuchen bzw. wechseln. Die Kommunikation erfolgt dabei üblicherweise über einen diskreten Nachrichtenaustausch zwischen den Kommunikationspartnern. Die Erkundung der verfügbaren Diensteanbieter und deren angebotene Dienste geschieht in der Regel durch ein zentrales Dienstverzeichnis, bei dem sich die Anbieter anmelden [DMES14]. Die Existenz eines Solchen ist aber nicht zwingend erforderlich [OAS06] z. B. falls alle Partner bereits im Voraus bekannt sind.

Die Auswahl und Kopplung unterschiedlicher Diensteanbieter und Dienstnutzer durch einen zentralen Manager wird als „Orchestrierung“ bezeichnet. Die passenden Kommunikationspartner können automatisch mittels Ontologien [FLR+13, DMES14] oder durch Modellierung von Fähigkeiten einzelner Diensteanbieter und deren Abgleich mittels Computer-aided Process Planning [PSA+15] identifiziert werden.

Die Anwendbarkeit des SOA-Ansatzes in der Fertigungstechnik wurde im Rahmen des EU-Projekts SOCRADES untersucht und erfolgreich bewiesen [DSSG+08]. Dabei wurde eine Verbindung zwischen der Enterprise Resource Planning (ERP) Ebene und der Feldebene mittels Web-Service-basierter Middleware hergestellt. Weitere Beispiele der SOA Anwendung kommen aus den Domänen der Logistik und der Halbleiterherstellung [Kom06]. Eine Studie bezüglich der Eignung von SOA für Programmierung industrieller Roboter-Zellen [VPN09] stellte fest, dass die SOA-Ansätze den Ingenieuren helfen sich auf dem Gebiet der eigenen Expertise zu fokussieren und den Schnittstellenaufwand zwischen unterschiedlichen Komponenten der Zelle reduzieren. Zusätzlich war SOA weniger kostenaufwendig als vergleichbare Techniken der OO. Darüber hinaus wird SOA als Architekturparadigma für Industrie 4.0 postuliert [KWH13].

Auf dem Gebiet der Prozessautomatisierung und der Prozessleittechnik wurde ein universeller serviceorientierter Zugriff als Brücke zwischen der Prozessleitebene und der Funktionalität auf höheren Ebenen der Automatisierungspyramide vorgeschlagen [SEE09] und

ein Format für einen Nachrichtenaustausch spezifiziert [EE13]. Darüber hinaus wurde die Anwendbarkeit einer SOA-basierten Middleware mit Fokus auf einer transparenten Verteilung der Dienste [ME12] sowie die Nutzung der Dienste als Schnittstelle zwischen den Batch- und MES-Systemen diskutiert [NAM12].

3.2.2. Agentensysteme

Agentensysteme folgen dem Grundgedanken komplexe Systeme als ein Verbund aus autonomen, intelligenten Akteuren zu modellieren [Epp13]. Der Begriff kommt ursprünglich aus dem Gebiet der künstlichen Intelligenz [WJ95, Woo09] und wird seit mehr als einer Dekade im Kontext der Automatisierung verwendet [Epp00].

Für die Zwecke dieser Arbeit dient das Begriffssystem der [VDI10] als Grundlage. Eine Diskussion alternativer Definitionen des Begriffes „Agent“ ist in [Yu16] zu finden. Laut [VDI10] sind Agenten wie folgt definiert: „Ein [technischer] Agent ist eine abgrenzbare (Hardware- oder/und Software-) Einheit mit definierten Zielen. Ein Agent ist bestrebt, diese Ziele durch selbstständiges Verhalten zu erreichen und interagiert dabei mit seiner Umgebung und anderen Agenten.“ Technische Agenten zeichnen sich durch folgende Eigenschaften aus, die in unterschiedlichem Maße ausgeprägt sein können [VDI10, Yu16]:

- Ein Agent arbeitet zielorientiert, d. h. er ändert sein Verhalten um das Ziel oder Ziele zu erreichen.
- Reaktivität und Interaktion erlauben es dem Agenten sich an die Umgebung anzupassen bzw. mit anderen Agenten zu interagieren, um eigene Ziele zu erreichen.
- Ein Agent agiert innerhalb eines festgelegten Handlungsspielraums.
- Ein Agent kapselt seinen Zustand, sein Verhalten, die eigene Strategie und das Ziel für den Zugriff von außen.
- Die Mobilität eines Agenten beschreibt seine Fähigkeit zwischen unterschiedlichen physischen und logischen Ausführungskontexten zu wechseln.
- Autonomie und Persistenz erlauben dem Agenten selbständig zu handeln und den inneren Zustand über seinen gesamten Laufzyklus zu konservieren.

Einige dieser Eigenschaften, wie z. B. die Kapselung, werden durch bereits vorgestellte Paradigmen der komponentenbasierten Entwicklung, z. B. mittels der Funktionsbausteine, abgedeckt. Aus diesem Grund existiert keine scharfe Trennlinie zwischen einem Programm oder einer Anwendungskomponente und einem technischen Agenten. Die Eigenschaften der Autonomie und der Interaktion sind allerdings aufgrund der verwendeten engen Kopplung durch Signalverbindungen weniger ausgeprägt [Epp13]. Eine mögliche Lösung für diese Problematik bietet die lose Kopplung der Agenten mittels SOA [Yu16].

Die Ausführung paralleler Aktivitäten auf Agentenbasis formt ein sogenanntes Multi-Agenten System, das eine Alternative zu der traditionellen hierarchischen Anlagensteuerung darstellt. Diese Systeme können allerdings sowohl in hierarchischen, als auch in nicht hierarchischen Systemen eingesetzt werden, z. B. im Kontext von CPPS [VHLL15].

Die Anwendbarkeit der Agentensysteme im Bereich der Leittechnik wurden im Laufe der letzten Jahre untersucht [Epp00, Epp13, YSE14]. Ähnlich wie bei der Nutzung anderer Technologien, spielen dabei insbesondere die domänenspezifischen Anforderungen eine

herausragende Rolle. So sind beispielsweise die Handlungsspielräume der Agenten in der industriellen Automatisierung bewusst sehr eng ausgelegt [Epp13]. Demnach sind Agenten, die sich eigene Aufgaben selbst suchen, unerwünscht. Trotz dieser Einschränkungen wurde eine Reihe von Aufgaben für Agentensysteme identifiziert und erfolgreich evaluiert [Yu16], z. B. in den Aufgaben der Anlagensteuerung oder der Verwaltung von Modellen.

3.2.3. Modellgetriebene Ansätze

Ein weiterer Ansatz für die Flexibilisierung der leittechnischen Anwendungen ist deren automatische Generierung aus höherwertigen Modellen. Solche höherwertige Modelle, z. B. R&I Fließschemata, sind in der Domäne der Leittechnik nicht nur bereits vorhanden, sondern werden von den Anwendern zu unterschiedlichsten Zwecken, z. B. zur Dokumentation, genutzt und gepflegt. Die für die Beschreibung solcher domänenspezifischen Modelle verwendeten Sprachen werden als Domain-Specific Languages oder domänenspezifische Sprachen (DSL) bezeichnet und sind oft speziell auf einen bestimmten Aspekt der Problemstellung zugeschnitten.

Meistens können die typischen Änderung der Umgebung einer Anwendung durch einfache Änderungen des abstrakten Modells beschrieben werden. Als Beispiele dafür dienten eine Änderung des kontrollierten physischen Systems oder eine Änderung des Rezepts eines Produkts. Kleine Änderungen auf der hohen Abstraktionsebene eines Modells können allerdings drastische Änderungen in der Struktur bzw. dem Quellcode der leittechnischen Anwendung auslösen.

Die Erzeugung dieser Änderungen bzw. der kompletten Anwendung soll durch modellgetriebene Softwareentwicklung automatisiert werden. Diese ist wie folgt definiert: „Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“ [SVE07]. „Formal“ bedeutet in diesem Kontext ein Modell, welches die komplette Beschreibung eines Aspektes der Anwendung beinhaltet. Erzeugen von „lauffähiger Software“ kann entweder durch die Generierung von Quelltext in einer anderen Sprache, z. B. ST, oder durch Interpretation des Modells zur Laufzeit erfolgen. Die automatische Erzeugung der Anwendung heißt nicht zwangsläufig, dass die komplette Anwendung erzeugt werden muss. Vielmehr können deren Strukturen nur bis zu einem bestimmten Grad der Abstraktion erzeugt werden, z. B. bis zu der Abstraktion der Funktionsbausteininstanzen, deren Typen manuell implementiert wurden.

Im Bereich der industriellen Automatisierung spielen die Sprachen der IEC 61131-3 oft die Rolle der Zielplattform für modellgetriebene Ansätze bei der Verwendung eines Codegenerators. Es existieren Ansätze, die den kompletten SPS-Code ausgehend von Unified Modeling Language (UML) Modellen [WVH11, FLR+13] bzw. Systems Modeling Language (SysML) Modellen [TF11] generieren. Weitere Systeme erzeugen FB-basierte Logik für Basisautomatisierung, z. B. Interlocks auf Basis von R&I Fließschemata [GWE14]. Die Methoden der AdA zum automatischen Engineering, die auf Modellen basieren, können auch als Teildisziplin der modellgetriebenen Softwareentwicklung behandelt werden, da das Engineering als Programmierung im Sinne des Software-Engineerings gesehen werden kann. Eine Übersicht über den Einsatz modellgetriebener Softwareentwicklung in der Automatisierungstechnik ist in [Vya13] zu finden.

3.2.4. Laufzeit-Rekonfiguration verteilter Automatisierungssysteme

Der effiziente Austausch bzw. ein Update der generierten Anwendung auf dem Laufzeitsystem stellt eine domänenspezifische Herausforderung dar. Im Gegensatz zu vielen generischen Softwaresystemen, kann die Beschaffenheit des kontrollierten physischen Systems (z. B. hohe Kosten des Stillstandes) ein Update der Software zur Laufzeit bzw. in bestimmten Zeitspannen erfordern.

Da die verteilten Automatisierungssysteme Gegenstand der IEC 61499 sind, kommen erwartungsgemäß die meisten Beiträge aus der IEC 61499 Community, wie z. B. [BZXN02] oder [LZVM11]. Die Anforderungen an den Vorgang der Rekonfiguration sind in [SMS+06] anschaulich zusammengefasst. In [SZ11] wird ein Ansatz vorgestellt, der eine dedizierte Rekonfigurationsanwendung vorsieht, die den Vorgang der Rekonfiguration überwacht. Da die Rekonfigurationsanwendung auf der gleichen Ressource wie die zu konfigurierende Logik ausgeführt wird, unterliegen beide den gleichen Echtzeitanforderungen. In [YV13] wurde die Anwendbarkeit der Konfigurationsbefehle der Norm für die Zwecke der Rekonfiguration untersucht und die vorhandenen Lücken auf der Anwendungsebene geschlossen.

Eine unterbrechungsfreie Rekonfiguration im Rahmen des FASA (Factory Automation System Architecture) Rahmenwerks wurde in [WO14] vorgestellt. In einem Experiment wurde der Algorithmus eines Funktionsbausteins bei einer Ausführungsfrequenz von 1 kHz ausgetauscht. Dabei ging die Kontrolle über ein instabiles physisches System nicht verloren. Zur Synchronisation des Zustands des Funktionsbausteins wurde ein Algorithmus aus [WRKO11] verwendet. Ein Überblick über das FASA Rahmenwerk ist in Abschnitt 3.3.3 zu finden.

3.2.5. Laufzeit-Rekonfiguration der IEC 61131-3-basierten Laufzeitsysteme

Die Rekonfiguration zentralisierter leittechnischer Anwendungen auf Basis von standardisierten IEC 61131-3 Umgebungen beinhaltet in der Regel das Anhalten des Laufzeitsystems für die Änderung der Software. Darüber hinaus lassen manche kommerzielle Leitsysteme den Austausch der Anwendung im beschränktem Maß zu (runtime-update oder delta-update genannt).

Als methodische Erweiterung dazu kann ein übergeordnetes Supervision System die Laufzeitumgebung in bestimmten „sicheren“ Zeitpunkten für ein Update anhalten [PSVHM15]. Dieser Vorgang kann modellgestützt durchgeführt werden. Dabei wird anhand eines Modells die zu ersetzende Anwendung in einer IEC 61131-3 Sprache erstellt. Anschließend wird eine Kommunikation mit der Laufzeitumgebung hergestellt und das sichere Anhalten initiiert. Das Laufzeitsystem hält nur in solchen Prozedur-Zuständen an, die vom Nutzer explizit als sicher markiert wurden. Nach dem Anhalten kann die ausgeführte Logik ausgetauscht werden und die zuvor gesicherten Zustände einzelner Bausteine wiederhergestellt werden.

Weitere, an IEC 61131-3 angelehnte Laufzeitsysteme, wie z. B. ACPLT/RTE (vgl. Abschnitt 3.3.4), lassen die Laufzeitänderung der gesamten Struktur der Anwendung mit dem Models@run.time Ansatz zu. Diese Möglichkeiten wurden in [KQE11, GWE14] am Beispiel des regelbasierten Engineerings erfolgreich demonstriert.

3.3. Laufzeitsysteme der Automatisierungstechnik

Im folgenden Abschnitt werden exemplarisch vier Laufzeitsysteme unter den Gesichtspunkten der Architektur und des Taskings kurz vorgestellt. Erwartungsgemäß sind die Aspekte der Architektur der Systeme aus Bereichen der akademischen bzw. der industrienahen Forschung durch Veröffentlichungen besser abgedeckt, als die der industriellen Laufzeitsysteme.

3.3.1. IEC 61131-3: CODESYS Runtime

Als Vertreter der IEC 61131-3 Softwarearchitektur wird die Laufzeitumgebung CODESYS Runtime der Firma 3S-Smart Software Solutions GmbH vorgestellt. Es handelt sich dabei um eine Soft-SPS. Das bedeutet, dass das eigentliche Laufzeitsystem auf unterschiedlichster Hardware (sowohl aus dem industriellen als auch aus Konsumenten-Segment) lauffähig ist. Die Anpassung der Software an die spezifische Hardware-Plattform erfolgt mithilfe von einem Software Development Kit (SDK) [3S-15b]. So werden z. B. die I/O-spezifischen Treiberbausteine angepasst. CODESYS Runtime dient als SPS-Kern für Plattformen vieler Firmen, wie z. B. ABB, WAGO oder Festo.

Die Laufzeitumgebung läuft unter unterschiedlichen (Echtzeit-)Betriebssystemen (Windows, Windows CE, Linux, VxWorks, QNX) [3S-15b]. Die Programmierung der Anwenderlogik erfolgt in einem dedizierten Entwicklungssystem (CODESYS Development System), das den kompilierten Binärcode zur Laufzeitumgebung überträgt. Die Entwicklungsumgebung erlaubt die Einbettung vom C Programmen in IEC 61131-3 Projekte.

In Bezug auf Scheduling wird die Echtzeitfähigkeit des Systems in einer Windows-Umgebung durch einen Systemtreiber, der auf die Hardware-Timer des Systems zugreift, sichergestellt [3S-15a]. Die Architektur der Laufzeitumgebung sieht einen IEC-Taskmanager vor [3S-15b], der das zyklische Scheduling übernimmt. Es werden sowohl das Preemptive als auch das Cooperative Scheduling-Modell des Betriebssystems seitens der Laufzeitumgebung unterstützt. Zwecks der erhöhten Zuverlässigkeit des gesamten Systems ist auch ein redundanter Betrieb von zwei gekoppelten Laufzeitsystemen möglich.

Die CODESYS Umgebung dient als Testumgebung für diverse Forschungs- und Standardisierungsaktivitäten der Domäne der Automatisierungstechnik, z. B. für PLC-Statecharts (vgl. Abschnitt 3.4.2) oder die objektorientierten Erweiterungen der Sprache ST [Wer09].

3.3.2. IEC 61499: 4DIAC FORTE

Das Eclipse-Projekt 4DIAC (The Framework for Distributed Industrial Automation and Control) zielt auf die Entwicklung einer quelloffenen IEC 61499-kompatiblen Entwicklungs- und Laufzeitumgebung [SZE10]. Die Laufzeitumgebung des Projekts heißt FORTE und folgt dem Soft-SPS Prinzip bezüglich der Hardwareabstraktion. Das Laufzeitsystem wird in C++ entwickelt und ist modular aufgebaut. Die meisten Erweiterungen sind somit als FB-Bibliotheken zuladbar. Das Laufzeitsystem unterstützt diverse Betriebssysteme und Hardwareplattformen, wie z. B. Lego Mindstorms nxt controller, Bachmann electronic M1 PLC und WAGO PFC200.

Die Umsetzung der IEC 61499 erfordert ein System der Ereignissteuerung. Diese wird in FORTE über die sogenannten Event Chains [ZGSS07] implementiert. Unter gewissen Einschränkungen können auch die Echtzeiteigenschaften der Chains zugesichert werden.

Die grundlegende Architektur des Laufzeitsystems besteht aus einer Abstraktionsschicht, einigen Systemdiensten wie Logging und einer Reihe von Bausteininstanzen, die aus den mitgelieferten Bibliotheken stammen. Das Engineering und die Typenentwicklung erfolgt über ein Eclipse-basiertes Integrated Development Environment (IDE) in Sprachen der IEC 61131-3. Die Umgebung erlaubt die Kompilierung einzelner Bausteintypen für das Zielsystem und stößt die Rekonfiguration der Laufzeitumgebung über die Dienste der IEC 61499 an.

Die quelloffene Laufzeitumgebung FORTE ist eine beliebte Plattform für die akademische und die industrienahe Forschung in der IEC 61499 Community.

3.3.3. FASA

Unter dem Namen FASA wird eine Softwarearchitektur für „die neue Generation der Automatisierungssysteme“ [WRKO11, WO14, WGKO15] entwickelt. Diese wird von mehreren Forschungszentren der Firma ABB entwickelt. FASA beinhaltet eine Referenzimplementierung eines Laufzeitsystems, die im Fokus dieser Zusammenfassung steht.

Die Architektur beschreibt ein verteiltes Automatisierungssystem, das zyklische Anwendungen ausführt und somit einen Kompromiss zwischen den Ansätzen der beiden großen Normen bildet. Das Ziel der Architektur ist eine flexible Zuordnung (eine n-zu-m Zuordnung) und Ausführung von mehreren Anwendungen auf mehrere Hardwareressourcen zu erreichen. Die eigentliche Zuordnung der Anwendungen an einzelne Ressourcen und die Berechnung der statischen Tabellen für das zyklische Scheduling einzelner CPUs involviert die Lösung eines NP-harten Problems. Dieses wird mittels eines Constraint Programms oder einer Heuristik gelöst [WGKO15].

Die C++ Referenzimplementierung der FASA-Laufzeitumgebung basiert auf einem Mikrokernel-Prinzip, weshalb die meisten Features der Umgebung als Module zuladbar sind. Der Kernel der Laufzeitumgebung enthält einen zyklischen Scheduler, der die zugeordneten Blöcke aktiviert und bis zum Zeitpunkt der erneuten Ausführung schläft. Das Scheduling unter Einsatz einer statischen Tabelle erlaubt die Nutzung der Slackzeit für Verwaltungsaufgaben innerhalb des Systems, wie z. B. das Umschalten des Schedules zwischen den Zyklen. Als IDE für die Systemkonfiguration wird die IDE des 4DIAC Projekts verwendet [FDGV13].

3.3.4. ACPLT/RTE

Die quelloffene Laufzeitumgebung ACPLT/RTE wird am Lehrstuhl für Prozessleittechnik entwickelt und findet in unterschiedlichen Forschungs- und Industrieprojekten Einsatz. Der Name besteht aus Abkürzungen: Aachener Prozessleittechnik (ACPLT) und Runtime Environment (RTE). Die wichtigsten Differenzierungsmerkmale gegenüber den anderen Laufzeitsystemen sind die ausgeprägten Möglichkeiten der Introspektion und der Reflexion des Systems und der nutzerspezifischen Anwendung. Dieses ist durch den konsequenten Einsatz des ACPLT/OV (Objektverwaltung) Meta-Modells [Mey02] möglich, das für die Abbildung der Großzahl von Objekten innerhalb des Systems verwendet wird. Die Abfrage der Meta-Informationen über die Objekte ist entweder durch ein Application Programming Interface (API) durch die Anwendung selbst oder durch ein geeignetes Kommunikationsprotokoll mit Meta-Modell Unterstützung, wie z. B. ACPLT/KS (Kommunikationssystem) [Alb03] oder OPC UA [IEC10], möglich.

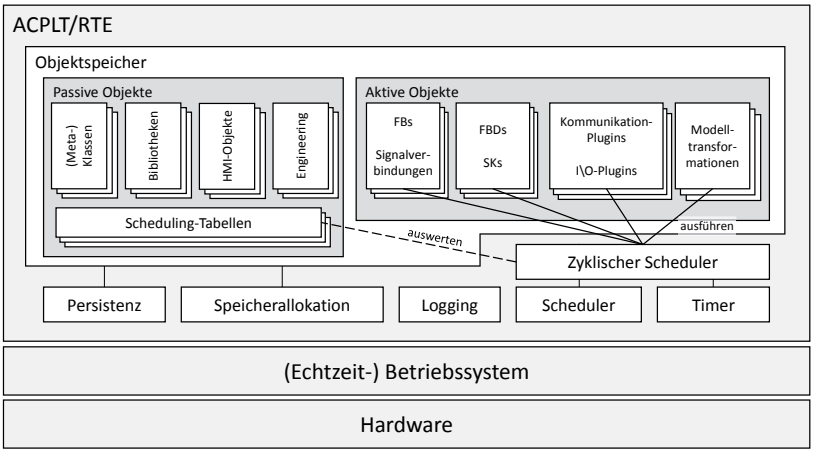


Abbildung 3.1.: Architektur der ACPLT/RTE Laufzeitumgebung mit Fokus auf das Scheduling.

Das System ist in C99 umgesetzt, was eine hohe Flexibilität bezüglich der eingesetzten Betriebssysteme und Hardwareplattformen ermöglicht. Zu den aktiv unterstützten Systemen zählen Windows und Linux Betriebssysteme sowie diverse Industrie-PCs der Firmen WAGO und Siemens.

Eine Übersicht der Architektur des Laufzeitsystems ist in Abbildung 3.1 zu finden und ähnelt in groben Zügen dem Aufbau der bereits vorgestellten Laufzeitsysteme. Das Laufzeitsystem wird über einem Betriebssystem eingesetzt und stellt grundlegende „Dienste“ wie Speicherallokation, Persistenz und Hardware-Abstraktion der Anwendung zur Verfügung. Die Modularität des Systems erlaubt das Laden vieler Klassen-Bibliotheken und Instanzen. Diese Module können nicht nur auf der Anwendungsebene operieren, z. B. als Funktionsbausteine im klassischen Sinne der IEC 61131-3, sondern auch den Funktionsumfang des Systems erweitern, z. B. als Kapselung der Feldkommunikation durch Treiberbausteine oder als Implementierung eines Kommunikationsprotokolls.

Die Großzahl der Komponenten wird im Objektspeicher gelagert, dessen Persistenz von der Laufzeitumgebung sichergestellt wird. Alle Objekte in diesem Speicher unterliegen der Möglichkeit der Selbsterkundung. Es wird im Allgemeinen zwischen den passiven und aktiven Objekten unterschieden [Mer16]. Zu der ersten Kategorie gehören Objekte, die keinen ausführbaren Code beinhalten, wie z. B. die Klassen und die Objekte des HMI-Modells. Die zweite Kategorie beinhaltet beispielsweise die Funktionsbausteine (FBs), die FBDs und die selbstständigen Komponenten (SKs) mit den beinhalteten Algorithmen. Das Scheduling ist in der Anlehnung an die IEC 61131-3 über einen zyklischen Scheduler realisiert, dessen Scheduling-Tabelle sich ebenfalls im Objektspeicher befindet und somit zur Laufzeit einsehbar und änderbar ist. Der Scheduler ruft die einzelnen Komponenten nach dem Prinzip des hierarchischen Schedulings auf (vgl. Abschnitt 6.1.7).

Die Möglichkeiten der Reflexion beeinflussen den Entwicklungszyklus der Anwendungen für ACPLT/RTE. Im Unterschied zur klassischen harten Unterteilung in Engineering-

und Betriebsphasen eines Systems, können Anwendungen für ACPLT/RTE zur Laufzeit engineert und migriert werden. Auch die Bibliotheken mit Bausteintypen können zur Laufzeit geladen werden. Die Typenentwicklung geschieht in einer Eclipse-basierten IDE in C [GE13b]. Darüber hinaus eröffnet die Betrachtung des ausführbaren Codes als generische Objektstruktur Möglichkeiten für Modellinteraktionen der AdA [GWE14, WKS+16].

3.4. Prozedurbeschreibungssprachen für leittechnische Anwendungen

Da im Rahmen dieser Arbeit eine Prozedurbeschreibungssprache für ressourcenadaptive Anwendungen vorgestellt wird, folgt an dieser Stelle zunächst eine kurze Einführung in die Prozedurbeschreibungssprachen, die in der Domäne der Automatisierungstechnik eingesetzt werden. Die in diesem Kapitel dargestellten Sprachen beinhalten nur eine kleine Auswahl aus der breiten Palette der Sprachen in der Automatisierungstechnik und dienen zur Einordnung der im Rahmen dieser Arbeit erarbeiteten Konzepte. Eine umfassende Analyse der Prozedurbeschreibungssprachen ist in [Yu16, Sch16] zu finden.

Die Grenze zwischen der konzeptionellen Beschreibung einer Prozedur und deren ausführbaren Implementierung verschwimmt zunehmend, da sich die meisten der vorgestellten Sprachen für beide Aufgaben eignen. Damit ist eine Ähnlichkeit zu FBD festzustellen, die sowohl für die Spezifikation der Programmlogik, als auch für deren Implementierung eingesetzt wird. Aus diesem Grund wird an dieser Stelle auf die Beschreibung einiger spezialisierten Programmiersprachen, wie z. B. ECC aus IEC 61499, verzichtet.

3.4.1. Sequential Function Chart

Eine Möglichkeit der Prozedurbeschreibung ist die Nutzung von Sprachen aus den existierenden Normen. Aus der IEC 61131-3 ist die Sprache SFC bekannt, die für die Definition der Prozeduren auf zyklischen SPSs verwendet wird (vgl. Abschnitt 2.2.4).

SFC ist aus der Sprache GRAFCET [IEC13, SSF13] abgeleitet. Der syntaktische Aufbau von SFC ähnelt im Groben einem endlichen Automaten. Ein SFC besteht aus Schritten, Transitionen und gerichteten Kanten zwischen diesen. Zu Beginn der Ausführung ist der initiale Schritt des SFCs aktiv. Der Übergang entlang einer Transition von einem aktiven Schritt zum anderen kann nur dann passieren, wenn die mit der Transition verknüpfte Transitionsbedingung gültig ist. SFC erlaubt auch parallele Verzweigungen – es können also zu jedem Zeitpunkt auch mehrerer Schritte aktiv sein. Die Bedingungen können in unterschiedlichen Sprachen formuliert sein, z. B. in KOP oder ST. Ist keine der Transitionsbedingungen aktiv, so verbleibt der SFC in dem aktuell aktiven Zustand.

In [Bau04] wird zwischen zwei Ausführungssemantiken für SFC unterschieden: der Maximal-Progress und der Lock-Step Semantik. Die Maximal-Progress Semantik führt die Übergänge zwischen den Transitionen solange aus, bis keine Transition mehr wahr ist. Die Lock-Step-Semantik erlaubt dagegen nur einen einmaligen Übergang jedes aktiven Schrittes pro Aufruf, d. h. pro Zyklus der Laufzeitumgebung. Für zyklische Laufzeitsysteme gehört die Lock-Step Semantik der SFCs zum Stand der Technik. Das Einführen einer unterschiedlicher Semantik ähnelt der Differenzierung zwischen der in- und der multi-cycle Semantik für PLC-Statecharts (vgl. Abschnitt 3.4.2).

Ein SFC führt Aktionen aus, die mit dem aktiven Schritt verknüpft sind. Die Wirkung der Aktionen ist durch sogenannte Aktionsbestimmungszeichen beschrieben. SFCs operieren in einem Ausführungsrahmen (POU) und können somit die internen Werte manipulieren. Darüber hinaus haben SFCs die Möglichkeit, Eingänge bzw. Parameter anderer Bausteine auf der globalen Ebene zu manipulieren, was ihnen das Orchestrieren bausteinübergreifender Abläufe ermöglicht. Somit werden SFCs auch für die Definition der Abläufe innerhalb einer SPS eingesetzt.

Die Manipulationen der Bausteine führen auf der globalen Ebene zu Abhängigkeiten zwischen dem SFC und den manipulierten Bausteinen, welche die Portabilität der Anwendungen negativ beeinflussen [GE13b]. Darüber hinaus haben SFCs eine Reihe weiterer Nachteile, die bereits in [YGE13, Yu16] diskutiert werden. Dazu gehören die komplizierte Logik und die große Anzahl der Aktionsbestimmungszeichen der Schritte, mehrdeutige Ausführungssemantik [Bau04] und die fehlenden Möglichkeiten der Hierarchisierung einzelner Schritte.

Aus diesen Gründen wurde in den letzten Jahren an Sprachen gearbeitet, welche die Vorteile, vor allem die Möglichkeit der direkten Einbettung in operative Laufzeitsysteme, der SFCs beibehalten, jedoch besser in Bezug auf das Engineering und die formale Semantik sind. Zu den zwei Alternativen gehören die PLC-Statecharts [WRKVH10] (vgl. Abschnitt 3.4.2), sowie der am Lehrstuhl für Prozessleittechnik entwickelte Formalismus namens Sequential State Chart (SSC) [YGE13] (vgl. Abschnitt 3.4.3). Beide Prozedurbeschreibungssprachen lehnen ihre Syntax an Harel's Statecharts [Har87], eine in UML als UML-Statechart übernommene Prozedurbeschreibungssprache [RJB04], an. Eine Anwendung der an die Statecharts angelehnter Sprache zum Zweck der Prozedurbeschreibung wurde bereits in den 90er Jahren vorgeschlagen [BCK98].

3.4.2. PLC-Statecharts

Die PLC-Statecharts wurden in [WRKVH10, Wit12] vorgestellt und passen die operative Semantik der UML-Syntax an den Ausführungszyklus einer SPS-ähnlichen Laufzeitumgebung an. Die Anwendung des Modells der zyklischen Ausführung erlaubt den Verzicht auf die ereignisorientierte Ablaufkontrolle innerhalb der PLC-Statecharts.

Es existieren zwei grundlegende Ausführungssemantiken für PLC-Statecharts. Durch die Einbettung in den Takt der Laufzeitumgebung kann genau ein Schritt pro Zyklus besucht werden, d. h. nur ein Zustandsübergang pro Zyklus ist möglich. Diese Semantik wird als multi-cycle Semantik bezeichnet. Zusätzlich zu der multi-cycle Semantik kann auch eine in-cycle Semantik für PLC-Statecharts eingeführt werden. Diese wird in Abschnitt 6.2.3 detailliert erläutert.

Aufgrund der Parallelen zum Abschnitt 3.4.1 folgt eine kurze Übersicht über die beiden Arten der Semantik:

> *In-cycle bzw. Maximal-Progress Semantik:* pro Zyklus der Laufzeitumgebung können mehrere Schritte des SFCs bzw. Zustände des Statecharts besucht werden.

> *Multi-cycle bzw. Lock-Step Semantik:* pro Zyklus der Laufzeitumgebung kann nur ein Schritt des SFCs bzw. Zuständ des Statecharts besucht werden.

Ähnlich zu ihren Prototypen, den UML-Statecharts, unterstützen PLC-Statecharts die Hierarchisierung der Zustände. Darüber hinaus beinhaltet die Definition der PLC-Statecharts ein Modell zur formalen Verifikation des Verhaltens des Statecharts mithilfe von UPPAAL. Dieses Modell beinhaltet eine minimalistische Abbildung des zyklischen Abtastverhaltens einer SPS durch einen UPPAAL-Automaten. Die formale Definition der Semantik adressiert eine Schwäche von SFCs.

Eine Referenzimplementierung der PLC-Statecharts erfolgte als ein Plugin für CODESYS V3 – einer industriellen IEC 61131-3 Programmier- und Laufzeitumgebung.

3.4.3. Sequential State Chart

Die Prozedurbeschreibungssprache SSC wurde in [YGE13] vorgestellt und adressiert einige Aspekte, die in der Definition der PLC-Statecharts unterspezifiziert wurden [Yu16].

Der erste Aspekt ist ein Ausführungsrahmen, der eine Kapselung des SSCs als ein FBN und dessen signalorientierte Verbindung zu weiteren Bausteinen ermöglicht. Ein weiterer Aspekt ist die Feinbeschreibung der Ausführungssemantik des Statecharts. Dazu gehört z. B. eine angepasste Semantik der Aktionen in der do- und exit-Phasen eines Zustands (so werden in SSC die Transitionsbedingungen nicht beim ersten Betreten des Zustands überprüft). Das formale Verhalten von SSCs wurde basierend auf dem in [WRKVH10] eingeführten UPPAAL-Modell beschrieben, damit die feinen semantischen Unterschiede klar erkennbar sind.

Eine weitere Eigenschaft der SSCs ist die Möglichkeit der Beschreibung von Service-orientierten Interaktionen aus den Aktionen der Zustände hinaus, sowie der Anschluss des Ausführungsrahmens an eine SOA, die mittels nachrichtenorientierter Kommunikationen umgesetzt wurde [YGE13].

Die syntaktischen Bestandteile der SSCs basieren sowohl auf der Syntax von Statecharts als auch auf der Syntax von IEC 61131-3 SFCs, die folgenden Vereinfachungen unterzogen wurden [YGE13, Yu16]:

- **Keine Nebenläufigkeit:** Da die SSCs mit dem Ziel einer strikten deterministischen Ausführung im Kontext der Laufzeitsysteme ohne hardwareunterstützte Nebenläufigkeit entwickelt wurden, wird deren Syntax bewusst ohne Parallelverzweigungen ausgelegt. Dem Programmierer bleibt somit nur die Möglichkeit Aktionen sequenziell in einem Zustand ablaufen zu lassen. Dies stellt die Ausführungsreihenfolge der Aktionen sicher.

- **Keine explizite Orthogonalität:** Eine ähnliche Argumentation gilt für den Verzicht auf explizite Modellierung orthogonaler Bereiche. Die Orthogonalität lässt sich durch eine sequentielle Ausführung der Unterkomponenten „simulieren“.

- **Keine Aktionen in den Transitionen:** Die Syntax der Statecharts erlaubt sowohl Ausführung von Zuweisungen bzw. Aktionen in den Transitionen als auch in den Zuständen (genauer, in den „entry“, „do“ und „exit“ Phasen der Zustände). Somit zeigen die Statecharts das Verhalten der Mealy- [Mea55] und der Moore-Automaten [Moo56] bezüglich der Ausgabe. Da SSCs aber die Syntax von IEC 61131-3 SFCs unterstützen sollten, die nur Moore-Verhalten zeigen (also keine Aktionen in den Transitionen), wird auch nur dieses Verhalten in SSC übernommen.

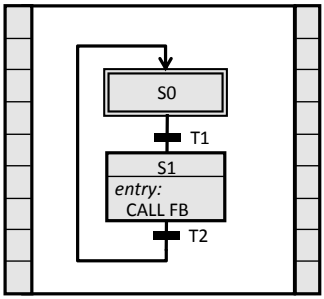


Abbildung 3.2.: Ein einfaches SSC im POU-Ausführungsrahmen.

- **Verzicht auf die Hierarchisierung der Zustände:** Explizite hierarchische Zustände werden von SSC nicht unterstützt und sollen mithilfe eingebetteter Unterfunktionen realisiert werden.

Im Zuge der weiteren Arbeit an der SSC-Spezifikation wurden die Statechart-Merkmale durch das Entfallen der do- und exit-Phasen der Zustände weiter vereinfacht [WKS+16].

Ein Beispiel für die graphische Darstellung eines SSCs ist in Abbildung 3.2 zu finden. Das SSC besitzt zwei Zustände, der Zustand „S1“ initiiert dabei einen Aufruf des Bausteins „FB“ innerhalb seiner „entry“-Ausführungsphase.

Eine Referenzimplementierung der SSCs wurde für die Laufzeitumgebung ACPLT/RTE umgesetzt und in mehreren Industriekooperationen erfolgreich validiert [WE15, Yu16].

4. Anforderungsanalyse und -spezifikation

Die Untersuchungen zum Stand der Wissenschaft und Technik in Kapitel 3 zeigen, dass in der Domäne der Automatisierungstechnik viel Interesse an flexibler, verteilbarer Anwendungsstruktur besteht. Zu diesem Thema wurden viele Beiträge geleistet, die teilweise in bereits auf dem Markt verfügbaren Produkten zu finden sind. Dazu gehören z. B. durch SOA lose gekoppelte Komponenten (vgl. Abschnitt 3.2.1) oder Agentensysteme (vgl. Abschnitt 3.2.2). Die Aspekte der adaptiven Anwendungseigenschaften (insbesondere temporaler Natur) und deren Anpassung wurden hingegen weitgehend außer Acht gelassen.

Aus diesem Grund werden in diesem Kapitel die Anforderungen an ein Modell für die adaptive Anpassung der temporalen Eigenschaften leittechnischer Anwendungen erarbeitet.

Dazu werden zunächst die (domänen-)spezifischen Anforderungen aus der Perspektive der Automatisierungstechnik im Allgemeinen und der Prozessleittechnik im Speziellen analysiert. Im zweiten Schritt werden aus diesen allgemeinen Anforderungen die speziellen Anforderungen an die zu erstellenden Methoden zur Flexibilisierung leittechnischer Anwendungen in Bezug auf deren temporale Eigenschaften abgeleitet.

4.1. Analyse der domänenspezifischen Anforderungen

Im Folgenden werden die domänenspezifischen Merkmale der (Prozess-)Leittechnik vorgestellt, die für das zu definierende Konzept maßgeblich sind. Es wird kein Anspruch auf deren Einzigartigkeit erhoben, da sicherlich viele dieser Merkmale auch für andere Anwendungsdomänen zutreffen.

Lange Lebenszyklen der eingesetzten Komponenten Die unterschiedlichen Komponenten einer prozesstechnischen Anlage haben relativ lange Lebenszyklen. Nach [Li12] besitzen mechanische Komponenten einen Lebenszyklus von mehreren Dekaden, während elektrische/elektronische Anlagenkomponenten alle drei bis fünf Jahre erneuert werden. Die Softwareelemente der Prozessleitebene werden am häufigsten aktualisiert – die Lebenszyklen der einzelnen Elemente werden mit 6 bis 12 Monaten angegeben. Auch wenn die einzelnen Elemente häufiger aktualisiert werden, z. B. das Hinzufügen zusätzlicher Regelkreise, bleibt die Software-Plattform unverändert und ihr Lebenszyklus ist normalerweise mit dem der prozessnahen Komponenten bzw. der elektronischen Anlagenkomponenten vergleichbar. Die eigentliche anlagen- und prozessspezifische Logik ist dabei durch die semantische Kopplung an die Anlage noch beständiger. Die langen Lebenszyklen erfordern eine wandelbare Software, die über den gesamten Lebenszyklus der Anlage Anpassungen unterliegen kann [VHDF+14, VHDB13].

Lange Laufphasen ohne Unterbrechung Zusätzlich zu dem langen Lebenszyklus der Anlage an sich kommen lange Betriebsphasen hinzu, in denen die Anlage ohne Unterbrechung läuft bzw. laufen muss. Diese Phasen sind vor allem in der Prozessindustrie anzutreffen und dauern bis zu mehreren Jahren am Stück, z. B. für eine Destillationskolonne [Fel01]. Die Ausfälle des Produktionssystems während einer solchen Phase sind mit allen Mitteln zu vermeiden, denn auch die kürzesten Ausfälle sind mit erheblichen Folgekosten verbunden. Viele der Anlagen sind bereits automatisiert, daher erfordert jede Modernisierung eine ökonomische Rechtfertigung sowohl in Bezug auf direkte Kosten, als auch für die durch die Modernisierungsmaßnahme entstehenden Kosten der Betriebsunterbrechung [Kop11a].

Konsequenzen der Störungen Als Ergänzung des vorhergehenden Punktes ist es wichtig zu bemerken, dass auch die kürzesten Ausfälle erhebliche Kosten produzieren können. Anders als bei der Fertigungstechnik, nimmt das Anfahren der Anlage nach einem ungeplanten Stillstand eine signifikante Zeit in Anspruch, während der kein (verkaufsfähiges) Produkt produziert werden kann. Auch die Konsequenzen einer Störung können sowohl für den Menschen, als auch für die Umwelt, deutlich gravierender sein als in anderen industriellen Bereichen. Aus diesen Gründen gilt die Automatisierungsbranche als „sehr konservativ“ und für neue Technologien „nicht empfänglich“ [KHAJ05].

Heterogene Nutzerkreise Da die Automatisierungstechnik eine klassische interdisziplinäre Domäne ist, haben die beteiligten Nutzer sowohl unterschiedlichste Hintergründe als auch Qualifikationen vorzuweisen. So müssen z. B. unterschiedliche Kenntnisstände der Nutzer in Regelungstechnik und im Bereich des Software Engineering berücksichtigt werden [VHDB13, Anforderung T6]. Zusätzlich kommt das komplexe Zusammenspiel der Technologiehersteller, Engineering-Dienstleister und der Endnutzer dazu (vgl. Abschnitt 2.2.8). Aus diesem Grund wird eine Unterteilung der Nutzer in unterschiedliche Gruppen in den NAMUR-Empfehlungen, wie z. B. in [NAM14], vorgenommen.

Jede Anlage ist ein Unikat Normalerweise handelt es sich bei jeder prozesstechnischen Anlage um ein Unikat [Kop11a], das mit einem nicht zu vernachlässigbaren Aufwand errichtet wurde. Obwohl die Anlagen aus einer Menge der in ihrer Größe und Komplexität variierenden Grundbausteine aufgebaut werden, bleiben viele Designentscheidungen und Lösungen auf die individuelle Anlage zugeschnitten. Somit erfordert jede Anlage eine speziell auf sie angepasste Betriebs- und Wartungsstrategie.

Bestehende bewährte Ansätze und domänenspezifische Sprachen Wie in jeder Domäne, existiert auch in der Leittechnik eine historisch geprägte Landschaft aus Sprachen (Anforderung M1 aus [VHDB13]), bewährten Ansätzen und etablierten Mustern. Das Befolgen dieser Muster und Ansätze ist eine Grundvoraussetzung für die Akzeptanz einer Technologie bzw. eines Produkts (Anforderung R3 aus [FVHF+15]) innerhalb der Domäne. Die Anforderung bezieht sich nicht nur auf die eingesetzte Technologie, sondern auch auf die Terminologie, die z. B. von den relevanten Normen oder dem jeweiligen Nutzerkreis stark geprägt wird. Als Beispiel dienen die SPS Programmiersprachen aus der IEC 61131-3 (vgl. Abschnitt 2.2.4).

Der Mensch als aktiver Systemgestalter Prozesstechnische Anlagen werden zentral aus einer Leitwarte gesteuert [Fel01], in der Anlagenfahrer [VDI15a] die Aufgaben der Prozessführung übernehmen bzw. überwachen. Auch wenn die Anlage automatisiert betreibbar ist, haben die Bediener jederzeit die Möglichkeit alle Aspekte der Prozessführung zu beobachten bzw. die Automatik im manuellen Modus zu überstimmen. Die komplexen Vorgänge werden teilweise immer noch manuell gesteuert [Fel01]. Die Rolle des Menschen wird auch in den Umsetzungsempfehlungen für das Zukunftsprojekt „Industrie 4.0“ mehrfach unterstrichen: „Der Mensch steht im künftigen smarten Produktionssystem im Mittelpunkt und die Technik soll seine kognitive und physische Leistungsfähigkeit durch die richtige Balance von Unterstützung und Herausforderung fördern“ [KWH13].

Neben diesen Merkmalen müssen zusätzlich die Unterschiede zwischen der Automatisierungstechnik im Kontext der Verfahrens- und der Fertigungstechnik berücksichtigt werden (vgl. Abschnitt 2.1.1).

4.2. Anforderungsspezifikation

Basierend auf den Ergebnissen der Anforderungs-Analyse, werden in diesem Abschnitt Anforderungen an das zu erarbeitende Modell zur Flexibilisierung der leittechnischen Anwendungen spezifiziert und in Kategorien der funktionalen sowie der nicht-funktionalen Anforderungen unterteilt.

4.2.1. Funktionale Anforderungen

Die funktionalen Anforderungen spezifizieren die gewünschten Funktionalitäten eines Systems [Gli07] und beantworten die Frage: „Was soll das System leisten?“.

(F1) Anpassung der Struktur und (F2) Anpassung der Eigenschaften einer Anwendung zur Laufzeit Anlagenänderungen, insbesondere Änderungen der Softwarekomponenten, müssen wegen der langen Betriebsphasen im laufenden Betrieb vorgenommen werden [Epp12, VHDF+14]. Dies sollte bereits bei der Entwicklung der Systeme berücksichtigt werden. Adaptive Softwarearchitekturen wurden auch im Kontext der CPS gefordert [KRS12]. Sowohl die Struktur der Anwendung (F1), als auch deren Eigenschaften (F2), müssen änderbar sein. Dieses muss insbesondere für anwenderspezifische Funktionen (Anforderung T2 aus [VHDB13]) und unter den Gesichtspunkten des Ressourcenverbrauchs gelten. Ein Beispiel für eine Strukturänderung kann das Hinzufügen einer Funktion sein, die den Funktionsumfang der Anwendung erweitert, z. B. das Hinzufügen eines neuen Reglers. Diese Änderungen bzw. Anpassungen gehen somit weit über das heute übliche Maß der Anpassung bestimmter Parameter einer leittechnischen Anwendung hinaus. Die Änderung der Eigenschaften einer Anwendung verändern Eigenschaften beinhalteter Funktionen, ohne deren Anzahl zu verändern. So kann z. B. die Regelgüte eines Reglers durch ein verbessertes Einschwingverhalten erhöht werden.

(F3) Anpassung an die äußeren Bedingungen (Adaptivität) Unter den äußeren Bedingungen sind im Kontext dieser Arbeit vor allem die (Rechen-)Ressourcen, die einer Anwendung zur Verfügung stehen, gemeint. Die potentiellen Möglichkeiten einer Struktur-

bzw. einer Eigenschaftsanpassung reichen für ein adaptives System nicht aus. Es müssen zusätzlich eine Reihe von Regeln bzw. Algorithmen definiert sein, die diese Anpassungen als Reaktion auf Änderungen der äußeren Bedingungen einleiten bzw. überwachen. Diese Regeln betreffen sowohl die Struktur als auch die Eigenschaften einer Anwendung. Diese Fluktuationen können nicht nur von transienter sondern auch von dauerhafter Natur sein. So kann z. B. eine zusätzliche Applikation dauerhaft auf einer Plattform installiert werden, was zu einer Abnahme der verfügbaren Ressourcen führt. Auf der anderen Seite kann die gesamte Hardware-Plattform im Zuge einer Modernisierung aufgerüstet werden – die Software muss von dieser Änderung ohne manuelle Rekonfiguration profitieren können.

(F4) Echtzeitfähigkeit Die Fähigkeit bestimmte Zusicherungen bezüglich des Zeitverhaltens eines Programms einhalten zu können, ist für die Beherrschung technischer Prozesse essentiell. Die Softwaresysteme müssen auch während des Rekonfigurationsvorganges diese Zusicherungen einhalten.

4.2.2. Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen beschreiben die Eigenschaften eines Systems [Gli07, CNYM12] und beantworten die Frage: „Wie soll das System die funktionalen Anforderungen umsetzen?“. Im Vergleich zu der Liste der funktionalen Anforderungen, ist die Liste der nicht-funktionalen Anforderungen länger und stärker an das Gebiet der Prozessleittechnik angepasst. Einige der aufgelisteten Aspekte finden sich in den standardisierten Richtlinien für Bewertung der Softwarequalität wieder [ISO05].

(N1) Kompatibilität mit existierenden Laufzeitumgebungen Durch die Verarbeitung analoger Signale sind die meisten Systeme und deren Architekturen auf den Einsatz zeitlich gesteuerter Systeme ausgelegt [VHDB13, Fel01]. Die eingesetzten Laufzeitumgebungen der PNKs folgen den Paradigmen der zyklischen Ausführung und der Softwarearchitektur der IEC 61131-3. Die zu erstellenden Konzepte dürfen diese Grundsätze nicht verändern und müssen mit bereits existierenden Laufzeitsystemen kompatibel bzw. mit minimalen Änderungen anpassbar sein.

(N2) Kompatibilität mit kooperativem Scheduling Auch wenn die IEC 61131-3 keine Aussage bezüglich des Einsatzes eines bestimmten Scheduling-Verfahrens macht, werden in den Umsetzungsempfehlungen [IEC03b] das unterbrechende und das kooperative Scheduling miteinander verglichen (vgl. Einführung in Abschnitt 2.1.2). Die Vorteile des kooperativen, von einer Taskliste gesteuerten Schedulers, sowie eine einfache Implementierung und die Minimierung des Laufzeitaufwands, sprechen für den Einsatz eines solchen Systems für die Laufzeitumgebungen der PNKs.

(N3) Minimaler Konfigurations- bzw. Migrationsaufwand Die Bewertung der Komplexität der Konfiguration und der Migration steht aus Anwendersicht im Mittelpunkt der Bewertung des gesamten Prozessleitsystems [TM09a]. Eine steigende Anzahl von Werkzeugen und Oberflächen überfordert die Endnutzer und schafft Probleme bei der Nutzerakzeptanz [Jor11] des Systems bzw. der Lösung. Bei der Bedienung komplexer Funktionalität sollte der Nutzer, wenn möglich, durch ein Assistenzsystem unterstützt werden. Der zu

erstellende Vorschlag soll keinen „harten Bruch“ in Bezug auf die bestehende, anwendungsspezifische Software darstellen [KM05]. Im Idealfall müssen die bereits existierenden Anwendungen bei dem ersten Systemeinsatz übernommen werden können und ohne weiteres Zutun einsetzbar sein (zunächst auch ohne die zusätzliche Funktionalität, z. B. ohne flexibles Zeitverhalten).

(N4) White-Box Engineering Dem Nutzer sollte es möglich sein, möglichst viele Aspekte des Systems zu untersuchen bzw. manuell verändern zu können. Diese nutzerorientierte Vorgehensweise erfordert eine White-Box Implementierung, d. h. eine Implementierung, die mit Mitteln der Introspektion und der Reflexion erkundbar bzw. änderbar ist. Diese Struktur lässt nicht nur Änderungen durch den Nutzer, sondern auch durch andere Systeme, z. B. regelbasierte Eingriffe oder andere Änderungen der Anwendungsstruktur im Rahmen der AdA zu.

(N5) Verwendung akzeptierter Programmiersprachen bzw. Konzepte Das einfache Engineering und die Nutzerakzeptanz setzen die Verwendung der in der Anwendungsdomäne akzeptierter Programmier- bzw. Beschreibungssprachen voraus. Denn nur solche Sprachen bieten eine Voraussetzung für die hersteller- und branchenübergreifende Nutzung und Dokumentation der erstellten Konzepte oder Programme. In der Domäne der Prozessleittechnik eignen sich für diese Zwecke insbesondere die graphischen Programmiersprachen der IEC 61131-3 [JT00, TM09a] (vgl. Abschnitt 2.2.4).

(N6) Breite Anwendbarkeit Die Methoden und Konzepte sollen nicht nur im Kontext einer bestimmten Aufgabe aus der Domäne der Leittechnik, wie z. B. der Implementierung eines Reglers einsetzbar sein, sondern sich für mehrere Anwendungsszenarien eignen. Eine weitere Möglichkeit der Erweiterung des Einsatzgebietes ist die Nutzung des Systems über die Grenzen einzelner Schichten der Automatisierungspyramide hinweg, z. B. zwischen der Betriebsleitebene und der Prozessleitebene.

5. Analyse der Ansätze für Anwendungen mit flexiblen temporalen Eigenschaften

In diesem Kapitel werden existierende Ansätze für Anwendungen mit flexiblen Eigenschaften und deren operative Implementierung vorgestellt. Die Ansätze wurden in zwei Kategorien aufgeteilt. Die erste Kategorie in Abschnitt 5.1 beinhaltet Vorgehensweisen zur dynamischer Änderung der temporalen Eigenschaften (vor allem der Ausführungszeit) einzelner Anwendungskomponenten. Die zweite Kategorie in Abschnitt 5.2 fasst Verfahren zur flexiblen Anpassung der Zykluszeit der Komponenten in zyklischen Systemen zusammen. Die Ausführungszeit in jedem Zyklus bleibt dabei unverändert.

Im zweiten Schritt werden Potentiale dieser Ansätze aus dem Bereich der Echtzeitsysteme in Bezug auf die Anwendbarkeit im Bereich der Automatisierungstechnik in Abschnitt 5.3 untersucht. Die Eignung wird anhand der nicht-funktionalen Anforderungen aus Abschnitt 4.2.2 beurteilt.

Durch den Fokus auf die temporalen Eigenschaften steht primär die Anforderung der Flexibilisierung der Anwendungseigenschaften (F2) aus Abschnitt 4.2.2 im Mittelpunkt der Betrachtung. Trotzdem darf die Struktur der Anwendung nicht vernachlässigt werden. Schließlich dient sie als Bezugspunkt für die Komponenten mit flexiblen Eigenschaften. Aus diesem Grund werden auch Ansätze für das Scheduling flexibler Komponenten und deren Kopplung mit in die Betrachtung einbezogen (insbesondere in Abschnitt 5.1.6).

Darüber hinaus sind alle vorgestellten Ansätze mit der Ausführung in Echtzeit kompatibel und erfüllen somit a priori die funktionale Anforderung F4.

5.1. Dynamische Änderung temporaler Eigenschaften einzelner Anwendungskomponenten

Im Allgemeinen sind zwei Vorgehensweisen für die Flexibilisierung des Zeitverhaltens einer Komponente erkennbar, die in jedem der vorgestellten Ansätze zu finden sind [GGH12]:

- Anpassung der Ausgabequalität der Komponente an die im Voraus bekannte Ressourcenverfügbarkeit und
- inkrementelle Verbesserung der Ausgabe bei unbekannter Ressourcenverfügbarkeit.

Unter Ressourcennutzung wird in dieser Arbeit primär die vom Algorithmus benötigte Rechenzeit verstanden. Der erste Ansatz passt die Qualität der Ausgabe eines Algorithmus an die festgelegte Rechenzeit an. Das Ziel ist hier z. B. eine möglichst hohe Genauigkeit einer numerischen Approximation der Lösung einer Differenzialgleichung bei einer vorgegebenen

Rechenzeit zu erreichen. Die zweite Vorgehensweise hält zu jedem Zeitpunkt eine „akzeptable“ Lösung bereit, die inkrementell verbessert wird, sodass die Berechnung jederzeit abgebrochen werden kann.

Neben der Qualität der Ausgabe können auch weitere Kriterien über den Ressourcenverbrauch eines Algorithmus entscheiden. So kann, z. B. die Laufzeit eines Algorithmus von bestimmten Eigenschaften der Eingabe abhängen (neben deren Größe). Sollte diese Eigenschaft im Voraus ermittelbar bzw. manipulierbar sein, so kann das System die Ressourcennutzung des Algorithmus vorhersagen bzw. beeinflussen.

5.1.1. Adaptive Algorithmen

Adaptive Algorithmen sind Algorithmen, deren Laufzeit nicht nur von der Größe der Eingabe, sondern auch von einem bestimmten Qualitätsmerkmal dieser abhängt [ECW92]. Ein Beispiel für eine Klasse solcher Algorithmen sind Sortieralgorithmen, die auf den „fast sortierten“ Probleminstanzen weniger Rechenzeit benötigen als auf den komplett ungeordneten Instanzen gleicher Größe. Adaptive Verfahren eignen sich insbesondere für das Problem des Sortierens, da viele Probleminstanzen in der Praxis bereits nah an der gewünschten Reihenfolge vorsortiert sind. Das Qualitätsmerkmal, von dem die Laufzeit zusätzlich abhängt, ist in diesem Fall die „Unordnung“ der Probleminstanz.

Eine Möglichkeit die Unordnung quantitativ auszudrücken, ist die Messung der Anzahl der Inversionen in der Eingabesequenz $(x_1, \ldots, x_n)$. Ein Paar (i, j) ist eine Inversion genau dann, wenn $i < j$ und $x_i > x_j$ [ECW92]. Die Anzahl der Inversionen für eine bereits sortierte Folge ist null und hängt nur von der Ordnung der Elemente in der Eingabesequenz ab.

Der Wert des Qualitätsmerkmals muss dem Algorithmus nicht im Voraus bekannt sein, sondern wird zur Laufzeit von ihm „entdeckt“. Dieses geschieht z. B. durch die Anzahl der Tauschoperationen während des Sortiervorgangs. Somit legt der Wert des Qualitätsmerkmals, neben der Eingabegröße, die tatsächliche Laufzeit des Algorithmus fest und kann in die Analyse seiner Laufzeit einbezogen werden. Diese Analyse hängt stark von der genauen Definition des Qualitätsmerkmals und dem untersuchten Algorithmus ab.

5.1.2. Anytime Algorithmen

Anytime Algorithmen [DB88] haben die Möglichkeit, Rechenzeit gegen Qualität der Ausgabe „einzutauschen“ und besitzen folgende charakteristische Merkmale [Zil96, Gra96, Kop11a]:

- **Qualitätsmaß der Lösung:** Der Algorithmus liefert neben dem Ergebnis auch eine Bewertung dessen Qualität. Dieser Wert muss relativ einfach zur Laufzeit berechenbar sein und kann von den abhängigen Systemen evaluiert werden. Die Qualität muss kein quantitativer Wert sein.

- **Vorhersagbarkeit:** Der Zusammenhang zwischen der Laufzeit und der Qualität der Ausgabe muss für die Ressourcenzuteilung bekannt bzw. vorhersagbar sein. Diese Information kann entweder über eine Funktion der Qualität in Abhängigkeit von der Laufzeit oder in Form einer Wahrscheinlichkeitsverteilung vorliegen.

- **Monotonie:** Die Qualität der Lösung verbessert sich mit der steigenden Laufzeit des Algorithmus monoton.

- **Unterbrechbarkeit und Fortsetzung:** Anytime Verfahren können jederzeit unterbrochen werden und geben die bis zu dem Zeitpunkt beste Lösung aus. Ebenso kann die Berechnung nach der Unterbrechung weiter fortgesetzt werden – die partiellen Ergebnisse unterliegen dabei weiterer Verfeinerung.

Neben der im letzten Punkt erwähnten Unterbrechbarkeit des Algorithmus, existiert noch ein zweiter Betriebsmodus der Ausführung über eine vereinbarte Zeit [Gra96] bzw. über diese Zeit hinaus. Wegen der Monotonie kann das Verfahren die zusätzliche Rechenzeit sinnvoll nutzen und die Lösung verbessern.

Im Allgemeinen kann man die inkrementelle, monotone Verbesserung der Lösung als das Hauptmerkmal dieser Klasse der Verfahren ansehen. Veröffentlichungen zu Anytime Algorithmen haben deswegen die Entwicklung solcher Verfahren für bereits bekannte Probleme im Fokus. Typischerweise wird nur die Rechenzeit als Ressource angesehen.

5.1.3. Ressourcenadaptive Algorithmen

Ressourcenadaptive (Resource-Aware oder „multi-fidelity“ [PSGS04]) Algorithmen haben die Möglichkeit, ihr Verhalten (z. B. über die Änderung der Parameter) in Abhängigkeit von bekannten Einschränkungen bzw. der Verfügbarkeit von Ressourcen zu variieren. Die Variation kann z. B. die Qualität der Lösung oder die benötigte Rechenleistung betreffen.

Anwendungen dieser Algorithmusklasse sind vor allem im Bereich des Cloud-Computings und der drahtlosen Netzwerke zu finden. Zum einen unterliegt die verfügbare CPU-Zeit Schwankungen. Zum anderen ist die verfügbare Bandbreite oder die Verbindungsqualität der Netzwerkanbindung betroffen. Andere Anwendungsbeispiele kommen z. B. aus der Softwareverifikation, wobei der verfügbare Arbeitsspeicher oft der limitierende Faktor ist [ATV08].

Die Umsetzung eines ressourcenadaptiven (RA) Ansatzes erfordert die Implementierung einer Ressourcenüberwachung, um die verfügbaren Ressourcen jederzeit erfassen zu können und eines Entscheidungsmechanismus, um das eigene Verhalten bzw. die Qualität der Lösung anzupassen [GGH12]. Die Formulierung der Regeln zur Anpassung kann entweder dem Endnutzer überlassen werden oder von den Algorithmen gekapselt sein – die tatsächlichen Mechanismen der Anpassung sind stark domänenspezifisch.

Ein Beispiel für ein RA-Framework aus dem Bereich der eingebetteten Systeme ist das ACTORS Projekt [BBE+11]. Das Ziel des Projekts ist die Entwicklung eines adaptiven Schedulers für das Linux Betriebssystem, der die Zuteilung der Ressourcen abhängig von den Anforderungen der Anwendungen anpassen kann. Die Zuteilung passiert anhand der vordefinierten diskreten Service-Levels der gemanagten Anwendungen, die im Dialog mit dem Ressourcen-Manager stehen. Die einzelnen Anwendungen können dem Ressourcenmanager ihren Bedarf an Ressourcen über die Angabe ihrer „Zufriedenheit“ mitteilen. Der Manager bekommt seinerseits die Übersicht über die verfügbaren Ressourcen des Systems. Im Projekt werden zwei Ressourcenklassen benutzt – die Bandbreite der CPU-Nutzung (d. h. der Anteil der gesamten CPU-Leistung, der der Anwendung zugeteilt wurde) und die maximale Wartezeit der Anwendung auf die Zuteilung der Ressource.

Es existieren weitere Beispiele für die Nutzung von RA-Algorithmen in Bereichen der Service-Orchestrierung [PSGS04, NK12], bei der Analyse von Datenströmen [GY06] oder im Rechencluster-Management [PCC+11].

Im Gegensatz zu Anytime Algorithmen steht bei ressourcensensitiven Ansätzen nicht die inkrementelle Verbesserung der Lösung im Mittelpunkt. So können z. B. bei vordefinierten Service-Levels unterschiedliche Versionen eines Algorithmus abgearbeitet werden, die die Zwischenlösung nicht beliebig verbessern. Darüber hinaus ist die Betrachtung von anderen Ressourcen als der Rechenzeit bzw. deren Kombination möglich. Als Beispiel dienen die „Power-Aware“ Echtzeitsysteme, die den Energieverbrauch des Systems berücksichtigen [AMM04].

5.1.4. Elastische Algorithmen

Als Kombination der ressourcensensitiven und Anytime Algorithmen für den Bereich des Cloud Computings wurden in [GGH12] die elastischen Algorithmen vorgeschlagen, in deren Definition der Begriff der Elastizität in seiner ökonomischer Bedeutung verstanden wird. Somit bedeutet die Elastizität das Maß der Änderung einer Variable der Kostenfunktion bei Änderung einer anderen Variable, z. B. wie reagiert der Preis der Cloud-Infrastruktur auf die Änderung der Nachfrage.

Die Definition der elastischen Algorithmen geht von einer inkrementellen Berechnung der Ergebnisse aus, wobei in jeder Iteration nicht nur der aktuell beste Wert des Ergebnisses, sondern ein zusätzlicher, für die inkrementelle Verbesserung notwendiger, Kostenwert berücksichtigt wird.

5.1.5. Imprecise Computation Model

Das Imprecise Computation Model wurde entwickelt um transiente, d. h. vorübergehende Überlastungen eines Echtzeitsystems ausgleichen zu können [LLB+94, But11]. In einem solchen Fall können die Ausführungszeiten der einzelnen Komponenten einer Anwendung reduziert werden, um zusätzliche Ressourcen verfügbar zu machen. Die Voraussetzung dafür ist aber ein auf diese Anpassbarkeit abgestimmter Entwurf der einzelnen Komponenten. Eine Möglichkeit dafür wäre z. B. die Nutzung von Anytime Verfahren, die durch die Komponenten gekapselt werden.

Das Imprecise Computation Model stellt eine Verallgemeinerung der Anytime Algorithmen dar [LLB+94] und hat nicht den eigentlichen Algorithmus bzw. seine Charakteristika, wie die inkrementelle Verbesserung der Ergebnisse, sondern die Aspekte der Organisation bzw. der Komposition solcher Algorithmen zu einem Gesamtsystem und dessen Betrieb innerhalb eines Betriebssystems im Fokus.

Die Vorgehensweise des Modells leistet einen positiven Beitrag zu der Fehlerrobustheit des Softwaresystems und ermöglicht die sogenannte „graceful degradation“, also eine angemessene Reaktion des Systems auf Fehler bzw. Überlastungen. In dem Kontext des Modells führt die Systemüberlastung zu einer schrittweisen Reduktion der Systemleistung, jedoch zu keinem Komplettausfall des Systems. Entfällt die Ursache für die Überlastung, kann die Systemleistung wieder aufgebaut werden, bis der optimale Systemzustand wieder hergestellt ist.

Die einzelnen Elemente des Modells sind in Abbildung 5.1 dargestellt. Die Rechtecke in der Abbildung stehen für einzelne Tasks, die während einer Berechnung ausgeführt

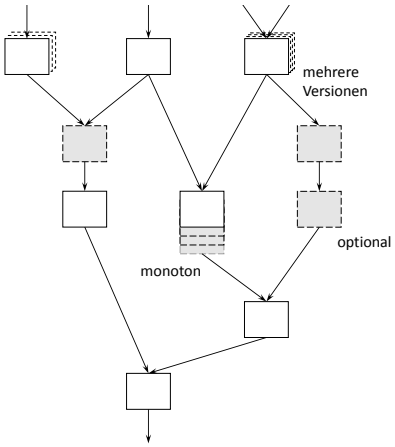


Abbildung 5.1.: Schematische Darstellung des Imprecise Computation Models [LLB+94].

werden. Die Pfeile in der Abbildung symbolisieren den Vorrang bzw. die Abhängigkeiten zwischen einzelnen Tasks. Im Allgemeinen werden drei „Grundmuster“ für den Aufbau eines Imprecise Computation Systems unterschieden:

- **0/1 Ausführung:** Unterschiedliche Tasks oder Systemkomponenten werden manuell als „verbindlich“ (weiß) oder „optional“ (grau) kategorisiert. Die optionalen Komponenten werden im Fall einer Überlastung bzw. Störung des Systems ausgelassen. Dabei wird entschieden, ob ein Task immer oder eventuell ausgeführt wird. Im Falle der Ausführung eines optionalen Tasks, muss das System sicherstellen, dass es genügend Rechenzeit für seine Durchführung bekommt. Das klassische Beispiel für optionale Tasks ist die eventuelle Neuberechnung bestimmter Werte, z. B. der Signalqualität oder der Messunsicherheit. Im Falle einer Überlastung können oft auch ältere Werte benutzt werden, die die Realität noch genügend genau abbilden.

- **Mehrversionen-Methode:** Von einem Rechenschritt werden mindestens zwei Varianten mit dem gleichen Interface zur Verfügung gestellt. Die Versionen unterscheiden sich in der Qualität des gelieferten Ergebnisses, benötigen dafür aber auch unterschiedliche Laufzeiten, was die Flexibilität des Schedulers erhöht. Ein Beispiel für die Mehrversionen-Methode ist die prozessbegleitende Simulation, die auf zwei unterschiedlichen Modellen, einem einfachen und einem komplexen, durchgeführt werden kann (vgl. Use-Case in Abschnitt 7.3). Während einer Systemüberlastung kann der Simulation-Task auf das einfachere Modell umschalten und die nachfolgenden Rechenschritte weiterhin mit aktuellen Daten versorgen.

  Die Analyse der Mehrversionen-Tasks kann auf die Betrachtung von 0/1 Ausführung reduziert werden: Die kürzeste Laufzeit einer Version wird dabei als verbindlich angesehen und die Laufzeitdifferenz zwischen den Versionen als optional.

- **Meilenstein-Methode bzw. Monotone Berechnung:** Monotone Berechnungen

„kapseln“ das Konzept der Anytime Algorithmen (vgl. Abschnitt 5.1.2). Dabei muss ein bestimmter Hauptteil verbindlich ausgeführt werden. Dieser liefert ein hinreichend gutes Ergebnis, das dann iterativ verbessert werden kann. Als Beispiel für monotone Berechnungen dienen viele Verfahren der numerischen Mathematik.

Die monotone Verbesserung kann entweder jederzeit abgebrochen werden oder in kurze ununterbrechbare Iterationen aufgeteilt werden, die das Ergebnis in diskreten Schritten verbessern. Die monotonen Berechnungen sind aus der Scheduling-Sicht am flexibelsten, da sie teilweise online gescheduled werden können [LLB+94].

Das Scheduling der Anwendung, die nach den oben beschriebenen Prinzipien aufgebaut ist, wird anhand der Fehlerfunktionen der einzelnen Tasks statisch oder dynamisch durchgeführt. Aus der Scheduling-Perspektive werden die einzelnen Tasks T_i jeweils in zwei Untertasks unterteilt: einen verbindlichen Task M_i und einen darauf folgenden optionalen Task O_i. Die Summe der WCET der Untertasks ergibt die WCET des ursprünglichen Tasks, falls der optionale Untertask kein monotoner Task ist. Monotone Tasks können beliebig lang ausgeführt werden und haben daher keine WCET. Der Scheduler muss dem verbindlichen Task die benötigte Ausführungszeit garantieren. Die Ausführungsdauer des optionalen Untertasks O_i wird von der Fehlerfunktion beeinflusst, die nur von dieser Dauer abhängig ist. Fehlerfunktionen können z. B. linear, konvex oder konkav sein. Bei terminierenden optionalen Tasks verschwindet der Fehler, sobald deren Ausführungszeit der WCET gleich ist. Bei monotonen Tasks konvergiert der Fehler mit fortschreitender Ausführungszeit gegen Null.

Ein häufiges Ziel für das Scheduling solcher Probleme ist die Minimierung der Summe der Fehler einzelner Tasks, die eventuell noch gewichtet sind, um Prioritäten der Tasks abzubilden. Für unterbrechendes Scheduling sind optimale polynomielle Algorithmen bekannt [LLS+91]. Das nicht-unterbrechende Scheduling-Problem ist NP-hart [LLS+91].

Die Forschungsarbeit im Bereich des Schedulings der Imprecise Tasks, basierend auf der Unterteilung in verbindliche und optionale Tasks, brachte ein Feld des „Reward-Based Scheduling“ hervor. Die Fehlerfunktionen für die Bewertung der Laufzeit der optionalen Tasks werden in diesem Feld durch Belohnung-Funktionen ersetzt. Die Belohnung hängt von der Laufzeit des O_i ab und kann, ähnlich zu der Fehlerfunktion, linear, konvex oder konkav sein. Die Problemstellung ist demnach ein gültiger Schedule, welcher die Summe der (gewichteten) Belohnungen der einzelnen Tasks maximiert. Eine Übersicht über die Komplexität einzelner Problemklassen und die verfügbaren Heuristiken im Rahmenwerk des Reward-Based Schedulings ist in [AMM04] zu finden.

Zusammengefasst bietet das Imprecise Computation Model drei unterschiedliche Grundmuster für die Definition der bedingt ausführbaren Programmlogik bzw. des Verhaltens einer Komponente, die im Zusammenspiel zwischen dem Programmierer (der die Tasks auslegt und einer der drei Kategorien zuordnet) und dem Laufzeitsystem (das die Laufzeiten der Tasks einplant und ausführt) zu einer adaptiven Systemreaktion auf Überlastungen führt. Trotz der ursprünglichen Auslegung des Modells auf die Amortisierung von Überlastungen, also der Ressourcenknappheit, kann der gleiche Ansatz auch für den sinnvollen Verbrauch von sonst ungenutzten Kapazitäten angewendet werden.

5.1.6. Predictably Flexible Real-Time Scheduling

Als Ergänzung zu den Scheduling-Verfahren für das Imprecise Computation Model wird ein dynamischer Scheduling-Ansatz vorgestellt, der die entstehende Slackzeit berücksichtigen und an Komponenten oder sporadische Tasks verteilen kann.

Dieser Ansatz für die Zusammenarbeit zwischen offline und online Scheduling-Verfahren wurde in [Foh12] unter dem Namen „Predictably Flexible Real-Time Scheduling“ vorgestellt. Die Vorteile der tabellenbasierten Scheduler, wie Vorhersagbarkeit und die einfache Implementierung, werden von den Nachteilen der fehlenden Flexibilität überschattet. Die flexibel arbeitenden online Scheduling Verfahren sind hingegen schwerer vorherzusagen und haben einen höheren Rechenaufwand während der Laufzeitphase.

Um von den Vorteilen beider Ansätze zu profitieren, schlägt das Modell eine Kombination aus online und offline Scheduling-Verfahren für das Systemdesign vor. Somit wird innerhalb eines Echtzeitsystems die Koexistenz der Scheduling-Ansätze für zeit- und ereignisgesteuerte Systeme ermöglicht.

In der offline Phase des Modells werden für die einzelnen Jobs eines Tasksets die erlaubten Zeitfenster berechnet, in denen die Ausführung des Jobs stattfinden darf, ohne die Echtzeitschranken zu verletzen.

In der online Phase bestimmt nun ein Algorithmus den tatsächlichen Ausführungszeitpunkt eines Jobs innerhalb des gültigen Zeitfensters. Die so gewonnene Flexibilität wird in dem Modell für die Ausführung aperiodischer Tasks angewendet, die ohne Anpassung der im Voraus kalkulierten Scheduling-Tabelle möglich ist.

5.2. Dynamische Änderung der Zykluszeit einzelner Anwendungskomponenten

Im Kontext der zyklischen Ausführung einzelner Anwendungskomponenten kann der Ressourcenverbrauch einer Komponente durch die Änderung ihrer Ausführungshäufigkeit bzw. ihrer Zykluszeit verändert werden. In diesem Abschnitt werden die gängigen Methoden für diese Anpassung vorgestellt.

5.2.1. Elastic Model

Die Grundannahme des Elastic Models [But11, BA02] geht von einer veränderlichen Periode einzelner Tasks aus, die in einem bestimmten Intervall variiert werden kann.

Ein elastischer Task ist ein periodischer Task, der neben der WCET w_i eine nominale per_{i_0} und eine maximale $per_{i_{max}}$ Periode, sowie einen Elastizitätskoeffizienten $e_i \geq 0$ enthält. Der Letzte gibt Auskunft über die Elastizität bzw. die Steifigkeit eines Tasks. Die Periode des Tasks kann frei in dem Intervall $[per_{i_0}, per_{i_{max}}]$ variiert werden und hat somit Einfluss auf die Utilization-Faktoren (vgl. Abschnitt 2.1.2) des Tasks und somit des Gesamtsystems.

Im Falle einer Systemüberlastung können die Utilization-Faktoren der Tasks durch die Variation ihrer Zykluszeit so angepasst werden, dass die gesamte Auslastung den gewünschten Vorgaben entspricht (z. B. eine bestimmte Schranke unterschreitet). Die Anpassung der Zykluszeit eines Tasks passiert somit bei dem Hinzufügen neuer Tasks, der Terminierung

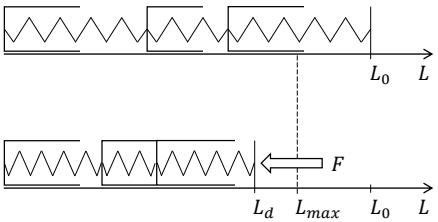


Abbildung 5.2.: Drei Tasks mit Längenbeschränkungen im Elastic Model dargestellt als elastische Feder [But11]. Oben: vor der Komprimierung (die Gesamtlänge $L_0 \geq L_{max}$), unten: nach der Komprimierung (die Gesamtlänge $L_d \leq L_{max}$).

eines periodischen Tasks oder einem Antrag auf eine Periodenänderung eines existierenden Tasks (im Rahmen des Intervalls). Alle Änderungen müssen durch einen Garantie-Algorithmus auf ihre Zulässigkeit (die Einhaltung der Lastvorgaben des gesamten Tasksets) überprüft werden.

Die Anpassung der Zykluszeit erfolgt anschaulich durch die Betrachtung der Menge der Tasks, als einer Menge elastischer Federn, deren Länge dem Utilization-Faktor des Tasks $\frac{C_i}{per_i}$ entspricht, wobei $per_i \in [per_{i_0}, per_{i_{max}}]$ die tatsächliche Periode des Tasks i ist.

Die Kompression der Jobs ist in Abbildung 5.2 schematisch dargestellt. Die Länge einzelner Federn L_d wird auf dem unteren Teil des Bildes durch die Ausübung der Kraft F verkleinert. Die Berechnung der einzelnen Federlängen ist ausgehend vom Kräftegleichgewicht des Systems analytisch möglich. Im Falle der Längenbeschränkungen einzelner Federn erfolgt diese Berechnung mithilfe eines iterativen Verfahrens. Mithilfe der gleichen Berechnungen können die Federn auch dekomprimiert werden – dies geschieht z. B. bei der Terminierung oder dem Hinzufügen eines Tasks. Die auf diese Weise errechneten Perioden der Tasks werden von dem untergeordneten unterbrechenden Echtzeit-Scheduler, z. B. einem EDF-Scheduler, realisiert.

5.2.2. Quality-of-Control-basierte Betrachtung

Eine Anwendung der flexiblen temporalen Eigenschaften aus dem Aufgabenfeld der Regelung bzw. der Integration der Regler in die Laufzeitsysteme wurde in [MFFR02, VFM03] vorgestellt. In diesen Veröffentlichungen wurden die Auswirkungen der dynamischen Änderung der Zykluszeit einzelner Tasks auf den Regelfehler u.a. am Modell eines inversen Pendels untersucht. Dieser Ansatz der Anpassung der Zykluszeit ist ähnlich zu dem Ansatz des Elastic Models (vgl. Abschnitt 5.2.1). Der Unterschied besteht in der Qualitätsbewertung und dem Auslöser für die Änderung der Zykluszeit eines Tasks.

Während im Fall des Elastic Modells Änderungen der Menge von Tasks die Änderung der Ausführungsraten auslösen, waren es in [MFFR02] eine Perturbation der Regeldifferenz. Somit wurde die Anpassung der Raten durch das physikalische System ausgelöst. In einem solchen Fall wurde der Regler so lange mit erhöhter Rate ausgeführt, bis ein stabiler Zustand der Strecke (wieder) erreicht wurde. Bei dieser Umverteilung der Ressourcen mussten Tasks mit kleinerer Priorität temporär mit einer kleineren Rate ausgeführt werden.

Die Zielsetzung des Elastic Modells ist die Existenz eines gültigen Schedules, d. h. eines

Schedules, in dem alle Tasks die Echtzeitschranken einhalten. Das Ziel der Quality of Control (QoC)-basierten Betrachtung ist hingegen die Minimierung der integrierten normierten Regelabweichung.

Die Integration der Konzepte aus den Bereichen der Regelungstechnik und der Scheduling-Theorie wird im Allgemeinen als „Control-Scheduling Codesign“ bezeichnet.

5.2.3. Job Skipping

Ein anderer Ansatz für die Variation der Ausführungsrate eines Tasks ist die Methode des Job Skippings, bei der bestimmte Jobs eines periodischen Tasks ausgelassen bzw. abgebrochen werden. Die auf diese Weise kurzfristig gewonnene Rechenzeit kann für andere Aufgaben eingesetzt werden, z. B. für die Ausführung sporadischer Tasks.

Ein Scheduling-Modell, das in der Lage ist die einzelnen Jobs auszulassen, wurde in [KS95] vorgestellt. In diesem Modell haben die Tasks T_i neben den Periode einen Skip-Parameter s_i, $2 \leq s_i \leq \infty$. Dieser Parameter gibt an, wie häufig Jobs ausgelassen werden dürfen. Dazu werden die Jobs eines Tasks in rote und blaue Jobs aufgeteilt. Ein roter Job darf nicht abgebrochen oder ausgelassen werden und muss rechtzeitig terminieren, ein blauer Job darf hingegen durch einen Abbruch oder eine Deadline-Überschreitung ausgelassen werden. Der Wert des Skip-Parameters s_i wird dann wie folgt gedeutet [BA02]:

- wurde ein blauer Job ausgelassen, so müssen die darauf folgenden $s_i - 1$ Jobs rot sein und
- wurde ein blauer Job nicht ausgelassen, so ist der darauf folgende Job ebenfalls blau.

Es kann gezeigt werden, dass das Schedulability-Problem NP-hart ist [KS95]. In [BA02] werden Schranken für das Scheduling der Tasksets mit Skip-Parametern mithilfe von EDF Scheduler aufgezeigt.

5.3. Diskussion in Bezug auf nicht-funktionale Anforderungen

Die bewerteten Ansätze wurden in Bezug auf das Objekt der Flexibilisierung in zwei Kategorien unterteilt. Die erste Gruppe der Ansätze hat das Ziel die Ausführungsdauer einzelner Tasks zu modifizieren, die zweite Gruppe die Zykluszeit der Tasks bei konstanter Ausführungszeit des einzelnen Tasks. Für die Anwendung in der Leittechnik sind die Ansätze aus allen zwei Bereichen interessant bzw. sinnvoll.

Aus den Ansätzen der ersten Gruppe bildet nur das Rahmenwerk des Imprecise Computation Models eine flexible Grundlage für den Einsatz im Bereich der Prozessleittechnik. Da es sich bei dem Modell nur um ein Rahmenwerk handelt, sind sowohl unterschiedliche Algorithmen als Grundbausteine, wie z. B. Anytime Algorithmen, als auch unterschiedliche Scheduling-Verfahren zu deren Ausführung, wie z. B. Reward-Based Scheduling oder Job Skipping, beliebig miteinander kombinierbar. Somit können auch die Ansätze der zweiten Gruppe in das Rahmenwerk eingebracht werden. Darüber hinaus kann das Konzept des Predictable Flexible Scheduling für Scheduler angewendet werden, deren online Phase für die Verteilung der anfallenden Slackzeit auf die einzelnen variablen Tasks übernehmen kann.

Die funktionalen Anforderungen des Kapitels 4 werden bis auf die Anforderung (F1) offensichtlich von dem Rahmenwerk abgedeckt. Die strukturellen Änderungen der Anwendung (z. B. das Hinzufügen neuer Komponenten) können problemlos durch die Mechanismen der 0/1 Ausführung in dem Rahmenwerk abgebildet werden.

Die in Kapitel 4 aufgestellten nicht-funktionalen Anforderungen werden wie folgt durch das Rahmenwerk des Imprecise Computation Models abgedeckt:

(N1) Kompatibilität mit existierenden Laufzeitumgebungen Durch die zyklische Ausführung der Logik innerhalb der Laufzeitumgebung lässt sich die verfügbare Ausführungszeit relativ einfach abschätzen bzw. sogar zur Laufzeit ermitteln. Somit ist die, für die optionale Komponenten verfügbare, Ausführungszeit klar abgrenzbar und eine technische Basis für die Ausführung dieser fast vollständig etabliert.

(N2) Kompatibilität mit kooperativem Scheduling Das Rahmenwerk setzt keinen spezifischen Scheduler voraus, und ist auch für das kooperative Scheduling anwendbar. Einige Elemente des Rahmenwerks erfordern dennoch Anpassungen. So kann z. B. eine Iteration einer monotonen Berechnung nicht zwischen den Meilensteinen abgebrochen werden.

Darüber hinaus erfordert der Einsatz von Ansätzen aus der zweiten Gruppe, z. B. Job Skipping, gewisse Anpassungen des Scheduling-Modells.

(N3) Minimaler Konfigurations- bzw. Migrationsaufwand Die bestehende Automatisierungssoftware kann durch die Anwendung der 0/1 Ausführung in das Rahmenwerk eingeführt werden. Die existierenden Tasks werden im Kontext des Rahmenwerks zu verbindlichen Tasks erklärt. Somit ist deren Ausführung gesichert.

Dieser einfache Schritt ist die Voraussetzung für eine Migration und ist praktisch ohne Engineering-Aufwand durchführbar. Die auf diese Weise entstandene Anwendung kann sukzessiv um zusätzliche komplexe Elemente, z. B. Komponenten mit monotonen Berechnungen, erweitert werden, die die vorhandene Slackzeit ausnutzen. Der Konfigurationsaufwand für den Nutzer ist typischerweise nur auf das Setzen der Prioritäten bzw. Gewichte einzelner Komponenten beschränkt.

(N4) White-Box Engineering Bereits die graphische Darstellung des Rahmenwerks in Abbildung 5.1 deutet auf eine gute Möglichkeit der Introspektion einzelner Komponenten hin. Die in Kapitel 6 vorgeschlagene Architektur zur Umsetzung des Rahmenwerks greift auf die bewährten Methoden des komponentenorientierten Softwareengineerings bzw. der auf Funktionsbausteinen basierenden Software zurück, die um einzelne Elemente des Imprecise Computation Models erweitert werden. Die bestehenden Engineering-Methoden sind somit weiterhin anwendbar.

(N5) Verwendung bekannter Programmiersprachen bzw. Konzepte Die Frage, wie die variablen Verhalten der einzelnen Komponenten, z. B. die 0/1 oder die monotone Ausführung beschrieben werden, ist nicht eindeutig gelöst. In der Literatur wird dieses Verhalten typischerweise in höheren Programmiersprachen, wie z. B. Java oder C++, beschrieben. Diese Sprachen sind weder domänenspezifisch für den Bereich der Leittechnik, noch eignen

sie sich für das White-Box Engineering (Anforderung N5). Eine domänenspezifische Sprache für die Beschreibung des Komponentenverhaltens, die auch White-Box Paradigmen unterstützt, wird in Abschnitt 6.2 eingeführt.

(N6) Breite Anwendbarkeit Das Rahmenwerk schränkt den „Inhalt“ und somit die Anwendung bzw. die Bestimmung der ausgeführten Komponenten nicht ein. Somit existieren keine Anwendungseinschränkungen, wie z. B. die Einschränkung des QoC auf den Bereich der Regelung.

6. Ein Rahmenwerk für die Integration von ressourcenadaptiven Anwendungen

In diesem Kapitel wird ein Rahmenwerk für die Beschreibung von RA-Anwendungen und deren nahtlose Einbettung in zyklische Laufzeitsysteme vorgestellt. Basierend auf den Ergebnissen der Analyse des vorherigen Kapitels, wurden die folgenden Ansätze für das vorgestellte Rahmenwerk ausgewählt:

- Imprecise Computation Model als grundlegende Architektur für den Aufbau von RA-Anwendungen,
- Predictably Flexbile Scheduling als das Scheduling-Prinzip, das die Vorteile von offline- und online-Scheduling vereint und
- Elastic Model für die heuristische Berechnung der vom Scheduler vorgegebenen Ausführungsdauer einzelner Komponenten im Rahmen des Schedulings.

Als Grundlage für die Koexistenz der RA-Anwendungen mit bereits eingesetzten Systemen dient eine flexible Softwarearchitektur des Laufzeitsystems, die in Abschnitt 6.1 unter dem Begriff „einheitliche Laufzeitarchitektur“ vorgestellt wird.

Um die Anforderungen des Engineerings für die Definition und die Implementierung zeitsensitiver Eigenschaften der Anwendung zu erfüllen, wird ein graphischer Beschreibungsformalismus eingeführt. Dieser weist eine große Ähnlichkeit zu den bereits verwendeten Prozedurbeschreibungssprachen auf. Das eingeführte Meta-Modell wird in Abschnitt 6.2 diskutiert.

Die Zuteilung der Zeitressourcen an die einzelnen Anwendungen erfolgt durch einen Systemscheduler, dessen Referenzarchitektur dem Prinzip des Predictably Flexible Schedulings folgt. Der Scheduler ist im Fokus des Abschnitts 6.3.

6.1. Einheitliche Laufzeitarchitektur

Bevor ein Scheduling-Problem aufgestellt und ein Algorithmus zu dessen Lösung definiert werden kann, muss eine Reihe von Annahmen über die auszuführenden Programmelemente und deren Abhängigkeiten getroffen werden. Dieses Ziel wird im Folgenden durch eine, an die beiden Programmiernormen IEC 61131-3 und IEC 61499 angelehnte, Softwarearchitektur für leittechnische Anwendungen erfüllt. Die vorgestellte Architektur ist Voraussetzung für eine flexible Programmausführung, sowie die Koexistenz der Ausführungskonzepte beider Standards.

Wie in Abschnitt 3.1 erwähnt, wird in dieser Arbeit die in [GE13b] als „einheitliche Laufzeitarchitektur“ eingeführte Systemarchitektur angewendet. Die Vorarbeiten haben

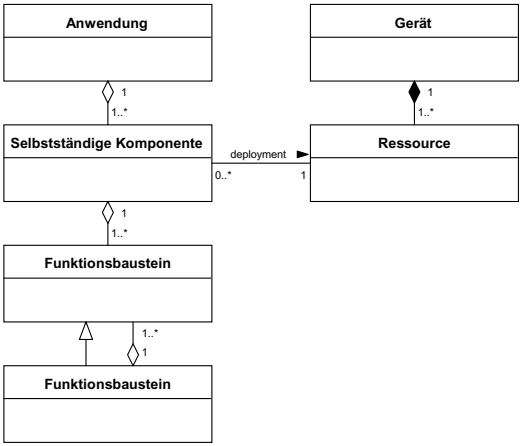


Abbildung 6.1.: Strukturelle Darstellung der einheitlichen Laufzeitarchitektur.

sich auf die Aspekte der Verteilbarkeit der Anwendungen fokussiert. Im Gegensatz dazu stehen im Rahmen dieser Dissertation die Scheduling-Aspekte der einzelnen Komponenten im Vordergrund.

6.1.1. Grunddefinitionen

Die Laufzeitarchitektur wird grob in eine Software- und eine Hardwarearchitektur unterteilt. Zunächst wird die Softwarearchitektur vorgestellt, deren Aufbau, mit der Ausnahme der selbständigen Komponenten, der Architektur der IEC 61499 (vgl. Abschnitt 2.2.6) ähnelt.

Die strukturelle Darstellung der Architektur in einer UML-artigen Notation ist in Abbildung 6.1 dargestellt. Die Softwarearchitektur besteht aus den folgenden Teilen:

- **Anwendung:** Eine Anwendung ist definiert als die Gesamtheit der Komponenten (Bausteine, Komponenten und deren Kommunikationsverbindungen), die einen gemeinsamen Zweck haben. Diese Definition verleiht der Anwendung eine gewisse Semantik und unterscheidet sich somit von der pragmatischen Definition einer Anwendung im Kontext der IEC 61499: „eine Anwendung [besteht] aus einem FBN, dessen Knoten Funktionsbausteine oder Unteranwendungen und deren Parameter sind und dessen Kanten Datenverbindungen und Ereignisverbindungen sind“.

  Die Anwendung ist ein logischer Namensraum, der zur Gruppierung von weiteren Strukturierungseinheiten, wie z. B. den selbstständigen Komponenten (vgl. Abschnitt 6.1.2), benutzt wird. Die Vorstellung der Anwendung als einen logischen „Container“, spiegelt die Möglichkeit der Wiederverwendbarkeit einzelner Anwendungsteile in anderen Anwendungen wieder. Somit können z. B. mehrere Instanzen eines Standardbausteintypen in unterschiedlichen Anwendungen Verwendung finden.

  Die Anwendung besitzt wegen ihrer logischen Natur keine direkten Schnittstellen

bzw. keinen Ausführungsrahmen. Für die Interaktion mit der Umgebung kann allerdings eine dedizierte Komponente als „Vertreter“ der Anwendung definiert werden. Existiert dieser, so definieren die Schnittstellen des Vertreters die Schnittstellen der Anwendung. In anderen Fällen wird die Vereinigung der beinhalteten Komponenten als die Anwendungsschnittstelle definiert.

- **Selbstständige Komponente:** Die selbstständige Komponente (SK) wird im Abschnitt 6.1.2 detailliert vorgestellt. Die SK ist eine neuartige POU-Klasse, die für die Kapselung der Verteilbarkeits- und der Schedulingsaspekte genutzt wird.

- **Funktionsbausteinnetzwerk:** Ein FBN enthält die Grundeinheiten der Anwendung bzw. der SK und spezifiziert Signalverbindungen zwischen diesen. Diese Beziehungen beschreiben zum einen den Datenfluss zwischen einzelnen POUs durch die Signalverbindungen. Zum anderen kann ein Bausteinnetzwerk Informationen über das Scheduling der enthaltenen POUs enthalten.

  Da sowohl die SKs, als auch die FBNs Informationen zum Scheduling enthalten, sind *beide POUs* als Plattform für den Einsatz bzw. die Kapselung der RA-Algorithmen bestens geeignet.

- **Funktionsbaustein:** Funktionsbausteine sind die Grundeinheiten der Anwendung. Sie kapseln die einzelnen Algorithmen und stellen die Schnittstellen zur Verfügung. Die IEC 61131-3 definiert zusätzlich noch Funktionen – das sind spezielle Bausteine ohne interne Speicher. Durch die Abwesenheit der Speicher müssen keine Instanzen der Funktionen gebildet werden. Es wird in dieser Arbeit keine explizite Unterscheidung zwischen Funktionsbausteinen und Funktionen vorgenommen.

Die vorgestellte Softwarearchitektur wird durch die folgende Hardwareabstraktion ergänzt, die mit der Systemarchitektur der IEC 61499 große Ähnlichkeit aufweist:

- **Gerät:** Ein Gerät entspricht einer PNK und stellt ein Hardware-Interface zwischen der physischen- und der cyber-Welt bereit. Das Gerät enthält die genutzte I/O- und Kommunikationshardware, z. B. I/O- und Feldbusbaugruppen bzw. Karten sowie eine oder mehrere Ressourcen.

- **Ressource:** Eine Ressource ist die Abstraktion einer CPU, d. h. einer Rechen- bzw. Ausführungseinheit. Die Ressourcen teilen die Infrastruktur des übergeordneten Geräts, z. B. die I/O-Vorrichtungen.

Sind Anwendungen Typen oder Instanzen?

Es sind beide Sichten sind vertretbar, ob Anwendungen typisierbar sind. Es existieren Anwendungen, die sowohl typisierbar als auch als einzigartige Instanzen, die aus Instanzen weiterer Unterteile zusammengesetzt sind. Die Fragestellung lässt sich auf weitere POU-Untertypen übertragen und ist nur bei Funktionsbausteinen einheitlich von beiden Standards adressiert: Beide definieren eine klare Typ-Instanz Beziehung auf der Ebene der Funktionsbausteine. Die IEC 61499 erwähnt die Typisierung auf einer höheren Kompositionsebene im Kontext einer „Unteranwendung“.

Tabelle 6.1.: Grobe Korrespondenz der Begriffe der drei vorgestellten Architekturen.

Einheitliche Laufzeitarchitektur	IEC 61131-3	IEC 61499
–	„Projekt“	System
Gerät	Konfiguration	Gerät
Ressource	Ressource	Ressource
Anwendung	–	Anwendung
Selbstständige Komponente	Task + Programm	–
FBN	Task + FBN	FBN
Funktionsbaustein	Funktionsbaustein/Funktion	Funktionsbaustein

Die einheitliche Laufzeitarchitektur im Kontext der IEC 61131-3 und der IEC 61499

Es existieren viele Ansätze für die Migration und den Vergleich der Architekturen beider Normen [DDV14, SWH+08, WZSS09]. Damit die vorgeschlagene Architektur besser eingeordnet werden kann, werden die Begriffe der Normen in der Tabelle 6.1 gegenübergestellt. Für den Vergleich zwischen der IEC 61131 und der IEC 61499 wurde der zweite Ansatz aus [SWH+08] gewählt. Dieser basiert auf der syntaktischen Äquivalenz des Begriffes „Ressource“. Im Gegensatz dazu, können auch funktionale Eigenschaften der einzelnen Elemente im Vordergrund stehen, was zu einer anderen Korrespondenz führt (wie z. B. der erste Ansatz in [SWH+08] oder [Yu16]). Die obere Hälfte der Tabelle repräsentiert die Hardware-Hierarchie der jeweiligen Architektur, die untere Hälfte die Beziehungen zwischen den einzelnen Software-Komponenten. Unter dem Begriff „Projekt“ in der mittleren Spalte ist das Projekt des Engineering-Werkzeugs gemeint, das mehrere Konfigurationen beinhalten kann [SWH+08].

6.1.2. Selbstständige Komponenten

In [GE13b] wurde der Begriff einer selbstständigen Komponente (SK) eingeführt, der im Kontext der Softwarearchitektur der IEC 61131-3 eine gesonderte Rolle einnimmt. Um eine bessere Verteilbarkeit und Portabilität der enthaltenen Programmlogik gewährleisten zu können, werden einige der klassischen Aufgaben der Laufzeitumgebung an die SKs delegiert. Dazu gehören z. B. der dedizierte Anschluss an ein nachrichtenorientiertes Kommunikationssystem und die Ablaufkontrolle der enthaltenen POUs.

Die Diskussion in [GE13b] bezog sich hauptsächlich auf die Verbesserungen der Verteilbarkeit und Portabilität der leittechnischen Anwendungen (vgl. Abschnitt 6.1.2) und konzentrierte sich auf die Strukturierung der Anwendungen. Eine verteilte, dienstorientierte Architektur wird als eine der Voraussetzungen für die Umsetzung von Industrie 4.0 genannt [FDT+15]. In diesem Abschnitt werden hingegen die SKs aus der operativen Perspektive dargestellt. Dabei stehen die für die Ausführung relevanten Eigenschaften der Komponente im Fokus der Betrachtung. Diese Eigenschaften umfassen z. B. die Ausführungsdauer und -rate einer Komponente bzw. die Ausführung der enthaltenen POUs.

SKs und FBNs enthalten dedizierte Ausführungsvorschriften für die enthaltene Programmlogik und können die Ausführung dieser selbstständig steuern bzw. überwachen. Ob die Ausführungskontrolle tatsächlich ganzheitlich an die einzelne POUs abgegeben wird

oder durch das Laufzeitsystem mit Mitteln der Introspektion als „Empfehlung“ für den Systemscheduler interpretiert wird, ist eine Designentscheidung und hängt nicht zuletzt davon ab, ob die POU überhaupt die Introspektion zulässt (d. h. ob die Komponente bzw. das FBN zur Laufzeit als ein White-Box Modell vorliegt).

Die Vorteile einer gekapselten Ausführungsstruktur der SKs und FBNs umfassen unter anderem:

- **Koexistenz zwischen unterschiedlichen Ausführungsparadigmen:** In einer Laufzeitumgebung können mehrere Ausführungsparadigmen existieren. Ein Beispiel dafür sind die der zeit- und der ereignisgesteuerten Systeme im Sinne der Kontrolllogik nach IEC 61131-3 und IEC 61499.

- **Verbesserte Portabilität:** Durch gemeinsame Verwaltung der Programmelemente und deren Ausführungsvorschrift in einer architektonischen Einheit wird die Portabilität dieser gesteigert. Dies ist ein Gegensatz zur getrennten Verwaltung dieser in Tasks und POUs nach dem Softwaremodell der IEC 61131-3.

- **Flexibilisierung der Ausführungsstruktur**: Durch beliebige Kombination unterschiedlicher Paradigmen der Ausführungskontrolle kann die Ausführungsstruktur flexibel gestaltet werden.

Der letzte Punkt wurde in [GE13b] dem Themenkomplex „hierarchisches Scheduling“ zugeordnet. Dieses Prinzip wird in Abschnitt 6.1.7 vorgestellt. Ein ähnlicher Ansatz wurde in einer späteren Publikation zur Migration der IEC 61131-3 Anwendungen auf die IEC 61499 vorgeschlagen [DDV14].

Die Komponenten bieten auch in Bezug auf die Anforderung der Migration bestehender Anwendungen einen guten Ansatzpunkt, um diese durch geeignete „Kapselung“ zu flexibilisieren (vgl. die Deployment-Facette einer Anwendung in Abschnitt 6.1.4).

6.1.3. Inter-Komponenten Kommunikation

Die Kommunikation zwischen den selbstständigen Komponenten folgt vorzugsweise dem Prinzip der losen Kopplung, um die Verteilbarkeit der Komponenten auf unterschiedliche Ressourcen bzw. Geräte zu ermöglichen bzw. zu erleichtern. Dieses kann z. B. durch nachrichtenorientierte Kommunikation ermöglicht werden. Spezielle Koppler-Bausteine werden benötigt, um eine Schnittstelle zwischen der nachrichtenorientierten Kommunikation auf der Ebene der Komponenten und der signalorientierten Kommunikation auf der Ebene der Funktionsbausteine zu erhalten. Beispiele für solche Bausteine sind die Kommunikationsbausteine aus IEC 61131-5 sowie die Bausteine für OPC UA, die durch PLCopen und die OPC Foundation spezifiziert wurden [PLC16].

Ist eine komplette Umstellung auf die Nachrichtenorientierung nicht möglich, so können ebenfalls die Koppler-Bausteine verwendet werden, die allerdings bei einer Anpassung der Anwendungsverteilung manuell oder durch ein Engineering-Werkzeug angepasst werden müssen.

Diese zwei Kommunikationsarten sind schematisch in Abbildung 6.2 dargestellt. Auf dem Bild ist die Kommunikation zwischen je zwei SKs innerhalb Anwendung 1 und Anwendung 2 dargestellt. Da Anwendung 1 auf einer Ressource ausgeführt wird, läuft sowohl

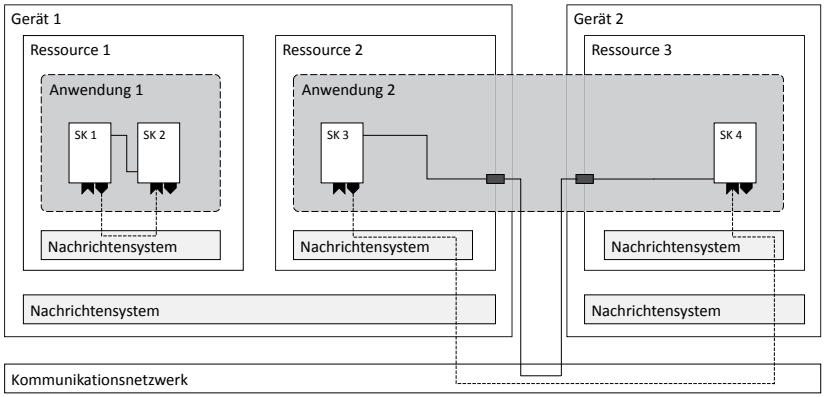


Abbildung 6.2.: Inter-Komponenten Kommunikation innerhalb einer verteilten Anwendung mittels einer Nachrichtenübertragung und signalorientierter Kommunikationsbausteine [GE13b].

die nachrichtenorientierte als auch die signalorientierte Kommunikation ausschließlich innerhalb dieser Ressource ab. Im Falle einer verteilten Anwendung 2, wird die nachrichtenorientierte Kommunikation transparent von Nachrichtensystemen der Ressourcen bzw. der Geräte übernommen. Für die signalorientierte Verbindung müssen dagegen Kommunikationskanäle angelegt werden, die durch schwarze Rechtecke zwischen SK 3 und SK 4 dargestellt sind.

6.1.4. Facetten einer Anwendung bzw. einer selbstständigen Komponente

Die Architektur einer verteilten leittechnischen Anwendung lässt sich aus unterschiedlichen Perspektiven betrachten. Die Gesamtheit der für eine Perspektive relevanten Eigenschaften einer Anwendung wird als eine Facette dieser Anwendung bezeichnet. Die Facetten an sich sind keineswegs eine ungeordnete Eigenschaftenmenge. Sie lassen sich auch feiner strukturieren wie im Folgenden am Beispiel der Scheduling-Facette der Anwendung und der Komponente gezeigt wird.

Es lassen sich drei grundlegende Perspektiven bzw. Facetten einer Anwendung identifizieren, die in Abbildung 6.3 zusammengefasst sind:

Composition-Facette Die Composition-Facette beinhaltet die Unterteilung der Anwendung in einzelne Strukturierungseinheiten (Komponenten). Sie beinhaltet die unterschiedlichen Designmuster, die Untergliederung der Anwendung in weitere Untereinheiten (z. B. in POUs in der Begriffswelt der Softwarearchitektur der IEC 61131-3), sowie die Definition notwendiger Schnittstellen.

Die wichtigsten Aspekte dieser Facette sind die Wiederverwendbarkeit der einzelnen Komponenten, die Wartbarkeit der Anwendung in ihrem Lebenszyklus, die Schnittstellen einzelner Komponenten und deren Design (Black-, Gray- oder White-Box Design).

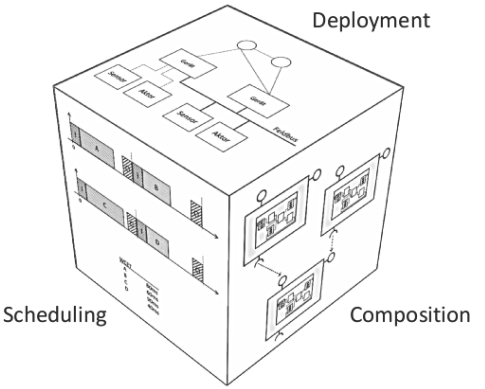


Abbildung 6.3.: Facetten einer Anwendung.

Deployment-Facette Die Deployment-Facette subsumiert Eigenschaften, die für die Zuordnung bzw. Verteilung einzelner Komponenten auf unterschiedliche sogenannte Ressourcen von Bedeutung sind. Ressourcen können entweder unterschiedliche Hardware-Plattformen darstellen oder logischer Natur sein (z. B. unterschiedliche CPU-Kerne einer SPS oder gar Threads auf einer CPU). Aspekte der Verteilung werden vor allem im Kontext der IEC 61499 angesprochen, wurden aber bereits im Umfeld der modellgetriebenen Entwicklung der IEC 61131-3 Software erwähnt [FT11].

Die Sicherstellung der Anwendungssemantik vor und nach der Verteilung beinhaltet viele weitere, teilweise offene, Forschungsthemen. Dazu gehört z. B. die Anforderung an die Kommunikation zwischen den einzelnen Komponenten auf unterschiedlichen Ressourcen. Darüber hinaus könnten einzelne Komponenten nur begrenzt oder gar nicht verteilbar sein.

Die Anforderungen an die Verteilbarkeit einer Komponente auf einen Hardware-Knoten können aus den unterschiedlichsten Domänen entspringen. Es werden beispielhaft die möglichen Anforderungen aus den Domänen der technischen Realisierung, der Redundanz und der funktionalen Sicherheit der Anwendung aufgelistet.

Beispiele für die Anforderungen aus dem Bereich der technischen Realisierung sind:

- Feste Lokalisierung einer Komponente bezüglich eines Geräts wegen der Anbindung der Komponente an ein physisches Prozess bzw. I/O Modul, welches die Verbindung zu der physischen Welt realisiert.
- Anforderungen an die Ressourcen eines Geräts, wie z. B. freier Speicher, verfügbare Rechenkapazität aber auch die Verfügbarkeit der benötigten Software-Komponenten bzw. Abhängigkeiten.
- Anforderungen an die Kommunikation zwischen den verteilten Komponenten implizieren Anforderungen an die Kommunikationsinfrastruktur, wie z. B. Zuverlässigkeit, Latenz, Bandbreite etc.

Beispiele für die Anforderungen aus dem Bereich der Redundanz sind:

- Ist die Redundanz der Anwendung durch mehrere Instanzen sichergestellt, so dürfen diese Instanzen nicht auf einem Hardware-Gerät laufen (dieses gilt gleichermaßen für homogene und diversitäre Redundanz).

- Das vorherige Argument kann beliebig komplex erweitert werden. Zum Beispiel müssen redundante Komponenten auf unterschiedlichen Hardware-Plattformen laufen, die sich nicht in einem Schaltschrank befinden oder gar von einem Stromversorgungsstrang versorgt werden dürfen.

Ein Beispiel für die Anforderungen aus dem Gebiet der funktionalen Sicherheit lautet:

- Anwendungsteile dürfen sich räumlich nicht beliebig von dem gesteuerten physischen System entfernen, damit ein ordnungsgemäßer Funktionsumfang auch im Fehlerfall (z. B. Kommunikationszusammenbruch) sichergestellt ist.

Die Definition der SK aus [GE13b] hat Auswirkungen auf die Verteilbarkeit der Anwendung und stellt diesbezüglich eine Änderung der Verteilungssemantik der IEC 61499 dar. Die Norm sieht eine beliebige Verteilung der leittechnischen Anwendung auf unterschiedliche Ressourcen vor und vereinfacht deren Implementierung durch den eindeutigen Ereignisfluss. Die vorgestellte Architektur bietet dagegen einen Kompromiss zur Verteilung von zeitgesteuerten Anwendungen, die der Philosophie der IEC 61131-3 folgen. Dabei definiert die Einteilung der POUs auf die SKs die „Sollbruchstellen“ der Anwendung in Bezug auf die Verteilung auf unterschiedliche Ressourcen. Die Komponente selbst bietet eine „atomare“ Ausführungsumgebung, die eine unveränderte Semantik der zugeordneten POUs sicherstellt (eine „virtuelle SPS“). Der Programmierer muss nur die Kommunikation zwischen den einzelnen SKs überprüfen, um eine fehlerfreie Verteilbarkeit sicherzustellen. Dabei liegen die Definition der SKs und deren Granularität in seiner Verantwortung. Der triviale Migrationspfad von einer IEC 61131-3 Anwendung ist die komplette Übernahme eines Programms in eine SK. Damit ist eine korrekte Ausführung, jedoch noch keine Verteilbarkeit sichergestellt.

Eine umfassende Untersuchung aller Aspekte der Verteilbarkeit sowie die Synthese einer Problemstellung und der Strategien liegen nicht im Fokus dieser Arbeit. Das entwickelte Modell des Schedulings legt aber einen Grundstein für die technische Realisierung der Komponenten-Migration im laufenden Betrieb (vgl. Abschnitt 6.3).

Scheduling-Facette Im Gegensatz zum Deployment, ist das Scheduling, also die zeitliche Ausführung einer Anwendung und deren Komponenten, das Hauptthema dieser Dissertation. Die Ausführung der Anwendung bzw. ihrer Komponenten muss die Anforderungen der richtigen Reihenfolge der Ausführung, sowie der Rechtzeitigkeit der Ausführung erfüllen. Es ist die Voraussetzung für das Einhalten der Echtzeitanforderungen an die Anwendung. Der Aufbau dieser Facette wird in Abschnitt 6.1.5 detailliert diskutiert.

Die Idee der Segregation der Anwendungseigenschaften in unterschiedliche semiorthogonale Kategorien ist auch in anderen Quellen zu finden. So wird, z. B. in der IEC 61131-3 zwischen den POUs und deren Datenkopplung über Verbindungen bzw. Variablen und den Tasks unterschieden, die die zwei der vorgestellten Aspekte darstellen. Die Verteilung der POUs auf unterschiedliche Ressourcen wurde in IEC 61499 angesprochen. Diese umfasst alle drei der vorgestellten Facetten.

Die drei Facetten finden sich im 4+1 Sichten-Architekturmodell für Softwaresysteme [Kru95] wieder. Die Composition-Facette entspricht dem "Logical view", die Deployment-Facette dem "Physical view" und die Scheduling-Facette dem "Process view". Die im 4+1 Modell zusätzlich enthaltene Entwicklungssicht beschreibt das System aus der Perspektive der Softwareentwicklung und steht nicht im Fokus dieser Arbeit.

Die Facetten werden als semi-orthogonal bezeichnet, da die Beeinflussung einzelner Aspekte durch andere nicht vollkommen ausgeschlossen werden kann. So hat eine Änderung der Hardware-Plattform im Zuge eines Deployments direkte Auswirkungen auf die plattformabhängigen der Scheduling-Facette, wie z. B. der WCET. Gleichzeitig implizieren die Anforderungen der rechtzeitigen Ausführung Anforderungen an die Kommunikationsinfrastruktur und schränken die in Frage kommenden Möglichkeiten für die Auswahl der Deployment-Ressourcen ein.

Darüber hinaus sind weitere Aspekte einer Anwendung identifizierbar, die nicht exklusiv einer, sondern mehreren Perspektiven zugeordnet werden können. Ein Beispiel dafür sind die softwaretechnischen Abhängigkeiten einer Komponente, die von der funktionalen Unterteilung und der Deployment-Perspektive einer Anwendung abhängen. Werden z. B. bei einer Anwendung Black-Box Komponenten oder Funktionsbausteine benutzt, so muss sichergestellt werden, dass auch nach der Verteilung die gleiche Version der Komponenten bzw. deren Abhängigkeit verfügbar ist. Im anderen Fall wäre durch Versionskonflikte das Gesamtverhalten der Anwendung, vor und nach dem Deployment-Vorgang, möglicherweise nicht identisch.

6.1.5. Scheduling-Facette einer selbstständigen Komponente (Task-Eigenschaften)

Die Task-Eigenschaften der selbstständigen Komponenten sind für den Scheduler und somit für die Ausführung der SK im Allgemeinen relevant. Diese leiten sich aus der Literatur zu Scheduling und zu den Echtzeitbetriebssystemen ab (vgl. Abschnitt 2.1.2). In diesem Abschnitt werden die Task-Eigenschaften von SKs als die Scheduling-Facette der Komponenten bezeichnet. Zur Harmonisierung mit den Begriffen der Scheduling-Theorie wird synonym dazu auch der Begriff „Task" verwendet. Trotz namentlicher und inhaltlicher Ähnlichkeit, ist dieser Begriff nicht mit den Tasks aus dem Software-Modell der IEC 61131-3 zu verwechseln.

In dieser Arbeit wird grundsätzlich, wie auch im Kontext des Schedulings der zeitgesteuerten Echtzeitsysteme, zwischen den zyklisch- und den sporadisch-ausgeführten Tasks unterschieden. Während die Ersten dem Gedanken der zyklischen Aufgaben und auch dem Hauptverwendungsmuster der SPS folgen, werden die Zweiten „nach Bedarf", d. h. beim Eintreten bestimmter Bedingungen ausgeführt. Zu jedem Zeitpunkt kann höchstens eine Instanz eines sporadischen Tasks ausgeführt werden bzw. auf die Ausführung warten. Zu einem sporadischen Task kann eine Separationsdauer angegeben werden. Das ist die minimale Zeitdauer zwischen dem Auftreten zwei aufeinanderfolgender Taskinstanzen.

In der Literatur [Mal09] wird darüber hinaus noch eine dritte Kategorie diskutiert – die aperiodisch-ausgeführten Komponenten. Der Unterschied zwischen sporadischer und aperiodischer Ausführung ist die fehlende Separation zwischen den Taskinstanzen. Als Konsequenz können unbeschränkt viele Instanzen gleichzeitig auf die Ausführung warten, was ein vorhersagbares Verhalten des Systems verhindert. Aus diesem Grund werden ape-

riodische Tasks nur im Kontext der weichen Echtzeit betrachtet. Es wird zunächst davon ausgegangen, dass alle Komponenten zyklischer Natur sind. Die zusätzliche Ausführung sporadischer Tasks wird in Abschnitt 6.3.3 adressiert.

Die Scheduling-Eigenschaften einer Anwendung leiten sich aus den End-to-End Anforderungen an diese ab [GE15]. Für die Fragestellungen dieser Arbeit sind jedoch primär die Scheduling-Eigenschaften der SKs von Bedeutung. Da die Komponenten einer Anwendung hierarchisch untergeordnet sind, fließen die Scheduling-Anforderungen an die Anwendung auch in die Scheduling-Facetten der einzelnen SKs ein. Allerdings erfolgt bei diesem Prozess eine Spezialisierung der Eigenschaften, die in diesem Abschnitt beschrieben wird.

Die Eigenschaften der Scheduling-Facette einer SK, die für das weitere Vorgehen, z. B. für das Scheduling benötigt werden, können in drei Kategorien unterteilt werden:

- **Anwendungsbezogene Eigenschaften:** Zu den anwendungsbezogenen Eigenschaften gehören die initialen Eigenschaften einer SK, wie z. B. Priorität, Zykluszeit als absolute Zeitdauer, Phase, Puffer-Verhalten (ob die Eingabe bzw. die Ausgabe gepuffert werden oder direkt über I/O wirksam werden soll).

  Zusätzlich können Angaben zur Variation der oben genannten Einschränkungen hinterlegt werden, wie z. B. die Veränderung der Phase (ob und in welchem Umfang die Phase in Bezug auf die Zykluszeit geändert werden kann).

- **Eigenschaften vor dem Deployment (bezogen auf eine Hardware-Plattformklasse der Ressource):** Diese Eigenschaften werden speziell für eine Plattformklasse ermittelt. Sie enthalten beispielsweise die WCET dargestellt als Anzahl der CPU-Zyklen und weitere Ressourcenanforderungen an die Plattform, wie z. B. den Arbeitsspeicherbedarf, der von der eingesetzten CPU-Architektur der Ressource abhängen kann. Zusätzlich können die Anforderungen an die Software-Infrastruktur einer Ressource sowie Qualitäts- und Vertrauensmerkmale der Eigenschaften enthalten sein.

- **Eigenschaften nach dem Deployment:** Nach dem Deployment stehen die Scheduling-Eigenschaften endgültig fest. So ist es möglich die WCET als absolute Zeitdauer anzugeben, da die CPU Frequenz bekannt ist. Darüber hinaus ist die Angabe der an den Grundzyklus der Ressource angepassten Zykluszeit möglich.

6.1.6. Kontrollfluss innerhalb der selbstständigen Komponenten und der Funktionsbausteinnetzwerke

Die flexible Ausführungsstruktur der SKs und FBNs eröffnet Möglichkeiten zu deren Nutzung als Grundeinheit für das vorgestellte Modell der RA-Algorithmen und deren Ausführungssemantik. In [GE13b] wurden die drei Grundtypen des Kontrollflusses in einer Komponente bzw. einem FBN unterschieden:

- **Aufrufbasierter Kontrollfluss:** Funktionsaufrufe sind aus den höheren Programmiersprachen bekannt und auch im Kontext der IEC 61131-3 Sprachen verfügbar (als Teil des strukturierten Texts). Bei einem Aufruf pausiert die Ausführung der aufrufenden Einheit und der Kontext wird auf einem Stapel gespeichert, solange die aufgerufene Funktion aktiv ist. Rekursive Aufrufe sind im Kontext der IEC 61131-3 untersagt [JT09].

Der Funktionsaufruf schafft eine Abhängigkeit zwischen den beiden beteiligten Komponenten. Diese Abhängigkeit bezieht sich auf den Datenfluss, die Ausführungszeit (die Ausführungszeit der Funktion trägt zu der Dauer der Ausführung des Aufrufers bei) und die Softwareverteilung (die Komponente bzw. die Bibliothek, deren Funktionen aufgerufen werden, muss verfügbar sein bzw. eine kompatible Version haben). Bezüglich des Datenflusses steht der Funktionsaufruf in Konkurrenz mit den Datenverbindungen aus den graphischen Programmiersprachen. Im Gegensatz zu diesen ist er implizit, d. h. nicht mithilfe der Baustein-Introspektion zu entdecken.

Es wird davon ausgegangen, dass Funktionsaufrufe nur innerhalb der Bausteinimplementierung sicher verwendbar sind und nicht für die Kopplung unterschiedlicher Bausteine/Komponente verwendet werden. Eine Ausnahme bilden die Aufrufe aus Schrittketten, die diskrete Prozeduren beschreiben, heraus.

- **Tasklisten- oder tabellenbasierter Kontrollfluss:** Die POUs werden nach einer festen Reihenfolge in einer Liste aufgerufen. Im Gegensatz zu der aufrufbasierten Steuerung, kehrt der Kontrollfluss nach dem Abarbeiten einer POU zurück zu dem Scheduler. Diese Unterscheidung ist nicht ganz präzise, da POUs Funktionsaufrufe verwenden können. Für diese Unterscheidung ist somit die Strukturierung des Programms in eine Taskliste mit gekoppelten, fein granulierten POUs entscheidend.

  Tasklistenbasierte Ausführungskontrolle ist der Ansatz aus der IEC 61131-3, in der die Tasks ein dedizierter Bestandteil des Softwaremodells sind. Eine Taskliste ist einfach implementierbar und kann entweder offline oder auch online berechnet bzw. modifiziert werden. Ein weiterer Vorteil der Tasklisten ist die einfache Berechnung der Ausführungsdauer. In diesem Fall werden einfach die Ausführungszeiten der enthaltenen Elemente aufaddiert. Ein Nachteil dieses Ansatzes ist die Rigidität der Listen. Dadurch finden diese vor allem in zeitgesteuerten Systemen ihren Einsatz [TFB13].

  Im Gegensatz zur Architektur der IEC 61131-3, gehören die Listen zu der SK und dürfen nur auf die in der Komponente enthaltenen Elemente verweisen. Diese Forderung trägt zu der Portabilität der SK und der Anwendung bei, da die Komponente nur zusammen mit der Taskliste migriert werden kann.

- **Ereignisgetriggerter Kontrollfluss:** Die Ausführung einer Einheit kann Ereignisse aussenden, die die Ausführung weiterer Einheiten anstoßen. Selbst wenn die in Frage kommenden Empfänger der Ereignisse bekannt sind (wie im Fall der IEC 61499-basierten Bausteinsemantik), kann das Aussenden der Ereignisse und somit die Ausführung einer Funktion im Empfänger von der Logik des Senders abhängen. Diese Flexibilisierung wird durch die komplexere Implementierung und Analyse des Ereignisflusses erkauft. So wird das Problem der Laufzeitabschätzung der Komponente nichttrivial und fordert den Einsatz komplexer Analysealgorithmen [Zoi08].

Neben diesen Grundtypen des Kontrollflusses sind auch weitere Kontrollflusstypen vorstellbar. So wird z. B. in Abschnitt 6.2 ein neues RA-Verfahren zur Steuerung des Kontrollflusses vorgestellt. Zunächst muss aber die Fragestellung der Koexistenz und der Kombination unterschiedlicher Modelle beantwortet werden. Diese Frage wird durch das Konzept des hierarchischen Scheduling adressiert, das in Abschnitt 6.1.7 vorgestellt wird.

6.1.7. Hierarchisches Scheduling

Die Idee des hierarchischen Schedulings basiert auf der Annahme von hierarchisch angeordneten Tasks, denen die vorhandenen POUs der Anwendung zugeordnet sind. Durch die Forderung einer festen Kopplung der Ausführungsvorschrift an die selbstständige Komponente (Abschnitt 6.1.2, Punkt 2), wird die Schachtelung der Tasks durch die Hierarchie der POUs induziert. Die Reihenfolge der Komponentenausführung muss (typischerweise während der Engineering-Phase) von dem Programmierer vorgegeben werden. Diese Aktivität ist bereits ein Grundbestandteil der Softwareentwicklung nach der IEC 61131-3 (durch die Zuordnung der POUs zu den Tasks) und der IEC 61499 (durch das Anlegen der Ereignisverbindungen) und ist somit für vorhandene Programme schon erfolgt.

Die Ausführung innerhalb eines Zyklus der Laufzeitumgebung beginnt mit der Abarbeitung eines Wurzel-Tasks, der Verweise auf *strukturierte* (white-box) Tasks oder *unstrukturierte* (black-box) Tasks enthält. Zum Beispiel enthalten die strukturierten Tasks eigene Tasklisten, die die Ablaufkontrolle innerhalb des Tasks reglementieren. Bei den unstrukturierten Tasks wird die Ausführungsvorschrift durch einen komponentenspezifischen Scheduler realisiert, welcher z. B. durch eine Routine in ST vorgegeben ist. Wenn alle Tasks „aufgelöst“ werden, lässt sich der Wurzel-Task als ein Baum darstellen, dessen Blätter nur Verweise auf unstrukturierte und die inneren Knoten nur Verweise auf strukturierte Tasks sind. Die Blätter dieses imaginären Baumes werden dann vom Scheduler in einem Tiefensuche-Durchlauf besucht. Der zu dem Blatt zugeordnete Task bekommt bei dem Besuch die Kontrolle über die ausführende Ressource.

Die inneren Knoten des Baums werden nicht im Voraus, sondern erst zur Laufzeit aufgelöst. In diesem Fall wird von der Übergabe der Ausführungskontrolle an den strukturierten oder den unstrukturierten Task gesprochen. Der Task übernimmt somit bis zu seiner Terminierung die Rolle des Schedulers für die beinhaltete POUs bzw. Tasks.

In den meisten Fällen wird eine solche Ausführung die Abarbeitung einer internen Taskliste beinhalten. Dieses Standardverhalten kann durch einen rekursiven Aufruf des systemweiten Tasklisten-Schedulers veranlasst werden (der z. B. im Falle einer objektorientierten Implementierung der Komponenten geerbt werden kann), damit entfällt die Notwendigkeit der Umsetzung der Scheduling-Schnittstelle für jede Komponente.

Allerdings ist es nicht die einzige denkbare Möglichkeit für das Scheduling. Vielmehr kann jedes der in Abschnitt 6.1.6 zusammengetragenen Kontrollflussprinzipien eingesetzt werden, um die interne Logik der POU abzuarbeiten. Zum einen können Optimierungen bzw. Anpassungen der tasklistenbasierten Ablaufsteuerung in das Framework des hierarchischen Schedulings eingebettet werden. Es kann eine Reihe von einfachen lokalen Optimierungen angewendet werden, um die Ausführungszeit der Komponente zu verkürzen. Zum Beispiel kann bei einem Funktionsbaustein ohne internen Speicher (einer „Funktion“ in der Terminologie der IEC 61131-3) auf die Ausführung der Logik verzichtet werden, wenn es seit dem letzten Zyklus keine Änderung der Eingangsports gegeben hat. Die Ausführungsdauer des Blocks beträgt in diesem Fall nur die Dauer der Überprüfung der Änderung, der an den Eingangsports anlegenden Signale.

Zum anderen eröffnet die Möglichkeit der ereignisgesteuerten Abarbeitung der Unterkomponente einen transparenten Weg für die Einbettung von IEC 61499-basierten Kontrolllogik in ein zyklisches Laufzeitsystem. Somit wird die zeitgesteuerte Aktivierung der Komponente mit der ereignisgesteuerten Ausführung der internen Logik kombiniert. In diesem Fall ruft die Komponente nicht den Standardscheduler für Tasklisten auf, sondern

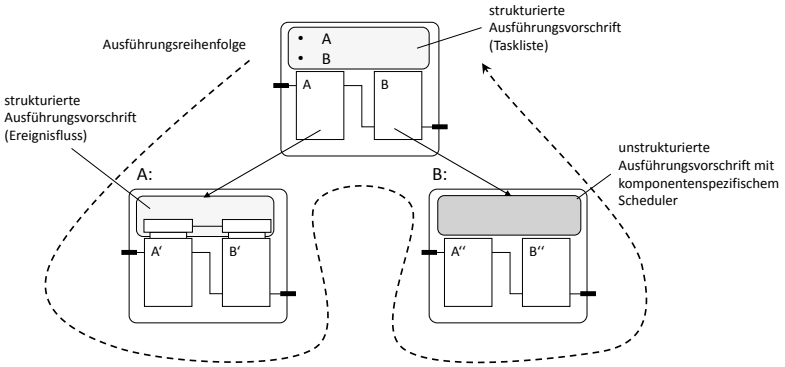


Abbildung 6.4.: Schematische Darstellung des Prinzips des hierarchischen Schedulings.

nutzt die interne bzw. die systemweite Implementierung eines ereignisgesteuerten Schedulers für interne Kontrolllogik. Die Implementierung und die Verwaltung der für den Ereignisfluss relevanten Unterstrukturen wird somit an den „inneren“ komponentenspezifischen Scheduler übertragen. Die einzelne Anforderung an diesen Scheduler ist zunächst die Einhaltung der WCET, die der ausgeführten Komponente zugeordnet wurde, damit der übergeordnete Scheduler die Sicherstellung der Deadlines gewährleisten kann.

Die hierarchische Ausführung ist beispielhaft in Abbildung 6.4 dargestellt. Die Wurzel-POU enthält zwei weitere POUs A und B, zwischen denen eine Signalverbindung besteht. Die POUs beinhalten die Ausführungsvorschrift, die im oberen Teil der POUs in grau dargestellt wird. Man kann bei der Vorschrift zwischen strukturierten und unstrukturierten Tasks unterscheiden. Die Wurzel-POU und POU A sind strukturiert, während B unstrukturiert ist. Bei den strukturierten Tasks sind weiterhin zwei unterschiedliche Typen sichtbar. Bei der Wurzel-POU ist es eine Taskliste, die von oben nach unten abgearbeitet wird. Bei A ist es ein Ereignisfluss, der die Ausführung der untergeordneten Einheiten A' und B' regelt. Die Ausführungsreihenfolge der POUs ist durch eine gestrichelte Linie angedeutet. Diese entspricht einer Tiefensuche im Baum, der in der Abbildung angedeutet ist.

Im Rahmen dieser Arbeit wird eine weitere Klasse der strukturierten RA-Tasks eingeführt, die die temporalen Eigenschaften der Komponenten (vor allem ihre Ausführungsdauer) kontextabhängig (z. B. bezogen auf die aktuelle Systemauslastung) anpassen kann. Dieser Scheduler und die Mittel für die Beschreibung der Variation des Verhaltens werden in Abschnitt 6.2 vorgestellt.

Welche Scheduler-Klassen von dem Laufzeitsystem bereitgestellt werden und welche direkt in eine Komponente eingebettet werden, ist eine Designentscheidung für das Laufzeitsystem. Die optimale Antwort auf diese Frage hängt davon ab, welche Scheduling-Prinzipien von den eingesetzten Komponenten genutzt werden und welche von dem Laufzeitsystem zugelassen sind. Eine transparente Möglichkeit der Einbettung weiterer komponentenspezifischer Scheduler schränkt die Flexibilität des Gesamtsystems nicht ein.

Die Grundannahme im Rahmen dieser Arbeit ist die Existenz einer Wurzel-Taskliste, die von einem Systemscheduler verarbeitet wird. Somit ist die Koexistenz der zyklischen und der ereignisgesteuerten Ausführungskontrolle nur in einem zeitgesteuerten System si-

chergestellt. Nach der Klassifikation der Ansätze für die Koexistenz beider Philosophien, die in [ZSSB09] vorgestellt wurde, fällt der Ansatz des hierarchischen Schedulings unter die Gruppe der auf der IEC 61131-3 basierenden Lösungen. Die Prinzipien des hierarchischen Schedulings haben dagegen auch im ereignisgesteuerten Kontext Bestand. Die einzige Forderung ist die gemeinsame Verwaltung der Logik und der Ausführungsinformation. Somit können z. B. SKs mit einer eingebetteten Taskliste in ein ereignisgesteuertes System eingebracht werden, solange die Abarbeitung der Liste durch ein Ereignis ausgelöst wird.

6.2. Meta-Modell für ressourcenadaptive Komponenten

In der Beschreibung des Prinzips des hierarchischen Schedulings in Abschnitt 6.1.7 wurde auf die unterschiedlichen Ausführungsprinzipien innerhalb der SKs eingegangen. Die Kernidee dieses Prinzips ist die Kapselung der Scheduling-Information durch die Komponente. In diesem Abschnitt wird eine Ausführungsart vorgestellt, die sich von der tasklisten- und der ereignisgesteuerten Ausführung unterscheidet. Als Grundlage für die Ausführung der gekapselten Logik der Komponenten wird eine explizit formulierte Prozedur genutzt, die zur Laufzeit evaluierbar ist.

6.2.1. Annahmen und Begriffsdefinitionen für das Scheduling

In diesem Abschnitt werden die Grundbegriffe und Rahmenbedingungen definiert, die für das verwendete Systemmodell gelten. Diese sind sowohl für das Meta-Modell der ressourcenadaptiven Komponenten (Abschnitt 6.2), als auch für den ressourcenadaptiven Komponentenscheduler (Abschnitt 6.3) gültig. Folgende Eigenschaften werden für das Laufzeitsystem angenommen:

Diskrete zyklische Zeit Eine der grundlegenden Entscheidungen für das Modelldesign ist die Wahl des zugrundeliegenden Zeitmodells. Grundsätzlich existieren zwei fundamentale Philosophien der Zeitmodellierung: das dichte (engl. dense) und das diskrete (engl. discrete) Zeitmodell. Beim ersten Ansatz wird die Zeit als eine dichte Menge modelliert, d. h. es existiert ein Zeitpunkt zwischen zwei beliebigen Zeitpunkten. Beim zweiten Ansatz wird die Zeitmenge als Menge diskreter Zeitpunkte modelliert, die eine feste zeitliche Auflösung definieren.

Viele technische Systeme und Prozesse sind zyklisch. Ein zyklisches Verhalten ist durch die immer wiederkehrenden Muster im Systemverhalten gekennzeichnet. In diesem Fall kann zur Vereinfachung der Modellierung ein diskretes zyklisches Zeitmodell gewählt werden [Kop11b]. Die Zeit wird in diesem Fall in gleichlange Perioden unterteilt. Ein Zeitpunkt kann durch die Angabe der Periode und der Phase, d. h. einer relativen Angabe zum Periodenanfang, eindeutig referenziert werden.

Das definierte Meta-Modell nutzt die Annahme einer diskreten zyklischen Zeit. Die Zyklusdauer wird dabei durch die Abtastrate des physischen Systems, auf die Dauer des Grundzyklus (vgl. Abschnitt 2.2.3), normiert. Diese Abtastrate ist an viele Nebenbedingungen wie die eingesetzte Hardware, Kommunikationsinfrastruktur etc., aber auch an die Zeitkonstanten des zu automatisierenden physischen Systems gekoppelt. Die Grundbeziehung zwischen der Abtastrate und der Signalfrequenz wird durch das Abtasttheorem hergestellt, was eine mindestens doppelte Abtastfrequenz vorschreibt. In der Praxis

wird normalerweise eine noch etwas höhere, z. B. eine sechs- bis zehnfache Abtastfrequenz [Abe06], gewählt.

Stabiler Grundzyklus während der Laufzeit Jede Ressource (vgl. Abschnitt 6.1.1) wird in einem Grundzyklus betrieben, der während deren Laufzeit nicht verändert werden kann. Die Dauer des Grundzyklus ist hauptsächlich von den Zeitkonstanten des physischen Prozesses abhängig, mit dem die Ressource über die I/O-Hardware und die Sensoren bzw. Aktoren interagiert. Da davon ausgegangen wird, dass die Eigenschaften und insbesondere die Zeitkonstanten des kontrollierten physischen Systems stabil sind, wird die Dauer des Grundzyklus ebenfalls als stabil angenommen.

Zeitgesteuertes Echtzeitsystem Es wird das zeitgesteuerte Modell eines Echtzeitsystems verwendet. Somit hängen die Aktivitäten des Systems nur vom Fortschritt der Zeit ab. Die Vor- und Nachteile dieses Ansatzes und der Alternativen wurden eingehend im Abschnitt 2.1.2 diskutiert. Das gewählte Aktivierungsparadigma erlaubt das Scheduling mittels einer Taskliste.

Stabilität des Prozessabbilds während einer Periode Es wird angenommen, dass das Prozessabbild, d. h. die Werte der über die I/O Komponenten gelesenen Sensoren, sich innerhalb einer Periode nicht verändern. Die gleiche Annahme gilt für die geschriebenen Werte der Outputs der I/O Komponenten, die die Signale an die Aktoren übertragen. Dieses Verhalten entspricht der Softwarearchitektur der IEC 61131-3 und somit den meisten Implementierungen der SPSen. Das Prozessabbild wird in jedem Zyklus synchronisiert (vgl. Abschnitt 2.2.3). Die Werte der I/O-Eingänge werden dabei am Anfang eines Zyklus in einen Zwischenpuffer eingelesen und die Ausgänge am Zyklusende vom einen weiteren Puffer auf die I/O-Ausgänge übertragen und somit in der physischen Welt wirksam.

Die Vorteile dieses Verhaltens umfassen eine einfache Synchronisierung der Zugriffe einzelner Elemente der Kontrolllogik auf die Daten des Prozessabbilds. Es muss nicht auf die Verfügbarkeit eines Eingangswertes gewartet werden. Auch die Konsistenz der Daten ist für alle Zugriffe in einem Zyklus sichergestellt. Bei den Ausgangswerten wird jeweils die letzte Änderung innerhalb eines Zyklus wirksam.

Der Nachteil der zyklischen Abtastung ist die vergleichsweise lange Reaktionszeit des Systems auf die Stimuli aus der physischen Welt. Wie bereits in Abschnitt 2.2.3 bemerkt wurde, kann diese im schlimmsten Fall das Zweifache der Zykluszeit betragen.

Uniprozessor-System Das Modell ist an die Architektur eines SPS-Systems angelehnt, das typischerweise eine Ressource enthält [IEC11]. Im Falle der Verfügbarkeit mehrerer Ressourcen, kann die Kommunikation zwischen diesen über einen gemeinsamen Speicherbereich erfolgen, der ähnlich wie das Prozessabbild für die Zyklusdauer konstant gehalten wird (die Zykluszeit der Komponenten wird in diesem Fall auf die Dauer des systemweiten Grundzyklus normiert). Eine weitere Möglichkeit ist der Einsatz eines komplexeren Systems zur Interprozesskommunikation, wie z. B. eines Nachrichtensystems. Wie bereits im letzten Paragraphen besprochen, sind diese Maßnahmen notwendig, um die Datenkonsistenz ohne Zuhilfenahme aufwändiger Synchronisationsmechanismen sicherzustellen.

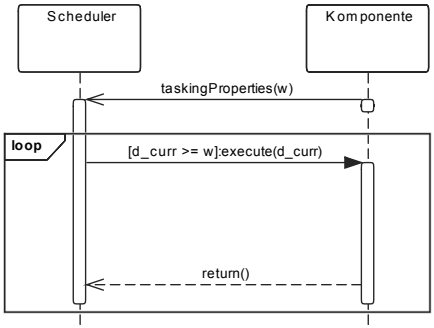


Abbildung 6.5.: Informationsaustausch und Aufrufsteuerung zwischen Komponenten und Komponentenscheduler.

Nichtunterbrechbarkeit der Tasks Es gilt die Annahme, dass die Tasks während der Ausführung nicht unterbrochen werden können. Die Vor- und Nachteile dieses Ansatzes wurden in Abschnitt 2.1.2 diskutiert. Die Vorteile des kooperativen Schedulings, wie minimaler Jitter und eine bessere Vorhersagbarkeit des Systemverhaltens, sind für viele Anwendungen in der industriellen Automatisierung überwiegend.

6.2.2. Ressourcenzuteilung durch den Komponentenscheduler zur Laufzeit

Da in diesem Abschnitt nur die interne Ausführung der selbstständigen Komponente Gegenstand der Betrachtung ist, kann der Datenaustausch zwischen der Komponente und dem übergeordneten Scheduler (dem Komponentenscheduler) auf das folgende einfache Interface reduziert werden:

- WCET der Komponente auf der aktuellen Ressource w (Ausgangsparameter, konstant im Betrachtungszeitraum, *nicht* zur Laufzeit ausgetauscht) und
- die vom Komponentenscheduler zugeteilte Zeit für die aktuelle Ausführung $\Delta^{curr} \geq w$ (Eingangsparameter, konstant während einer Periode, *explizit* zur Laufzeit ausgetauscht).

Der erste Parameter w beschreibt im Kontext der RA-Algorithmen die *maximale* Zeit, die zu einer *minimalen* konsistenten Ausführung der Komponente benötigt wird. Die Zusammenhänge zwischen den beiden Parametern sind in Abbildung 6.5 als Sequenzdiagramm dargestellt.

Die Bedingung $\Delta^{curr} \geq w$ garantiert die Sicherstellung der Echtzeitanforderungen auf der Komponentenebene und muss von dem Komponentenscheduler bei der Zuteilung der Zeiten eingehalten werden. Für die Zuteilung der Rechenzeit existieren zwei Fälle:

- $\Delta^{curr} = w$: In diesem Fall kann die Komponente nur die intern akkumulierte Slackzeit verwerten, die z. B. als Folge der zu pessimistischen Abschätzung der WCET oder eines Ausführungspfades, der kürzer als der WCET-Fall ist, verfügbar ist.

Dadurch ist eine effiziente Ressourcennutzung auch bei der Nutzung „starrer“ Scheduler möglich (d. h. solcher Scheduler, die einer Komponente nur die „minimale angefragte“ Zeit w zuweisen).

- $\Delta^{curr} > w$: In diesem Fall teilt der Komponentenscheduler der Komponente mehr Zeit als deren WCET zu. Somit kann die Komponente auch die Slackzeit verwerten, die im aktuellen Zyklus während der Ausführung anderer Komponenten bereits entstanden ist oder entstehen wird.

6.2.3. In-cycle Sequential State Chart (ISSC)

Nach den Prinzipien des hierarchischen Schedulings (vgl. Abschnitt 6.1.7) soll die ausgeführte Kontrolllogik (die primär aus Funktionsbausteinen besteht) und die Vorschrift zu deren Ausführung bzw. der Scheduler innerhalb einer Komponente zwar logisch getrennt sein, aber gemeinsam in einer POU verwaltet werden. Es wurden zwei Möglichkeiten der Darstellung bzw. Speicherung der Ausführungsvorschrift vorgestellt und diskutiert: die Darstellung als Taskliste nach IEC 61131-3 sowie die Darstellung als Ereignisflussnetz nach IEC 61499 (Ereignisfluss zwischen den „Köpfen“ der Funktionsbausteine).

In diesem Abschnitt wird eine Prozedurbeschreibungssprache vorgestellt, die für die Beschreibung der ressourcenadaptiven Ausführungskontrolle innerhalb der FBDs und SKs eingesetzt werden soll.

Durch die Zielsetzung der Arbeit, die Ressourcen und insbesondere die Slackzeit innerhalb eines Zyklus effizienter zu nutzen, muss das Verhalten einer Komponente auch innerhalb des Zyklus variierbar sein. Es ist somit offensichtlich, dass das Mittel für die Beschreibung dieser Verhaltensvariation die in-cycle Semantik und Zeitsensitivität unterstützen muss. Darüber hinaus muss die in-cycle Semantik das Durchlaufen mehrerer Zustände bzw. Schleifen innerhalb der Prozedur ermöglichen. Dieses ist bereits für die Umsetzung eines optionalen Zustands erforderlich, die z. B. für die Umsetzung des „Imprecise Computation Models“ (vgl. Abschnitt 5.1.5) vorausgesetzt wird.

Von den existierenden Prozedurbeschreibungssprachen wird die in-cycle Semantik von PLC-Statecharts [WVH11, Wit12] sowie von einer möglichen Semantik für PLC unterstützt (vgl. Abschnitte 3.4.1 bzw. 3.4.2). Diese Semantik erlaubt die wiederholte Ausführung der Aktionen eines Zustands innerhalb eines Zyklus und abstrahiert somit die eventuelle zyklische Ausführung des Laufzeitsystems (vgl. Abschnitt 3.4.2). In [WVH11] wurde die in-cycle Semantik der PLC-Statecharts am Beispiel einer mehrfachen Ausführung der Aktionen eines Zustands, die von einer Zählervariable abhing, demonstriert. Die in-cycle PLC-Statcharts bieten allerdings weder die Möglichkeit mehrere Zustände innerhalb eines Zyklus zu besuchen, noch die Möglichkeit das Verhalten des Charts von der Ausführungszeit abhängig zu machen.

Im Folgenden wird eine Prozedurbeschreibungssprache eingeführt, deren Semantik den benötigten Anforderungen der Zeitsensitivität und der in-cycle Semantik genügt. Es ist sinnvoll die Syntax und die Ausdrucksstärke einer bestehenden Prozedurbeschreibungssprache zu übernehmen und nur die nötigen Aspekte der Semantik anzupassen. Diese Vorgehensweise hat sich im Falle der zyklischen Semantik für Statecharts am Beispiel von PLC-Statecharts und SSCs bewährt. Für den Zweck der Beschreibung von RA-Algorithmen wurde daher die Syntax der SSCs ausgewählt, die im Folgenden mit einer in-cycle Semantik ausgestattet wird. Die aus dieser Kombination entstandene Sprache wird als In-cycle

Sequential State Chart (ISSC) bezeichnet. Um Mehrdeutigkeiten zu vermeiden, wird im Folgenden zwischen den ISSCs und den klassischen SSCs differenziert, die auch im Fall einer Mehrdeutigkeit als multi-cycle Sequential State Charts bezeichnet werden.

Der Einsatz von Prozedurbeschreibungssprachen in der Leittechnik ist an die folgenden Anforderungen gebunden, die sich teilweise mit den Anforderungen an die flexiblen Algorithmen in der Leittechnik im Allgemeinen (vgl. Kapitel 4) decken. Die Liste lehnt sich an die Vorarbeiten [YQGE12, YGE13] in Co-Autorschaft sowie Anforderungsanalysen der DSLs [KPKP06, Gro09] an:

- Die Sprache soll eine Kombination aus einer funktionalen Beschreibung und Programmierung durch graphische Repräsentation sein. Diese Repräsentation ist so präzise, dass sie direkt operativ einsetzbar ist, d. h. sie kann interpretiert oder für die Synthese des ausführbaren Codes verwendet werden.

- Die Sprache soll den domänenspezifischen Konzepten der Leittechnik folgen, den Stand der Technik und die historische Entwicklung dieser Domäne berücksichtigen.

- Die Eindeutigkeit der Semantik muss durch Formalisierung bzw. eine Abbildung auf ein formales Modell gewährleistet werden.

- Die Balance zwischen der Ausdrucksstärke und der Komplexität bzw. der Zweckmäßigkeit und der Kompaktheit muss so gewählt werden, dass die vorhandenen syntaktischen Mittel gerade für die Beschreibung der benötigten Prozeduren ausreichen. Jedes zusätzliche Element erhöht die Komplexität der Semantik und führt zu potentiellen Problemen bei der Implementierung [Yu16].

Die Einhaltung der ersten zwei Anforderungen ist durch die Übernahme der Beschreibungselemente und der Syntax der SSCs erfüllt. Die Eindeutigkeit der Semantik ist durch die Abbildung auf TA bzw. auf die TA-Netzwerke von UPPAAL gewährleistet. Die letzte Anforderung wird durch die teilweise Reduktion der Beschreibungsstärke von SSC erreicht. Die Sprache SSC verzichtet ihrerseits bereits auf viele Elemente von SFC und UML-Statecharts. Diese Einschränkungen sind nicht endgültig – bei Bedarf können die Syntax und die Semantik angereichert werden.

Es können weitere Vereinfachungen der syntaktischen Elemente der ISSCs im Vergleich zu SSC vorgenommen werden, ohne deren Eignung für die Definition der Ablaufsteuerung von RA-Algorithmen negativ zu beeinflussen bzw. sich signifikant auf die Größe des Charts oder dessen Komplexität auszuwirken:

- **Vereinfachter Ausführungsrahmen:** Da die ISSCs vielmehr mit Tasklisten als mit einfachen Bausteinen vergleichbar sind, werden die Anforderungen an den Ausführungsrahmen eines Charts bzw. an seine externen Schnittstellen abgeschwächt. Die ISSCs operieren im Rahmen der gesteuerten POU und verwenden somit dessen Ausführungsrahmen als Schnittstelle.

- **Aktionen nur in der entry-Phase:** In SSC können die Aktionen drei Phasen eines Zustands – „entry“, „do“ und „exit“ zugeordnet werden. Die Ausführung einer Aktion in der do-Phase hat wegen der fehlenden multi-cycle Semantik der ISSCs keine Bedeutung mehr – das Statechart kann nicht mehr in einem Zustand über mehrere Zyklen verharren. Die Ausdrucksstärke von exit-Anweisungen ist wegen der

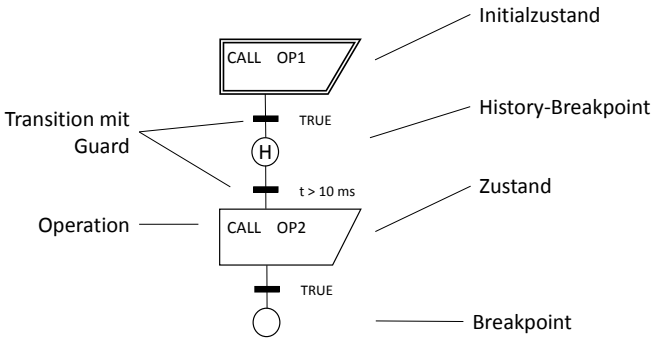


Abbildung 6.6.: Graphische Darstellung eines ISSCs.

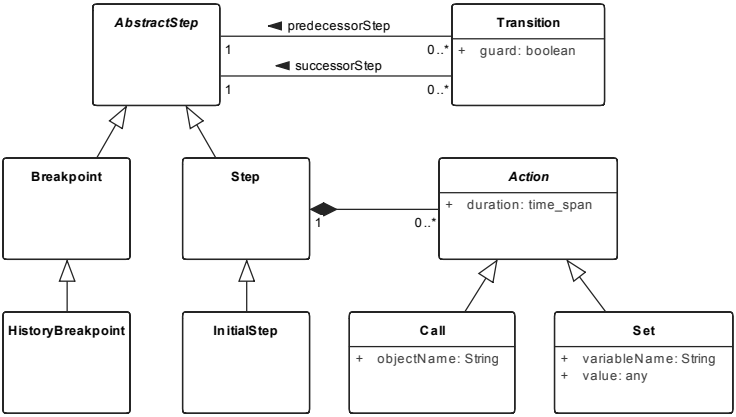


Abbildung 6.7.: Klassendiagramm des ISSCs.

Abwesenheit der hierarchischen Zustände bereits eingeschränkt und kann durch eine entry-Anweisung des nachfolgenden Zustands simuliert werden.

Basierend auf diesen Vorüberlegungen wird im Folgenden der syntaktische Aufbau der ISSCs sowie deren graphische Darstellung informell präsentiert. In Abbildung 6.6 ist ein einfaches ISSC mit jeweils zwei Zuständen und zwei Breakpoints dargestellt. Der untere Zustand wird dabei nur dann betreten, wenn die verbliebene Ausführungszeit mehr als 10 ms beträgt. Sonst wird die Ausführung des Charts für den aktuellen Zyklus in dem oberen Breakpoint abgebrochen. Weitere ISSC-Beispiele sind in darauffolgenden Abschnitten zu finden.

Die Struktur des ISSCs wird der Vollständigkeit halber als UML-Klassendiagramm in Abbildung 6.7 dargestellt. Die einzelnen Elemente sind:

Zustände (oder Schritte) Die Zustände der ISSCs repräsentieren den Fortschritt der damit beschriebenen Prozedur. Zu jedem Zeitpunkt ist genau ein Zustand des Charts aktiv. Zustände enthalten Aktionen, die beim Betreten des Zustands ausgeführt werden. Alle Aktionen eines Zustands werden in der vorgegebenen Reihenfolge innerhalb des Zyklus ausgeführt.

Breakpoints Neben den Anfangszuständen wird eine spezielle Klasse von Zuständen definiert, die „Breakpoints“ heißen. Dieser Klasse der Zustände können keine Aktionen zugeordnet werden. Somit haben Breakpoints einen einzigen Zweck: sie definieren Zeitpunkte, zu denen die Ausführung des ISSCs unterbrochen wird bzw. das ISSC verlassen werden kann. Das „Verlassen“ des Statecharts übergibt den Kontrollfluss nach dem Prinzip des hierarchischen Schedulings zurück zu dem übergeordneten Scheduler.

Transitionen und Guards Transitionen verbinden Zustände bzw. Breakpoints miteinander. Die Transitionen sind mit Guards beschriftet. Guards sind Ausdrücke, die zu einem booleschem Wert auswertbar sind. Eine Transition ist aktiv, falls der zugeordnete Guard zu TRUE evaluiert wird. Aktive Transitionen erlauben Übergänge zwischen den Zuständen entlang dieser. Die Guards haben keine Seiteneffekte und können auf unterschiedliche Art und Weise realisiert werden, z. B. durch explizite boolesche Ausdrücke oder durch einfache Funktionsbausteinketten. Eine Diskussion diesbezüglich ist in [Yu16] zu finden.

Transitionen besitzen eine eindeutige Evaluierungsreihenfolge und sichern somit einen deterministischen Zustandsübergang zu.

Ein Zustand, der kein Breakpoint ist, muss immer verlassen werden können, d. h. es muss sichergestellt werden, dass in jedem Fall mindestens eine aktive Transition existiert.

Aktionen Die SSCs nutzen zwei Typen von Aktionen: Aufrufe der lokalen Funktionsbausteine und Variablenzuweisungen. Die Zuweisungen werden in zwei weitere Kategorien unterteilt: die Zuweisungen an die Variablen des Ausführungsrahmens bzw. an die Variablen des Interfaces des SSCs sowie die Zuweisungen der Werte an die Eingänge lokaler Funktionsbausteine. Beide Typen werden von ISSC übernommen.

Im vorgestellten Meta-Modell werden Aktionen mit einer expliziten Ausführungsdauer versehen und bleiben für die Dauer der Ausführung ununterbrechbar. Aus diesem Grund wird auf die Unterscheidung zwischen Aktionen und Aktivitäten verzichtet. Die letzten können z. B. nach der Semantik von PLC-Statecharts von dem Zyklus der Laufzeitumgebung „angehalten“ werden [WVH11].

Lauf eines ISSCs Ein Lauf beginnt in dem designierten Anfangszustand bzw. in dem zuletzt aktiven Breakpoint (vgl. den Punkt „History-Verhalten der Breakpoints“). Nach der Ausführung der Aktionen werden die Guards der Transitionen in der vorgegebenen Reihenfolge ausgewertet. Bei dem ersten aktiven Guard folgt ein Übergang in den nächsten Zustand bzw. Breakpoint. Eine wichtige Implikation dieser Forderung ist die Abwesenheit der Deadlocks. Es muss immer ein Übergang zwischen zwei Zuständen (die keine Breakpoints sind) existieren, d. h. mindestens eine ausgehende Transition muss immer aktiv sein. Ein Lauf endet in einem Breakpoint, umgekehrt muss aber das Besuchen eines Breakpoints den Lauf nicht notwendigerweise beenden. Zu einem konkreten ISSC existieren typischerweise mehrere Läufe, die sich je nach den befolgten Transitionen unterscheiden können.

History-Verhalten der Breakpoints Breakpoints können explizit ein History-Verhalten besitzen (ein Buchstabe „H“ in einem Kreis, die Notation ist den Harel's Statecharts entliehen [Har87]). Falls der Lauf eines ISSCs in einem History-Breakpoint unterbrochen wurde (d. h. kein Guard der ausgehenden Transitionen aktiv war), beginnt die wiederholte Ausführung des ISSCs in diesem Breakpoint. Falls ein History-Breakpoint keine ausgehenden Transitionen besitzt, „verbleibt“ der ISSC nach dem Erreichen des letzten Breakpoints immer wieder in diesem Breakpoint. Das Chart muss in diesem Fall manuell zurückgesetzt werden, z. B. auf den Initialzustand.

Endzustände ISSCs verfügen über keine expliziten Endzustände. Stattdessen wird auf die Breakpoints zurückgegriffen. Die Konvention ist, dass der Breakpoint ohne ausgehende Transitionen einem Endzustand gleichwertig angesehen wird. Bei der Frage wann ein ISSC als „beendet“ oder „terminiert“ betrachtet wird, muss zwischen den Statecharts mit und ohne History-Verhalten differenziert werden. Ein ISSC ohne History-Verhalten ist mit dem Lauf beendet, weil der neue Durchlauf des Statecharts wiederum in dem Initialzustand anfängt. Ein ISSC mit History-Verhalten ist dann beendet, wenn ein Breakpoint ohne ausgehende Transitionen erreicht wurde.

Um zeitabhängiges Verhalten der Algorithmen beschreiben zu können, werden ISSCs mit einer zeitbehafteten Semantik ausgestattet, die sie sehr ähnlich zu TA macht.

Das Zeitverhalten eines ISSCs folgt den nachstehenden Grundsätzen:

- Die Zeit vergeht nur während der Ausführung der Aktionen in den Zuständen. In einem Zustand ohne Aktionen vergeht demnach keine Zeit.
- Transitionen sind zeitlos (die Dauer der Auswertung der einer Transition zugeordneter Guards kann bei Bedarf durch Aktionen des vorgehenden Zustands bei der Modellierung berücksichtigt werden).

ISSCs werden mit einer speziellen Variable – einer Uhr – ausgestattet. Diese ermöglicht das Abfragen der für die Komponente noch verfügbaren Ausführungszeit. Die Möglichkeiten der Auswertung dieser Zeit ist neben der in-cycle Semantik die grundlegende Erweiterung von Statecharts, die in keiner dem Autor bekannten Prozedurbeschreibungssprache der Domäne der Prozessleittechnik verfügbar ist. Die Kombination dieser Eigenschaft, der in-cycle Semantik und der Ausführungszeitsensitivität, macht Ablaufsteuerung der ressourcenadaptiven Anwendungen mit Mitteln der ISSCs erst möglich.

Es ist wichtig zwischen der Ausführungszeitsensitivität und der generellen Möglichkeit Zeitverhalten in die Beschreibungsmittel für Prozeduren einfließen zu lassen genau zu unterscheiden. Die Möglichkeit Timer bzw. Pausen zwischen den einzelnen Schritten bzw. Aktionen zu definieren, ist in den meisten Prozedurbeschreibungssprachen, wie z. B. IEC 61131-3 SFC vorhanden. Ein weiteres zeitabhängiges Verhalten existiert bei den SSCs: Diese erlauben die Abfragen des Ausführungsstatus gestarteter Unterabläufe. Der Fortschritt der Prozedur kann durch die Nutzung dieser Abfragen von den Zeitpunkten der Terminierung der Unterabläufe abhängen und ist somit auch implizit zeitsensitiv.

Diese Möglichkeiten reichen jedoch für die Beschreibung der Ablaufstruktur der RA-Algorithmen mit einer in-cycle Semantik nicht aus. Bei dem Timer handelt es sich um statische Pausen, die als absolute Zeitspannen ausgedrückt werden, z. B. fünf Sekunden.

Daher sind sie nicht für Beschreibung variabler temporaler Verhalten geeignet. Die Abfragen der Terminierung genügen diesem Ziel auch nicht, da sie keine Aussagen über den absoluten Zeitpunkt der Terminierung zulassen. Stattdessen wird die multi-cycle Semantik der SSCs verwendet: das Statechart überprüft die Terminierung durch das Abfragen einer speziellen Variable genau einmal pro Zyklus und Unterablauf.

Zusammengefasst lassen sich die Eigenschaften der eingeführten ISSC-Sprache im Vergleich zu SSC in der Notation der Veröffentlichungen [YGE13, Yu16] wie folgt beschreiben:

> ISSC ≈ SSC + Breakpoints + Ausführungszeitsensitivität + in-cycle Semantik − Zustandshierarchie − komplexer Ausführungsrahmen − do- und exit-Phasen.

Formal gesehen kann ein ISSC $\mathcal{I} = (S, B, H, G, \Delta, A, \alpha, O, s_0, c, w)$ als ein Tupel dargestellt werden, wobei S eine Menge der Zustände, B eine zu S disjunkte Menge der Breakpoints, H eine zu S und B disjunkte Menge der History-Breakpoints, G eine Menge boolescher Guard-Variablen $g \in \mathbb{B}$, $\Delta \subseteq (S \cup B \cup H) \times G \times (S \cup B \cup H)$ eine Transitionsrelation, A eine Menge der Aktionen, $\alpha : S \to 2^A$ die Zuordnung der Aktionen zu den Zustanden, O eine Menge der partiellen Ordnungen der Aktionen für jede Menge der Aktionen $\alpha(s)$ mit $s \in S$ und $s_0 \in S$ der Initialzustand ist. Eine einfache Zuordnung der Aktionen auf Zustände ist ausreichend, da nur Aktionen in der entry-Phase eines Zustands zugelassen sind. Es wird eine möglichst einfache mathematische Modellierung von ISSC beabsichtigt. So werden z. B. die Variablen auf denen die Guards operieren nicht explizit modelliert, sondern nur als bereits ausgewertete Ausdrucke miteinbezogen. Es wird angenommen, dass ISSCs deterministisch bezüglich der Transitionen sind. Diese Forderung impliziert den wechselseitigen Ausschluss der Transitionen eines Zustands bzw. deren Guards. Letzteres kann durch das Bilden zusätzlicher Guard-Variablen, die eine Konjugation negierter Guards benachbarter Transitionen oder die Zuordnung der Prioritäten an die Transitionen, erreicht werden. Der letzte Weg wird oft bei der graphischen Definition der SSCs bzw. SFCs gewählt, z. B. durch eine explizite Nummerierung oder durch die Priorisierung der Transitionen in der Reihenfolge „von links nach rechts“ bezüglich der graphischen Darstellung des Zustands bzw. des Breakpoints.

Eine Uhr c, die vom Statechart selbst nicht zurückgesetzt wird, jedoch in die Guards eingehen kann, erweitert das Modell. Die Uhr hat den Wert der Zeit, die seit dem Beginn der von dem ISSC gesteuerten Komponente vergangen ist und kann jederzeit von einem Guard abgefragt werden. Der letzte Eintrag des Tupels ist einer Funktion $w : A \to \mathbb{R}^+$, die die WCET der Ausführung jeweiliger Aktionen darstellt.

Es ist dem aufmerksamen Leser aufgefallen, dass die Uhr c nicht der Uhr t für die verbliebene Ausführungszeit der Komponente aus Abbildung 6.6 entspricht. Dieser Wert kann einfach als Differenz $t = \Delta^{curr} - c$ dargestellt werden, wobei Δ^{curr} die aktuell zugeteilte Ausführungszeit des ISSCs ist.

Die Ähnlichkeit der ISSCs zu TA lässt eine relativ einfache Abbildung der ISSCs auf diese zu, was Gegenstand des nächsten Abschnitts ist.

6.2.4. Formalisierung der ISSC-Semantik mithilfe von Timed Automata und UPPAAL

Die Forderung nach der Eindeutigkeit der Semantik von ISSC wird in diesem Abschnitt durch die Abbildung der Charts auf die TA-Netzwerke des UPPAAL Toolkits adressiert.

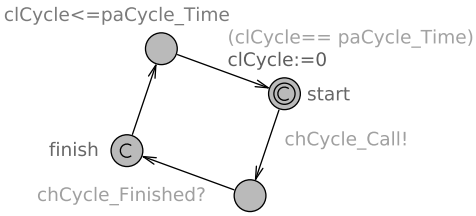


Abbildung 6.8.: Der Automat für den Hauptzyklus der Laufzeitumgebung.

Dieses Automatenmodell bietet einen pragmatischen Weg, formal eindeutige Modelle mit einer komfortablen Syntax (wie z. B. Guards, Unterteilung der Logik in mehrere Automaten sowie Kommunikationskanäle zwischen diesen) zu erstellen. In seiner hauptsächlichen Anwendung als Model-Checker bietet UPPAAL weiterhin die Möglichkeit die Eigenschaften der abgebildeten Automaten abzufragen.

Zunächst werden die Elemente des ISSC-Modells auf die Elemente des Automatenmodells von UPPAAL abgebildet. Die Grundlage für dieses Modell bilden Vorarbeiten [YGE13] bzw. Arbeiten von Witsch et al. [WRKVH10, WVH11], in denen das Verhalten von SSCs bzw. PLC-Statecharts auf einer zyklischen Laufzeitumgebung mithilfe von UPPAALs TA-Netzwerken formalisiert wurde.

In Abschnitt 2.1.4 wurde bereits eine kurze Einführung in UPPAAL präsentiert. Für die Modellierung gelten die folgenden Konventionen der Benennung bzw. der Darstellung:

- Guards der Transitionen werden in Abbildungen immer in Klammern aufgeführt, z. B. „`(clCycle==paCycle_Time)`“.
- Invarianten der Zustände (d. h. Bedingungen die während der Aktivität des Zustandes gelten) stehen in unmittelbarer Nähe des Zustands und beinhalten einen Vergleichsoperator, z. B. „`clCycle<=paCycle_Time`“.
- Namen der Zustände stehen in unmittelbarer Nähe dieser und beinhalten keine Operatoren.
- Synchronisationsaufrufe (d. h. Sende- und Empfangsoperationen auf den binären Kommunikationskanälen) sind an den abschließenden Sende- (`!`) bzw. Empfangsoperatoren (`?`) erkennbar, z. B. „`chCycle_Call!`“.
- Uhrenvariablen beginnen mit einem Präfix „`cl`“, Parameter beginnen mit einem Präfix „`pa`“ und Kommunikationskanäle beginnen mit einem Präfix „`ch`“.
- In TA-Zuständen mit einer Markierung „C“ (committed) vergeht keine Zeit.

Zunächst wird der Hauptzyklus der Laufzeitumgebung modelliert, der in Abbildung 6.8 dargestellt ist. Der Lauf beginnt im Zustand „`start`“, in dem keine Zeit vergehen darf. Die erste Transition emittiert ein Signal über den Kanal „`chCycle_Call`“. Danach wartet der Automat solange, bis ein Signal über den Kanal „`chCycle_Finished`“ empfangen wird. Durch eine Invariante verbleibt der Lauf solange in dem oberen Zustand bis die interne Uhr „`clCycle`“ den Wert des Parameters „`pCycle_Time`“ – der Zykluszeit – erreicht. Im

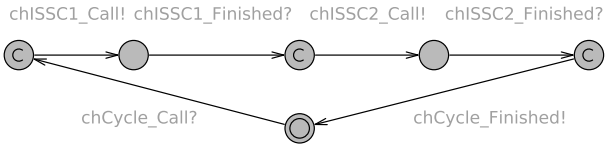


Abbildung 6.9.: Der Automat für den Komponentenscheduler (in diesem Fall werden zwei ISSCs innerhalb des Grundzyklus ausgeführt).

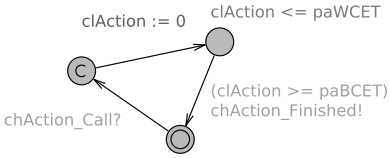


Abbildung 6.10.: Das Template für die Modellierung einer Aktion.

Anschluss daran vollzieht sich ein Übergang in den Startzustand und die Uhr wird zurückgesetzt. Im Gegensatz zu den Vorarbeiten wird der Einfachheit halber das Einlesen der Ein- und Ausgänge der Laufzeitumgebung nicht explizit im Hauptzyklus modelliert. Bei Bedarf kann dies komplikationslos hinzugefügt werden.

Im nächsten Schritt wird der Komponentenscheduler modelliert. Für die Zielsetzung der einfachen Modellierung der ISSC-Semantik reicht ein einfacher Ansatz für den Komponentenscheduler aus. Dabei wird die Ausführung einer einfachen Liste nachgebildet. Der Automat ist in Abbildung 6.9 zu finden. Der Lauf des Automaten wird ausschließlich durch Synchronisationsaufrufe bestimmt. Der Beginn des Hauptzyklus, der vom Automaten in Abbildung 6.8 durch das Senden auf dem Kanal „`chCycle_Call`“ kommuniziert wird, löst die sequentielle Ausführung von zwei ISSCs durch die passenden Kommunikationssignale aus. Nach dem Beenden des zweiten Statecharts wird das Ende des Zyklus durch den Kanal „`chCycle_Finished`“ an den Hauptzyklus der Laufzeitumgebung (Automat in Abbildung 6.8) übermittelt.

Im Folgenden werden die Regeln für die Abbildung einzelner Elemente der ISSCs auf die Elemente der TA-Netzwerke vorgestellt. Zunächst muss die Modellierung der Aktionen angesprochen werden, die in den einzelnen Zuständen der Statecharts aufgerufen werden. Bei der Modellierung werden die genauen Auswirkungen der Aktionen nicht berücksichtigt, da nur die Ausführungsdauer dieser im Mittelpunkt der Betrachtung steht. Somit ist eine Aktion nach dem Template modellierbar, das in Abbildung 6.10 dargestellt ist. Die Kommunikationskanäle „`chAction_Finished`“ und „`chAction_Call`“ sowie die Parameter „`paWCET`“ und „`paBCET`“ werden für die jeweilige Instanz der Aktion durch globale Kanäle bzw. Parameter initialisiert. Die Parameter entsprechen der WCET bzw. der Best Case Execution Time (BCET) einer Aktion. Die Ausführung der Aktion wird durch ein Signal auf „`chAction_Call`“ initiiert. Dabei wird die interne Uhr „`clAction`“ zurückgesetzt. Danach erfolgt die Simulation der Ausführungszeit: durch das Zusammenspielen der Zustandsinvariante „`clAction <= paWCET`“ mit dem Guard „`clAction >= paBCET`“ ist der Automat gezwungen in dem oberen Zustand für die Zeitdauer aus dem Intervall

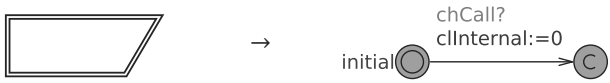


Abbildung 6.11.: Transformation des Initialzustands.

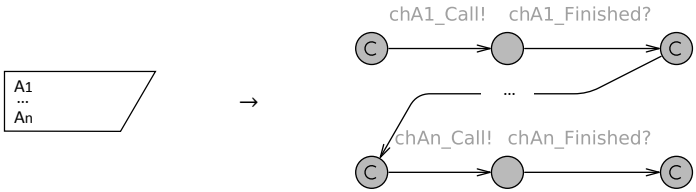


Abbildung 6.12.: Transformation eines Zustands mit Aktionen.

[paBCET, paWCET] zu bleiben, bevor ein Signal auf „chAction_Finished“ ausgelöst wird und damit das Ende der Ausführung der Aktion markiert. Die BCET wurde ausschließlich für Testzwecke abgebildet und kann durch die Belegung mit Null aus dem Modell „ausgeblendet“ werden.

Das Modellieren der ISSCs in UPPAAL kann wegen der großen Ähnlichkeit beider Modelle mithilfe einfacher Überführungsmuster auf der Ebene einzelner Elemente stattfinden. Für die Modellierung wird davon ausgegangen, dass der Automat eine interne Uhr „clInternal“ (die der Uhr c aus der formalen Definition von ISSC entspricht) und einen Parameter „paD_T_Curr“ besitzt. Letzter enthält die für die Ausführung der ISSCs zur Verfügung stehende Zeit und entspricht somit der Variable Δ^{curr} aus Abschnitt 6.2.2. Analog zum Modell der Aktionen stehen ferner zwei Kommunikationskanäle „chCall“ und „chFinished“ für das Model des ISSCs zur Verfügung.

Der Initialzustand des Statecharts wird auf die UPPAAL Konstruktion in Abbildung 6.11 überführt. Der Automat wartet in seinem Anfangszustand „intial“ auf die Synchronisation über „chCall“ und setzt die interne Uhr zurück, um die Zeit der eigenen Ausführung messen zu können. Die in einem Initialzustand des Charts enthaltene Aktionsaufrufe können durch die im Folgenden beschriebene Abbildung der Zustände abgedeckt werden.

Nicht-Breakpoint Zustände werden durch die Transformation in Abbildung 6.12 zu einer Kette aus Automatenzuständen umgewandelt. Jeder Zustandsübergang löst eine Aktion aus, die durch einen Automaten in Abbildung 6.10 modelliert wird. Die Aktionen werden in der Reihenfolge „von oben nach unten“ abgearbeitet.

Als nächstes werden die Transitionen zwischen den Statechart-Zuständen abgebildet.

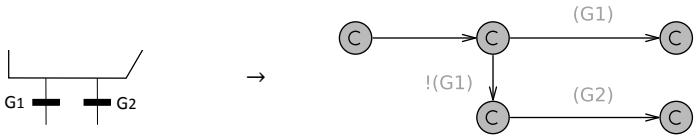


Abbildung 6.13.: Transformation der Transitionen (Priorisierung „von links nach rechts“).

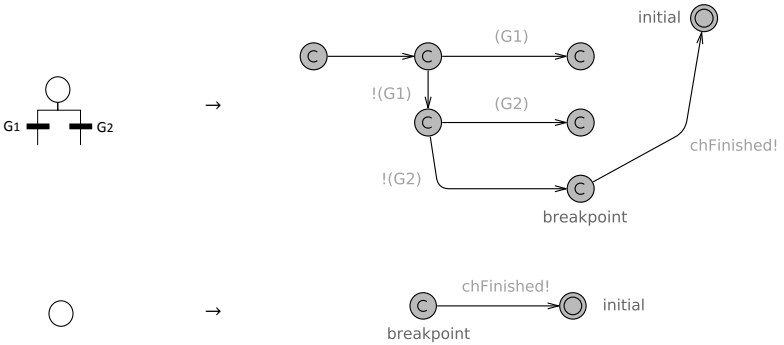


Abbildung 6.14.: Transformation eines Breakpoints ohne History-Verhalten (Priorisierung der Transitionen „von links nach rechts“).

Diese werden durch das Muster in Abbildung 6.13 überführt (beispielhaft für zwei Guards). In der Abbildung wird die explizite Priorisierung der Transitionen in der Reihenfolge „von links nach rechts“ angenommen. Es wird davon ausgegangen, dass die Guards des ISSCs auf zwei Arten boolescher Variablen operieren. Diese sind Variablen der gesteuerten Funktionsbausteine und Variablen der internen Uhr c, die den Fortschritt der Zeit während der Ausführung des Statecharts misst. Die erste Kategorie der Variablen wird einfach auf eine Menge boolescher Variablen in UPPAAL abgebildet und muss somit extern belegt werden. Die Guards mit der Uhr-Variable werden dagegen ins UPPAAL Modell als Transitionen-Guards direkt übernommen. Die Abfrage der für die Ausführung noch verfügbaren Zeit kann im ISSC durch die Abfrage der Differenz ($\Delta^{curr} - c$) erfolgen wobei c die interne Uhr des Statecharts ist. Diese Abfragen innerhalb der Guards können direkt in UPAALL auf die Differenzen `paD_T_Curr` − `clInternal` abgebildet werden. Man stellt fest, dass in der vorgestellten Transformation keine Möglichkeit existiert den Lauf des TA fortzusetzen, wenn der letzte Guard (`G2` im Beispiel in Abbildung 6.13) nicht erfüllt ist. Dieses Verhalten bildet die in-cycle Semantik der ISSCs nach – die Ausführung der Charts darf nicht außerhalb der Breakpoints unterbrochen werden.

Die verbliebenen Elemente der Statecharts sind die Breakpoints. Deren Verhalten hängt davon ab, ob der Breakpoint ein History-Verhalten besitzt (also ob Breakpoint b in der Menge H in der Definition des ISSCs enthalten ist). Es wird zunächst der Fall mit abgeschaltetem History-Verhalten betrachtet. Ein Breakpoint wird beispielhaft in Abbildung 6.14 in die TA-Syntax überführt. Im Unterschied zu einem normalen Zustand, wird nach der negativen Überprüfung des letzten Guards (hier, `G2`) die Ausführung des Charts durch den Kanal „`chFinished`“ unterbrochen und das Chart springt zurück in den Initialzustand des Automaten in Abbildung 6.11. Die Zeit darf nur im Initialzustand vergehen – somit „wartet“ das Chart in diesem Zustand auf die nächste Ausführung.

History-Breakpoints werden durch die Transformation in Abbildung 6.15 umgewandelt. Der hauptsächliche Unterschied zu der Transformation in Abbildung 6.14 sind die Eigenschaften des Zustands „breakpoint“. Nach der Auswertung des letzten Guards signalisiert das Chart das Beenden des aktuellen Laufs durch den Kanal „`chFinished`“, wartet aber

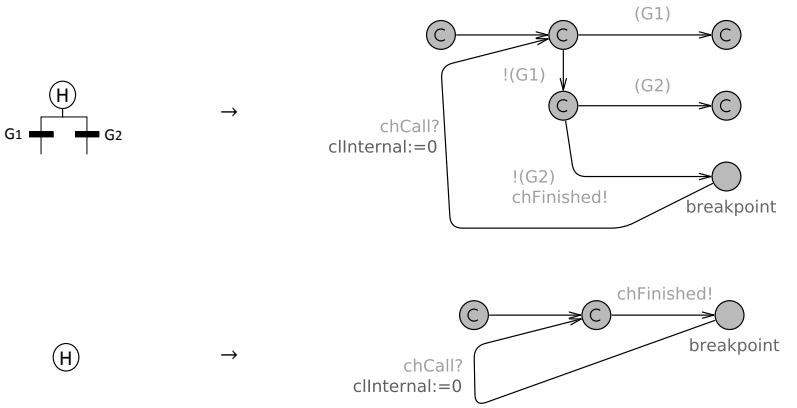


Abbildung 6.15.: Transformation eines Breakpoints mit History-Verhalten (Priorisierung der Transitionen „von links nach rechts“).

in diesem Zustand. Im Gegensatz zu der Transformation ohne History-Verhalten darf in dieser Konfiguration nun die Zeit vergehen (der Zustand ist dementsprechend nicht explizit als „committed“ markiert worden). Die erneute Ausführung des ISSCs beginnt somit genau in diesem Zustand und nicht in dem Initialzustand des Charts. Dabei wird ähnlich zu der Transformation in Abbildung 6.11 die interne Uhr zurückgesetzt. Im speziellen Fall, dass der History-Breakpoint keine ausgehende Transitionen besitzt, erzeugt die Transformation eine nicht-blockierende Schleife in der keine Zeit verstreichen darf (vgl. Abbildung 6.15 unten). Damit verbleibt, der Semantik der ISSCs entsprechend, der Automat in einer „Senke“.

Mit den beschriebenen Elementen ist die Transformation und somit das Verhalten der ISSCs vollständig. Durch die Abbildung auf UPPAAL TA-Netzwerke ist aber nicht nur das Verhalten formal beschrieben worden, sondern auch der Weg für die Anwendung des im UPPAAL enthaltenen Computational Tree Logic (CTL) Model-Checkers offen. Die möglichen Szenarien für dessen Nutzung während des Engineerings werden in Abschnitt 6.2.6 aufgeführt.

Die beschriebene Transformation ist für ein komplettes ISSC beispielhaft in Abbildung 6.16 dargestellt. Die Ähnlichkeit der Struktur des Charts im oberen Teil der Abbildung und des UPPAAL-Automaten im unteren Teil ist nicht zu übersehen – die Zustände und die Aktionen werden einfach übernommen. Die Guards sind übernommen worden und mit zusätzlichen ausschließenden Bedingungen ergänzt worden, damit der UPPAAL-Automat deterministisch ist. Ein komplettes Automatennetzwerk, welches diesen Automaten enthält, ist in Abbildung 6.19 auf Seite 90 zu sehen.

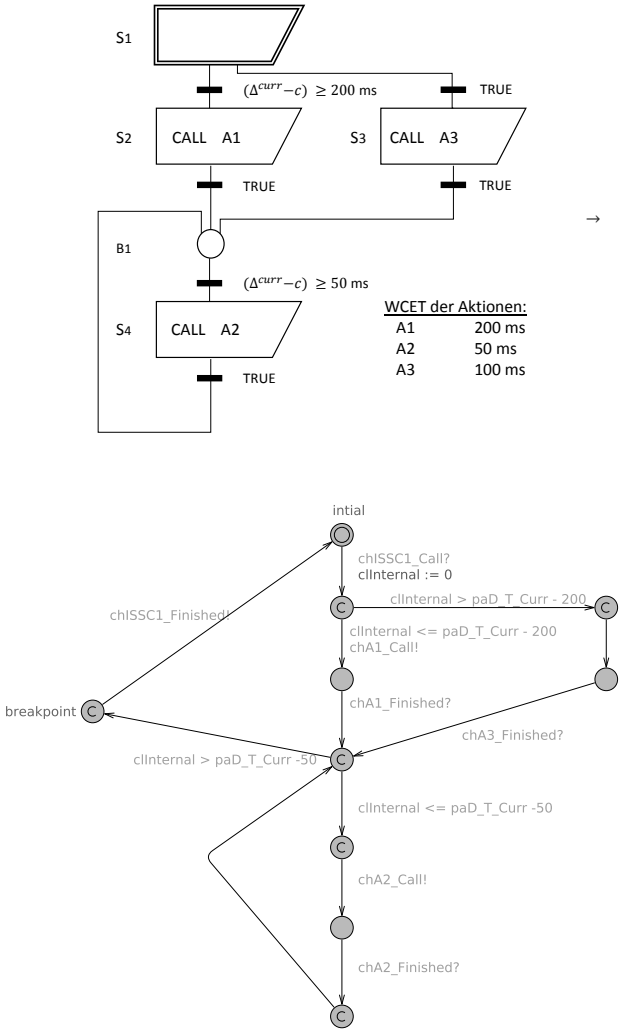


Abbildung 6.16.: Transformation eines ISSCs nach UPPAAL an einem einfachen Beispiel.

6.2.5. Beschreibung ressourcenadaptiver Algorithmen mit ISSC – Konventionen und Muster

Konventionen

Mit den vorgestellten In-cycle Sequential State Charts lässt sich die Ausführung verschiedenster POUs komfortabel beschreiben. Die Beschreibung der Abläufe durch den Nutzer bringt aber auch Gefahren mit sich. In diesem Abschnitt werden drei Konventionen für ISSC vorgeschlagen, die diese Gefahren minimieren sollen. Diese Konventionen stellen keine Einschränkung der Beschreibungsstärke der Statecharts dar und erleichtern in vielen Fällen das Engineering.

Eine Gefahr bei der Definition der ISSCs ist die Entstehung von Deadlocks, d. h. Zuständen, die über keine aktive Transition verlassen werden können. Durch einen solchen Zustand könnte, unter den angenommenen Bedingungen, ein Zyklus nicht terminieren bzw. ein nicht-konsistenter Zustand entstehen, falls man den Deadlock gezwungenermaßen durch den Abbruch der ISSC-Ausführung auflösen muss.

Eine zusätzliche Gefahr bietet die von SSC bzw. SFC teilweise geerbte Syntax des Beschreibungsmittels. Da sowohl SSC als auch SFC durch die multi-cycle Semantik keine Vollständigkeit der Ausgangstransitionen fordern, könnten Nutzer in der Engineering-Phase mögliche Deadlocks übersehen. Diese Problematik lässt sich mithilfe des UPPAAL Model-Checkers lösen, durch den man die Abwesenheit von Deadlocks beweisen kann (vgl. Abschnitt 6.2.6). Wegen der partiellen Abbildung der Guards in UPPAAL (es werden nur die Guards abgebildet, die explizit über die Zeit quantifizieren) können aber nicht alle Guards überprüft werden. Diese Problematik kann durch die explizite Einführung einer Standard-Transition mit Guard „TRUE“, die als letztes für jeden Zustand ausgewertet wird und immer aktiv ist, gelöst werden. Diese Konvention entspricht dem „else“ Konstrukt in den Verzweigungen der strukturierten Programmiersprachen. Somit gilt:

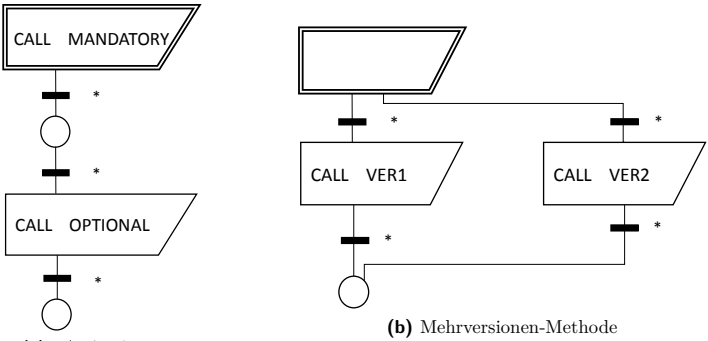
> **Konvention 1**: Jeder Zustand, der kein Breakpoint ist, muss mindestens eine ausgehende Transition besitzen, deren Guard mit TRUE ausgewertet wird.

Ein weiteres Problem sind die Zyklen innerhalb der Statecharts, die eventuell nie verlassen werden können und für den Gesamtzyklus der Laufzeitumgebung somit ähnliche Probleme wie Deadlocks verursachen können. Aus diesem Grund werden die Zyklen zwischen Nicht-Breakpoints ausdrücklich verboten. Es ist wichtig zu bemerken, dass Zyklen, bei denen die Häufigkeit der Ausführung zur Übersetzungszeit bekannt ist (z. B. eine 10-fache Ausführung eines Zustands), nicht als Zyklen in diesem Kontext gelten, da sie einfach „ausgerollt“ werden können und die Terminierung des Charts nicht gefährden. Zusammengefasst:

> **Konvention 2**: Ein ISSC darf nur Zyklen zwischen den Breakpoints enthalten. Die Abschnitte zwischen den Breakpoints sind demzufolge zykelfrei.

Diese Konvention erzwingt eine im Voraus bekannte obere Schranke für die Ausführungszeit jedes Abschnittes zwischen den Breakpoints eines ISSCs.

Eine weitere Ursache für Deadlocks kann die fehlerhafte Formulierung der Zeitguards, d. h. der Teilausdrücke der Guards, die über die interne Uhr quantifizieren, sein. Aus den Beispielen im letzten Abschnitt wird sichtbar, dass die Guards für die interne Uhr häufig



(a) 0/1-Ausführung

(b) Mehrversionen-Methode

Abbildung 6.17.: ISSC-Muster für ressourcenadaptive Algorithmen.

nur das „sichere Betreten“ des jeweiligen Zweigs garantieren. In diesem Fall kann die Formulierung der Guards automatisch geschehen. Somit gilt:

> **Konvention 3**: Soll nur das sichere Betreten des Zweiges/des Zustands garantiert werden, so werden die entsprechenden Zeitguards *automatisch* berechnet.

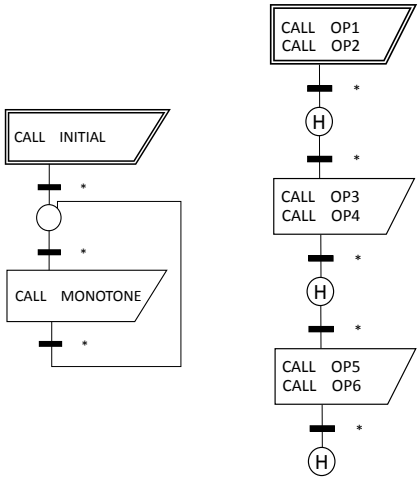
Im Falle der Abwesenheit der logischen Bedingungen reicht ein einfacher Algorithmus zur Berechnung der Zeitguards aus. Dieser wird in Abschnitt 6.2.6 vorgestellt. In der graphischen Darstellung wird aus diesem Grund im Folgenden auf die explizite Angabe dieser Guards verzichtet.

ISSC-Muster für RA-Algorithmen innerhalb der POUs

In diesem Abschnitt werden die Grundmuster vorgestellt, die unter anderem für die Implementierung der adaptiven Anwendungen nach den Paradigmen des Imprecise Computation Models (vgl. Abschnitt 5.1) notwendig sind. Dafür wird die 0/1-Ausführung, die Mehrversionen-Methode und die Meilenstein-Methode in ISSC nachgebildet. Darüber hinaus wird die Implementierung von Transaktionen angesprochen.

Die dargestellten Muster machen von den Konventionen des letzten Abschnitts Gebrauch. So werden z. B. die Guards der Transitionen nicht um die Zeitguards erweitert. Es wird davon ausgegangen, dass diese automatisch berechnet werden. Die zu berechneten Guards wird im Folgenden durch den Platzhalter „*“ angedeutet. Das Ziel der Auflösung der Platzhalter ist somit nur das gefahrlose Betreten des jeweiligen Zweigs bzw. des Zustands sicher. Für die beispielhafte Anwendung der Muster und für viele Fälle in der Praxis reicht dieses Verhalten aus. Die Charts können bei Bedarf um komplexere Guards erweitert werden, die auch die Semantik der gesteuerten POU berücksichtigen können. Das können z. B. die Ergebnisse der Berechnung einzelner Bausteine oder die Reaktion auf einen externen Eingang der POU sein.

Als erstes wird die 0/1-Ausführung in Abbildung 6.17a vorgestellt. Das Chart besteht aus zwei Zuständen, die die beiden Teile des Algorithmus repräsentieren: den verpflichtenden und den optionalen Teil. Der Initialzustand stößt die Ausführung des Pflichtteils



(a) Meilenstein-Methode **(b)** Transaktion

Abbildung 6.18.: ISSC-Muster für ressourcenadaptive Algorithmen (Fortsetzung).

durch den Aufruf einer Aktion mit dem Namen „MANDATORY“ an. Durch den darauf folgenden Breakpoint kann die Ausführung des ISSCs bereits nach dieser einen Aktion terminieren bzw. terminiert werden. Sollte dies nicht der Fall sein, so kann die Ausführung im zweiten Zustand mit dem Aufruf der Aktion „OPTIONAL“ fortgesetzt werden. Diese ist dem optionalen Teil des Algorithmus zugeordnet. Danach wird der zweite Breakpoint erreicht und die Ausführung zwangsweise terminiert. Dieses einfache Muster lässt bereits die Ausdrucksstärke der ISSCs erahnen: Die Information zur Reihenfolge der Aktionen und deren optionale bzw. verpflichtende Ausführung ist einfach und kompakt dargestellt.

Das nächste Muster stellt in Abbildung 6.17b die Ausführungssteuerung einer Mehrversionen-Methode graphisch dar. Die Ausführung beginnt im Initialzustand. Da dieser keine Aktionen enthält, werden direkt die zwei ausgehenden Transitionen ausgewertet, um entweder den linken oder den rechten Zweig des ISSCs zu betreten. Angenommen, die WCET des linken Zweigs (der Version 1) ist kürzer als die des rechten (der Version 2). Der WCET Parameter w des Charts stellt die mögliche Ausführung des linken Zweigs und damit der Aktion „VER1“ sicher – die dafür benötigte Zeit wird von dem Komponentenscheduler garantiert. Falls mehr Zeit zur Verfügung steht, kann eventuell auch der rechte Zweig betreten werden. In diesem Fall wird die Aktion „VER2“ angestoßen. Der Breakpoint im unteren Teil der Abbildung stellt eine Terminierung sicher und dient als „Endzustand“.

Die beiden betrachteten Muster hatten eine begrenzte Höchstausführungsdauer. Im ersten Beispiel war das die Summe der Ausführungsdauer beider Algorithmen. Im zweiten Beispiel war das deren Maximum. Das nächste Muster in Abbildung 6.18a implementiert eine Meilenstein-Methode und kann somit beliebig lange ausgeführt werden. Der Chart führt zunächst eine Aktion „INITIAL“ aus, in der ein Grundergebnis berechnet wird. Im

weiteren Verlauf der Ausführung kann dieses Ergebnis durch die wiederholte Ausführung der Aktion „MONOTONE“ des zweiten Zustands verbessert werden. Im Gegensatz zu Abschnitt 5.1.5, wird in diesem einfachen Beispiel keine Trennung zwischen der Berechnung an sich und dem Meilenstein gemacht. Diese ist durch das Einführen weiterer Zustände und Breakpoints in der Chart-Schleife und somit einer Verfeinerung der nicht unterbrechbaren Abschnitte problemlos möglich. Auch das Unterbrechen nach dem Erreichen eines bestimmten Gütekriteriums, z. B. das Unterschreiten gewisser Fehlerabschätzung, kann mithilfe eines weiteren Breakpoints mit einer entsprechenden Transition erfolgen.

Das letzte Muster in Abbildung 6.18b implementiert eine Prozedur, die nicht zum Imprecise Computation Model gehört. Dieses Beispiel setzt eine Transaktion mit vordefinierten Unterbrechungspunkten um und macht dafür von Breakpoints mit History-Semantik Gebrauch. Die Transaktion stellt sicher, dass die in einem Zustand enthaltenen Aktionen garantiert innerhalb eines Zyklus der Laufzeitumgebung ausgeführt werden. Das trifft beispielsweise auf die Aktionen „OP1“ und „OP2“ zu. Die Ausführung kann zwischen den Zuständen „pausieren“, d. h. für eine unbestimmte Anzahl Zyklen im History-Breakpoint verbleiben. Ein Anwendungsszenario für dieses Muster ist das Laden von Instanzen in das Laufzeitsystem. Ähnlich zu einer Datenbank müssen manche Aktionen, wie z. B. das Einschalten der Bausteine, auf den Zyklus synchronisiert passieren. Nach dem Verlassen des letzten Zustands verbleibt die Transaktion in dem untersten Breakpoint. Damit ist eine einmalige Ausführung dieser sichergestellt. Die Anwendung dieses Musters wird im Abschnitt 7.5 anhand eines Use-Cases demonstriert.

6.2.6. Engineering-Aspekte

Abschätzung der WCET

Die Abschätzung der WCET eines Programms ist eine nichttriviale Aufgabe und hängt von vielen Faktoren ab, die die Laufzeit beeinflussen können. Einige Faktoren sind an das Programm selbst gebunden. Dazu gehört, z. B. der Kontrollfluss innerhalb der Programmlogik. Andere Faktoren hängen von der Hardware- und Softwareplattform des Systems ab. Zur ersten Kategorie gehören, z. B. die Cache- und Speicherarchitektur der Plattform. Zur zweiten Kategorie gehören die Architektur des Betriebssystems und die Eigenschaften dessen Subsysteme, z. B. des Schedulers und der Speicherverwaltung.

Die methodische Analyse der Verfahren zur Berechnung der WCET einzelner Aktionen bzw. Komponenten würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund wird an dieser Stelle eine bereits erfolgte Abschätzung angenommen. Eine Übersicht über gängige Laufzeit-Analyseverfahren ist in [WEE+08] und [SOG14] zu finden.

Simulation der ISSCs mit UPPAAL

Die Abbildung der ISSCs auf UPPAAL erlaubt die Nutzung des Tools für die Simulation und das Testen der Charts während des Engineerings. Der Funktionsumfang von UPPAAL ermöglicht den simulierten Durchlauf der Charts bzw. der abgeleiteten Automaten sowie das Aufzeichnen von Läufen des Charts.

Als Beispiel wird das System in Abbildung 6.19 betrachtet. Die einzelnen Automaten in der Simulationssicht sind durch die Anwendung der Transformation entstanden, die in Abschnitt 6.2.4 detailliert beschrieben wurde. Beispielsweise entspricht der als „ISSC1“

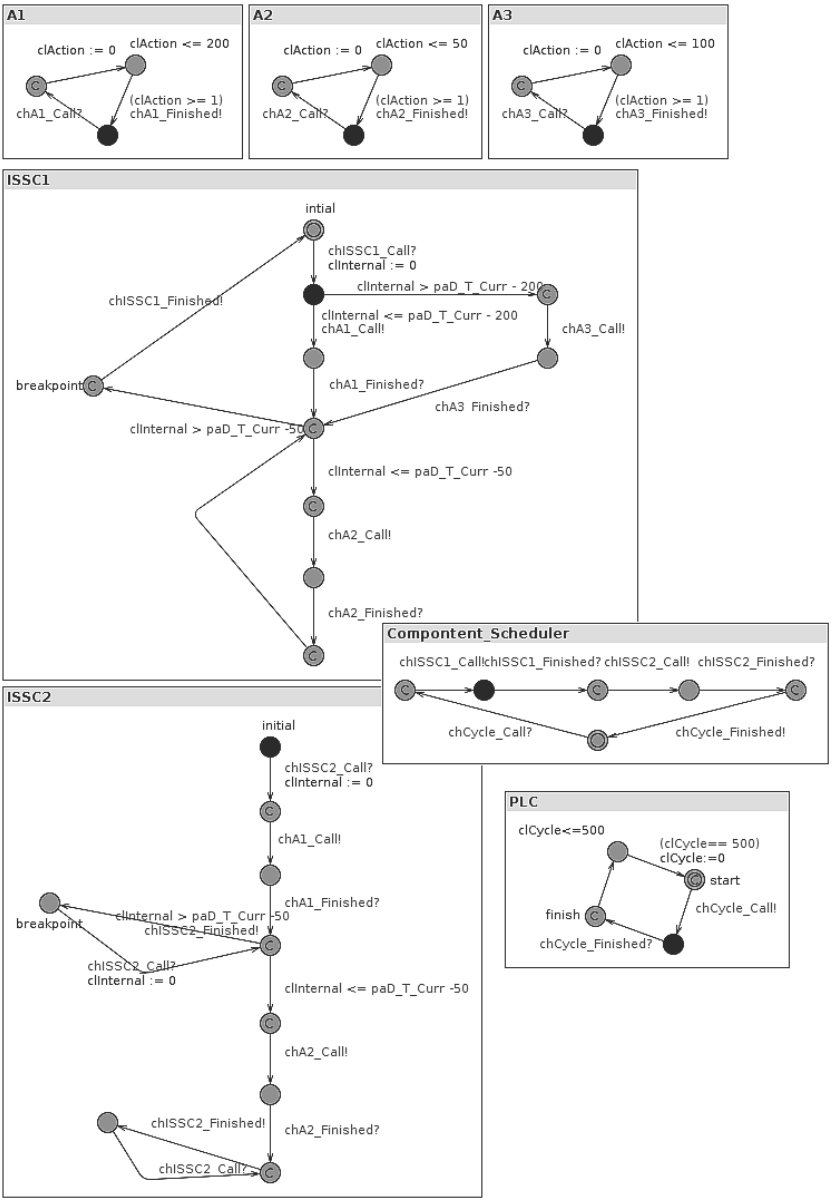


Abbildung 6.19.: Simulationssicht des UPPAAL-Tools, in der die zyklische Umgebung, ein Komponentenscheduler, zwei ISSCs und die dazugehörigen Aktionen ausgeführt werden.

bezeichnete Automat dem ISSC in Abbildung 6.16 auf Seite 85.

Anwendbarkeit des Model-Checkers

Neben der Simulation liegt eine weitere Anwendungsmöglichkeit von UPPAAL in der formalen Verifikation. Das Ziel ist es, bestimmte logische Aussagen zu dem definierten Systemmodell, in diesem Fall das abgebildete Modell der zyklischen Laufzeitumgebung und der darin eingebetteter ISSCs, zu bestätigen oder anhand eines Gegenbeispiels zu widerlegen.

Die Gefahren der Deadlocks wurden bereits in Abschnitt 6.2.5 angesprochen. Die Sicherstellung der Abwesenheit der Deadlocks ist die erste Anwendung des Model-Checkers und kann durch eine einfache Anfrage „`A[] not deadlock`“ überprüft werden.

Eine weitere Anwendung des Model-Checkers ist die grobe Abschätzung der erreichbaren Zustände der ausgeführten ISSCs. Beispielsweise kann mithilfe der Formel „`E<> ISSC1.optional`“ die Existenz eines Laufes, in dem ein ISSC mit Namen „`ISSC1`“ den Zustand „`optional`“ erreicht, überprüft werden. Der Quantor „E“ fordert die Existenz eines Pfades, der Diamond-Quantor „<>“ fordert das Erreichen des gewünschten Zustands auf diesem Pfad. In ähnlicher Weise kann untersucht werden, ob z. B. optionale Zustände immer besucht werden. In diesem Fall kann die Logik des Statecharts vereinfacht werden. Die TCTL Formel dazu Lautet in UPPAAL-Syntax „`A<> ISSC1.optional`“, der Existenz-Quantor wurde durch den All-Quantor „A“ ersetzt, der die Existenz der Eigenschaften auf allen Pfaden fordert.

Der Model-Checker kann weiterhin für grobe Abschätzungen der Verwertung der Zyklus-Zeit des Laufzeitsystems verwendet werden. Dazu wird der Wert der internen Uhr „`clCycle`“ der Laufzeitumgebung in Abbildung 6.8 untersucht. Man kann die Zykluszeit nach unten bzw. nach oben abschätzen, indem die Formeln für das Infimum „`inf{PLC.finish}: PLC.clCycle`“ bzw. das Supremum „`sup{PLC.finish}: PLC.clCycle`“ ausgewertet werden. Um die Abschätzung realistischer zu gestalten, kommen die bereits vorgestellten BCET-Parameter der Aktionen („`paBCET`“ in Abbildung 6.10) zum Einsatz. Diese Parameter können neben dem obligatorischen Parameter für die WCET optional modelliert werden. Durch die Auswertung der angegebenen Formeln werden Szenarien für den kürzesten bzw. den längsten möglichen Zyklus der Laufzeitumgebung generiert. Die Differenz des Supremums „`sup{PLC.finish}: PLC.clCycle`“ und der Zykluszeit „`paCycle_Time`“ in Abbildung 6.8 ist die, in jedem Fall, ungenutzte Slackzeit. Diese Zeit sollte im Rahmen der Systemauslegung minimiert werden.

Alle vorgestellten Überprüfungen können auch während des Engineering-Vorgangs durch eine automatische Transformation nach UPPAAL kontinuierlich vorgenommen werden.

Automatische Belegung der Zeitguards für einfache ISSCs

Die letzte Konvention des Abschnitts 6.2.5 fordert eine automatische Berechnung der Zeitguards, falls diese nur das „sichere Betreten“ der Verzweigungen bzw. Zustände garantieren sollen. Unter „sicher“ wird an dieser Stelle die Einhaltung der Echtzeitschranken verstanden. Diese Bedingung ist bei allen in Abschnitt 6.2.5 vorgestellten Mustern der Fall.

Da es nur um das sichere Betreten der Zustände geht, wird davon ausgegangen, dass bei dieser Problemstellung die Guards des ISSCs mit „*“ vorbelegt sind. Algorithmus 1 stellt eine skizzenhafte Implementierung der Berechnung der Zeitguards für ISSCs ohne History-Breakpoints dar. Neben den Zeitguards wird auch die WCET des ISSCs ausgegeben.

Eingabe : ISSC I, alle Guards mit $*$ belegt
Ergebnis : Beschriftung der Transitionen mit Zeitguards, WCET w des ISSCs I

Markiere jeden Zustand $s_i \in S$ mit $WCET(s_i) := \sum_{a_j \in \alpha(s_i)} WCET(a_j)$;
Markiere jeden Breakpoint $b_i \in B$ mit $WCET(b_i) := 0$;
Markiere jeden Zustand/Breakpoint $z_i \in S \cup B$ mit $total(z_i) := 0$;
Unterteile I in azyklische Abschnitte I_t zwischen $t \in B \cup \{s_0\}$;
foreach *Abschnitt* I_t **do**
    foreach *Zustand* $z_i \in S$ *von* I_t *in topologischer Reihenfolge* **do**
        if z_i *ist ein Blatt* **then**
            $total(z_i) := max\{total(z_i), WCET(s_i)\}$;
        else
            $min := \infty$;
            foreach *Nachfolger* $s_i \in S$ *von* z_i *in Priorität-Reihenfolge* **do**
                if $total(z_i) < min$ **then**
                    $min := total(z_i)$;
                end
            end
            $total(z_i) := max\{total(z_i), WCET(z_i) + min\}$;
        end
    end
end
foreach *Transition* t *von* s *nach* d *mit* $s \in B$ *und* $d \in S$ **do**
    Setze den Zeitguard von t als „$(\Delta^{curr} - c) \geq total(d)$“;
end
foreach *Transition* t *von* s *nach* d *mit* $s \in S$, s *hat mehrere Nachfolger und* $d \in S$ **do**
    Seien S' die Nachfolger von s;
    Ordne $s' \in S'$ aufsteigend bezüglich $total(s')$ in eine Ordnung O';
    Sei p die Position von d in O';
    if d *ist nicht des letzte Element in* O' **then**
        Sei $e \in S$ der Nachfolger von d bezüglich O';
        Setze den Zeitguard von t als „$total(e) > (\Delta^{curr} - c) \geq total(d)$“;
    end
end
Ersetze alle $*$ Bedingungen durch $TRUE$;
$w = total(s_0)$;

Algorithmus 1 : Skizzenhafte Berechnung der Belegung der Zeitguards für ISSCs ohne History-Breakpoints.

Zunächst werden die WCET einzelner Zustände der Charts $WCET(s_i)$ als Summe der WCET der enthaltenen Aktionen initialisiert. Die WCET der Breakpoints werden mit 0 belegt. Danach läuft der Algorithmus die einzelnen azyklischen Abschnitte des Charts topologisch „von unten nach oben“ entgegen der Ausführungsreihenfolge ab und berechnet zu jedem Knoten die Zeit *total*, die nach dem Betreten des Knotens für die Abarbeitung der Aktionen mindestens benötigt wird.

Die Abarbeitungszeit für die Blätter jedes Abschnittes wird gleich der WCET des jewei-

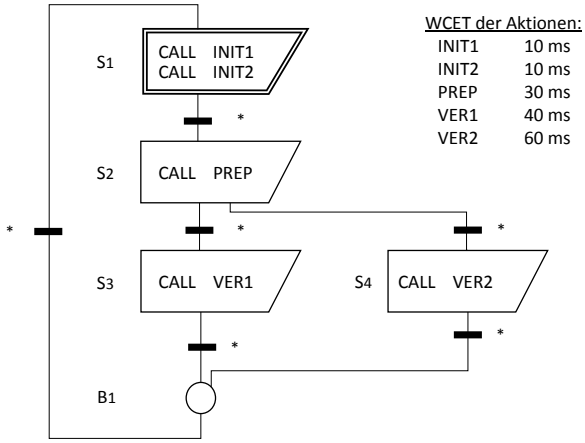


Abbildung 6.20.: Beispiel-ISSC für eine automatische Berechnung der Zeitguards.

ligen Zustands gesetzt. Für Knoten, die Nachfolger haben, wird in Zeile 11 die minimal benötigte Abarbeitungszeit für einen direkten Nachfolger berechnet. Die Abarbeitungszeit des Knotens wird als Summe der WCET der Zustand-Aktionen und dem Minimum der Zeiten der aktiven Nachfolger gesetzt.

Als nächster Schritt werden in der Schleife (Zeilen 20-22 des Algorithmus 1) die Guards der aus Breakpoints ausgehenden Transitionen um die berechneten Zeitguards ergänzt. Der Einfachheit halber werden nur die Transitionen von Zuständen mit mehreren Nachfolgern oder einem Breakpoint ausgehend ergänzt. Bei anderen, vor allem linear aufeinander folgenden Zuständen, reicht die zugesicherte Zeit für die Ausführung aller Aktionen aus. Transitionen, die in Breakpoint-Zuständen enden, werden nicht modifiziert – das Betreten der Breakpoints soll immer möglich sein. Die auf diese Weise ergänzten Guards garantieren zum einen die Abwesenheit der Deadlocks in dem Chart. Zum anderen werden die Zustände bzw. Zweige des Charts nur dann betreten, wenn die Ausführungszeit für die Abarbeitung der Zustände bzw. deren Aktionen ausreichend ist.

Die in die Zustände mit mehreren Nachfolgern eingehenden Transitionen werden in Zeilen 23-32 bearbeitet. Dort werden für jeden Zustand die nachfolgenden Zustände nach der Abarbeitungszeit *total* sortiert und die Guards so gesetzt, dass der Zustand nur betreten wird, falls für deren Abarbeitung die verbliebene Abarbeitungszeit ausreicht.

Die Anweisung in Zeile 33 schätzt die WCET w des gesamten Charts ab, die dem Komponentenscheduler mitgeteilt wird. Im Falle der Abwesenheit der History-Breakpoints gleicht diese Zeit der Abarbeitungszeit des an dem Initialzustand s_0 hängenden azyklischen Abschnitts. Diese Zeit ist in der Variable $total(s_0)$ gespeichert.

Die Anwendung des Algorithmus wird am Beispiel des ISSCs in Abbildung 6.20 illustriert. Das Chart beschreibt ein gemischtes Zeitverhalten. Die Verzweigung vom Zustand S_2 zu den Zuständen S_3 und S_4 folgt dem Mehrversionen-Muster. Die Schleife vom Breakpoint B_1 zu dem Initialzustand S_1 folgt dem Meilenstein-Muster. Die Abschätzung der Laufzeiten einzelner Aktionen der Zustände sind ebenfalls in Abbildung dargestellt. Als

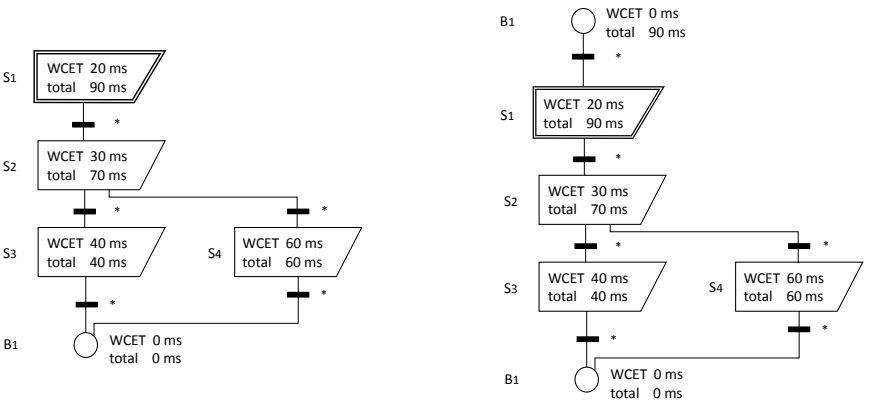


Abbildung 6.21.: Azyklische Abschnitte des ISSCs aus Abbildung 6.20 und die berechneten Markierungen der Zustände.

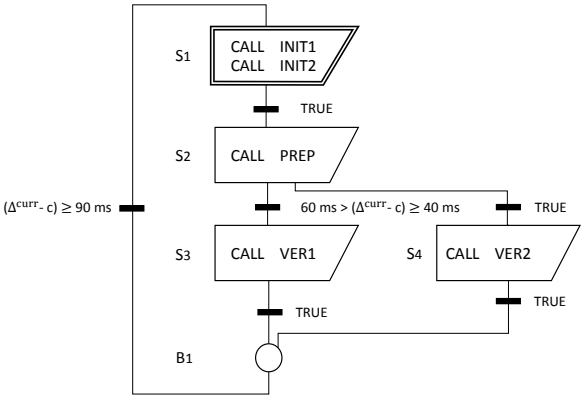


Abbildung 6.22.: ISSC aus Abbildung 6.20 nach der Terminierung des Algorithmus 1 inklusive der berechneten Zeitguards. Die WCET des ISSC beträgt 90 ms.

erstes wird das Chart in azyklische Abschnitte zwischen den Breakpoints bzw. zwischen dem Initialzustand und einem Breakpoint unterteilt. Das Ergebnis dieser Prozedur ist in Abbildung 6.21 zu finden. In beiden Abschnitten werden die Beschriftungen der Zustände WCET und *total* nach dem Algorithmus 1 „von unten nach oben“ berechnet. Aus diesen Informationen werden die Guards für einzelne Transitionen berechnet, wie die Abbildung 6.22 zeigt. Man sieht, dass nur zwei der Transitionen zusätzliche Guards bekommen haben. Damit ist ein Übergang aus dem Zustand S_2 weiterhin sichergestellt bzw. der Breakpoint B_1 wird nur bei ausreichend verfügbarer Ausführungszeit verlassen. Die gesamte WCET von 90 ms des Charts beläuft sich auf den WCET-Wert des Initialzustands S_1.

Die History-Breakpoints fordern eine leichte Abwandlung des Algorithmus für die Berechnung der WCET des gesamten Charts. Da die Ausführung des ISSCs theoretisch in jedem dieser Breakpoints starten kann, muss das Maximum der Abarbeitungszeit von jedem azyklischen Abschnitt, der an dem Initialzustand oder einem History-Breakpoint hängt, gebildet werden. Somit wird dem Chart vom Komponentenscheduler garantiert ausreichend Ausführungszeit zur Verfügung gestellt.

Eine weitere abgewandelte Version des Algorithmus kann zu der Berechnung der maximalen Laufzeit des ISSCs verwendet werden. Dieses ist nur bei zykelfreien Charts sinnvoll, z. B. bei der Mehrversionen-Methode oder der 0/1 Ausführung.

6.3. Ressourcenadaptiver Komponentenscheduler für zyklische Laufzeitsysteme

In den letzten zwei Abschnitten wurden eine Softwarearchitektur für die Ausführung und ein Meta-Modell für die Beschreibung von RA-Algorithmen eingeführt. In diesem Abschnitt wird ein Referenzmodell für einen Komponentenscheduler vorgestellt, der die Zuteilung der Ausführungszeit an die RA-Komponenten übernimmt.

Für dieses Referenzmodell wurde ein Ansatz ausgewählt, dessen Grundprinzip dem Predictably Flexible Real-Time Scheduling Modell ähnelt (vgl. Abschnitt 5.1.6). Die Aktivitäten des Schedulers werden in eine offline und eine online Phase aufgeteilt. Damit werden die Vorteile der Vorhersagbarkeit sowie einer relativ einfachen Laufzeitimplementierung mit einer gewissen Flexibilität kombiniert. Die Flexibilität ist notwendig für die optimale Ressourcennutzung des Systems. Der erfolgreiche Einsatz der offline Scheduler in den leittechnischen Laufzeitsystemen, z. B. FASA oder ACPLT/RTE (vgl. Abschnitte 3.3.3 bzw. 3.3.4), spricht ebenfalls für die Anwendung eines solchen Verfahrens. Ein mit einem offline Scheduler ausgestattetes Laufzeitsystem erfüllt somit die Erwartungen der Nutzerkreise in der Domäne der Leittechnik (Anforderungen N1 und N5 aus Kapitel 4).

6.3.1. Nomenklatur

Für das Scheduling der Komponenten gelten weiterhin Grundannahmen, die im den Abschnitten 2.1.2 und 6.2.1 erläutert wurden. Die klassische Definition des Tasks aus Abschnitt 2.1.2 wird für die folgenden Ausführungen abgewandelt. Ein Task T_i ist ein Tupel, der aus folgenden Komponenten besteht (diese Parameter sind der Tasking-Facette der Komponente entnommen):

- WCET der Komponente $w_i \in \mathbb{R}^+$ in Zeiteinheiten,

- Periode $per_i \in \mathbb{N}$ in Anzahl der Grundzyklen des Laufzeitsystems und
- der Flexibilitätsparameter $flex_i \in \{0, 1\}$ der Komponente.

Die Dauer des Grundzyklus wird als *cyctime* bezeichnet. Der boolesche Parameter $flex_i$ gibt an, ob die Komponente mehr Rechenzeit als w_i sinnvoll verwenden kann. Dieses ist beispielsweise bei den durch ISSCs gesteuerten Komponenten aus dem Abschnitt 6.2 der Fall. Tasks mit $flex_i = 0$ werden für die Kapselung der bereits existierenden Software verwendet. Der Parameter per_i eines Tasks zeigt an, dass eine Menge periodischer Tasks T gescheduled wird. Es wird zunächst angenommen, dass eine Task Instanz, ein Job, zwischen dem ersten und dem letzten Grundzyklus der Task-Periode ausgeführt werden muss. Somit gleicht die relative Deadline der Periode ($dead_i = per_i$) (vgl. Abbildung 2.3 auf Seite 11).

Durch den Einsatz eines offline Schedulers können die Randbedingungen an den zu findenden Schedule weiter verfeinert werden, indem die Tasks in weitere, nicht disjunkte, Mengen unterteilt werden:

- $R \subseteq T$ ist die Menge Tasks mit speziellen Release- und Deadline-Zeiten. Sollen die Jobs, von der Standarddefinition abweichende Release- und Deadline-Zeiten bekommen, so werden sie dieser Untermenge zugeordnet.
- $A \subseteq T$ ist die Menge der Phasen-ausgerichteten (aligned) Tasks. Diese Tasks müssen so gescheduled werden, dass die Phase, d. h. der Abstand zum Beginn der eigenen Periode, immer konstant ist. Der Scheduler legt somit die Werte der Phasen fest. Die Phasen-Ausrichtung kann für bestimmte Aufgaben des Tasks entscheidend sein, z. B. für die zyklische Abtastung.
- $F \subseteq A$ heißt die Menge der Phasen-fixierten (fixed) Tasks. Diese Tasks schränken die Entscheidung des Schedulers weiter durch die explizite Vorgabe der Phase ein (die Einschränkung der Phasen-Ausrichtung gilt weiterhin, da $F \subseteq A$). Die Fixierung der Phase ist insbesondere für die Modifikation eines bestehenden Schedules relevant, da dort bestimmte Tasks ihre Phase nicht mehr verändern dürfen.

Eine detaillierte Beschreibung dieser Mengen in deren Auswirkung auf die modellierte Problemstellung ist im nächsten Abschnitt zu finden.

6.3.2. Aktivitäten der offline Phase

Das Ziel der offline Phase ist für eine Menge von Tasks T mit $T_i = (w_i, per_i, flex_i)$ und weiteren optionalen Randbedingungen, z. B. einer Reihenfolgebeziehung unter Tasks, eine Scheduling-Tabelle über die sogenannte Hyperperiode zu bilden. Die Hyperperiode bezeichnet eine Periode, deren Länge das kgV der Perioden der einzelnen Tasks per_i ist. Bei der zyklischen Ausführung der Taskmenge T wiederholen sich die Ausführungen einzelner Jobs in jeder Hyperperiode. Der Problematik einer eventuell zu langen Hyperperiode kann durch die Wahl einer harmonischen Periodendauer für die einzelnen Tasks entgegengewirkt werden. Die Scheduling-Tabelle ordnet die einzelnen Jobs dem jeweiligen Grundzyklus innerhalb einer Hyperperiode zu. Gleichzeitig wird eine gültige Reihenfolge der Instanzen sichergestellt, falls eine Reihenfolgebeziehung spezifiziert wurde. Zusätzlich kann die optimale Ausführungsdauer für adaptive Tasks mit $flex_i = 1$ berechnet werden.

Die Anzahl der Grundzyklen in der Hyperperiode wird im Folgenden als $hyperperiod = kgV(per_1, \ldots, per_{|T|})$ bezeichnet und die Länge eines Grundzyklus des Systems als *cyctime*.

Das nicht unterbrechende Scheduling-Problem für periodische Tasksets mit bekannten Release-Zeiten ist NP-hart (d. h. mit existierenden Algorithmen nicht effizient lösbar) [JSM91]. Die vorgestellte Problemstellung weicht von dem zitierten Scheduling-Problem wegen der Annahme eines festen Grundzyklus etwas ab. Der Beweis der NP-Härte kann aber durch eine polynomielle Reduktion auf die Entscheidungsvariante des Behälterproblems (Bin Packing Problem) geführt werden. Dabei werden die Größe eines Behälters auf die Länge des Grundzyklus und die einzelnen Objekte und deren Gewichte auf die Tasks bzw. deren Ausführungsdauer abgebildet. Die Periode jedes einzelnen Tasks gleicht der Anzahl der Behälter. Da die Periode aller Tasks nun gleich ist, entspricht die Hyperperiode der Anzahl der zu füllenden Behälter. Das Scheduling-Verfahren muss nun jeden Grundzyklus mit den Tasks „füllen“, was der Zuordnung der Objekte zu den Behältern entspricht.

Die Berücksichtigung zusätzlicher Randbedingungen, wie z. B. der von der Periode abweichender Deadline- oder Reihenfolgebeziehungen unter den Tasks, erhöhen die Komplexität des Problems zusätzlich. Daher wird ein manuelles Scheduling sogar für kleinere Probleminstanzen unübersichtlich und ein automatisiertes Verfahren für die optimale Lösung des Problems notwendig. Dieses wird mit Mitteln der gemischt-ganzzahligen Optimierung umgesetzt.

Die Aktivitäten der offline Phase lassen sich wie folgt zusammenfassen:

- Aufstellung des Optimierungsproblems und dessen Lösung,
- Sicherstellung der Reihenfolge der Jobs innerhalb der einzelnen Grundzyklen und
- Dekomprimierung der Jobdauer mithilfe des elastischen Modells, bis die gesamte Slackzeit innerhalb des Grundzyklus verbraucht ist.

Bis auf den ersten Schritt sind alle Aktivitäten in polynomieller Zeit lösbar. Das Optimierungsproblem des ersten Schritts ist für Probleminstanzen mit $|T| \leq 500$ praktisch lösbar. Die Gründe für die Nutzung einer Heuristik (d. h. eines suboptimalen, aber effizienten Lösungsverfahrens) für den letzten Schritt sowie deren Aufbau werden am Ende des Abschnittes detailliert erläutert.

Formulierung des Schedulingproblems als ILP

Bei der Formulierung eines Optimierungsproblems ist die Frage nach den Entscheidungsvariablen und insbesondere nach deren Anzahl entscheidend. Die praktische Lösbarkeit eines Problems hängt maßgeblich von der Zahl dieser Variablen ab.

Ein naiver Weg ist das Abbilden einzelner diskreter Zeiteinheiten, z. B. in Millisekunden oder Sekunden, auf boolesche Entscheidungsvariablen, die genau dann wahr sind, wenn die Ausführung eines Jobs zu dem genauen Zeitpunkt passieren soll. Dieser aus der Literatur [Art12] bekannte Ansatz kann für die Abbildung komplexer Abhängigkeiten zwischen den Tasks, wie z. B. das Warten auf eine gemeinsam genutzte Ressource, benutzt werden. Der Nachteil des Ansatzes ist eine große Anzahl der Entscheidungsvariablen. Schließlich muss jeder Zeitpunkt der Hyperperiode in der vorgegebenen Auflösung abgebildet werden. Somit hängt die Anzahl der Entscheidungsvariablen von der tatsächlichen Dauer des Zyklus bzw. deren Auflösung ab.

Ein alternativer Ansatz, um diesem Problem entgegenzuwirken, ist die Definition sogenannter Slots, die mehrere Zeiteinheiten zusammenfassen und auf Entscheidungsvariablen abgebildet werden. Während die Berechnung der Slots für Problemstellungen mit Ressourcenzugriff nichttrivial ist [Foh12], lassen sich die Slots für die vorliegende Problemstellung sehr einfach definieren. In einem zyklischen System ohne Nebenläufigkeit und einem während der Zyklusdauer konstanten Prozessabbild sind keine Ressourcenkonflikte möglich. Aus diesem Grund muss kein Job auf ein Ereignis warten. Die Abwesenheit der Notwendigkeit der Synchronisation innerhalb des Zyklus reduziert das Problem der Ausführungsplanung der einzelnen Jobs auf ein einfacheres Reihenfolgeproblem. Die Lösung des Problems ist eine Ausführungsreihenfolge der Jobs innerhalb eines Zyklus. Die Konsequenz ist die Reduktion der Anzahl der möglichen Slots innerhalb eines Zyklus von der Anzahl der Zeiteinheiten auf die Anzahl der Tasks, d. h. $|T|$. Das ist durch die Tatsache zu begründen, dass pro Zyklus höchstens eine Instanz jedes Tasks ausgeführt werden kann.

Diese Vereinfachung respektiert die RA-Algorithmen aus Abschnitt 6.2, die durchaus bestimmte POUs mehrfach ablaufen lassen können. Denn bei der Ausführungsplanung der Komponenten ist der interne Aufbau der Komponente opak und es gelten für das Scheduling die Grundsätze der IEC 61131-3 inklusive einer Obergrenze für die Anzahl der Komponentenausführungen pro Zyklus.

Ein weiterer Schritt in Richtung der Reduktion der Problemgröße ist die Reduktion der Anzahl der Slots auf eins pro Grundzyklus. Somit wird nur die Zuordnung des Jobs zu dem Grundzyklus berechnet. Die eventuellen Reihenfolgebeziehungen unter den Jobs werden daher nicht mehr vom ILP, sondern von einem Postprocessing-Schritt berücksichtigt, der eine legale Reihenfolge der Jobs innerhalb des Grundzyklus herstellt.

Mit diesen Vorüberlegungen wird eine Menge der Grundzyklen $C = \{1, \ldots, hyperperiod\}$ (cycles) formuliert, in denen Jobs gescheduled werden können. Das zu lösende Optimierungsproblem wird über der Menge boolescher Entscheidungsvariablen $x_{c,t} \in \{0, 1\}\ \forall\ c \in C, t \in T$ definiert (bei der Modellierung wird der boolesche Ausdruck TRUE und 1 synonym verwendet). Es gilt:

$$x_{c,t} = 1 \Leftrightarrow \text{Instanz des Tasks } t \text{ wird im Zyklus } c \text{ ausgeführt.}$$

Zunächst wird eine einfache Gütefunktion bzw. deren Minimum genutzt:

$$min\ J_1 = \sum_{t=1}^{|T|} \sum_{c=1}^{|C|} c \cdot x_{c,t} \tag{6.1}$$

das unter den folgenden Nebenbedingungen bestimmt werden soll:

$$\sum_{c=1}^{|C|} x_{c,t} = \frac{hyperperiod}{per_t} \qquad \forall \quad 1 \leq t \leq |T| \tag{6.2}$$

$$\sum_{\substack{d=1 \\ (c-1)\cdot per_t+1 \leq d \leq c \cdot per_t}}^{|C|} x_{d,t} \leq 1 \qquad \forall \quad 1 \leq t \leq |T|,\ 1 \leq c \leq |C| \tag{6.3}$$

$$\sum_{t=1}^{|T|} x_{c,t} \cdot w_t \leq cyctime \qquad \forall \quad 1 \leq c \leq |C|. \tag{6.4}$$

Die Gütefunktion aus der Gleichung 6.1 definiert die Kosten als Summe der aufeinander folgenden Nummern der Grundzyklen, in denen Jobs ausgeführt werden. Somit verfolgt der Solver das Ziel die Jobs möglichst früh und somit kompakt zu schedulen.

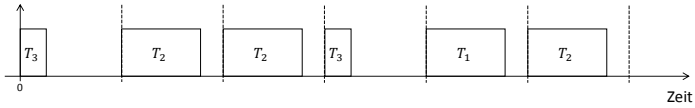


Abbildung 6.23.: Eine Hyperperiode des Laufzeitsystems mit den zugeteilten Jobs.

Das Nebenbedingungssystem 6.2 erzwingt, dass jeder Task t genau $\frac{hyperperiod}{per_t}$ mal pro Hyperperiode aufgerufen wird. Der Wert des Bruchs ist ganzzahlig, da die Hyperperiode als kgV der Perioden einzelner Tasks gebildet wurde.

Das zweite Nebenbedingungssystem 6.3 schreibt vor, dass eine Taskinstanz nicht häufiger als einmal pro Periode des Tasks ausgeführt wird. Zusammen mit Bedingungen 6.2 wird somit ein Job genau einmal pro Periode des Tasks gescheduled. Die Phase der einzelnen Jobs innerhalb der Periode ist dabei variierbar.

Das letzte Nebenbedingungssystem 6.4 sichert die Einhaltung der Länge des Grundzyklus zu. Die Summe der WCET der Tasks, die einem Grundzyklus zugeordnet sind, darf somit die Konstante *cyctime* nicht überschreiten.

Das Optimierungsproblem besitzt $|C| \cdot |T|$ boolesche Entscheidungsvariablen und $|C|^2 \cdot |T|^2$ viele Nebenbedingungen.

Zunächst wird die Problemformulierung und deren Lösung anhand eines einfachen Beispieles illustriert. Sei die folgende Menge an Tasks gegeben: $T_1 = (90\,ms, 6, 1)$, $T_2 = (90\,ms, 2, 1)$ und $T_3 = (30\,ms, 3, 1)$. Die Dauer des Grundzyklus des Systems sei 100 ms. Die Länge der Hyperperiode ist $kgV(\{6, 2, 3\}) = 6$. Die Formulierung des Problems als General Algebraic Modeling System (GAMS) Instanz ist im Anhang A zu finden. Die berechnete Lösung der Probleminstanz ist in Abbildung 6.23 dargestellt.

In Abbildung 6.23 sind unterschiedliche Phasen der Jobs des Tasks T_2 zu erkennen. Die Jobs des Tasks werden abwechselnd im ersten bzw. zweiten Zyklus des Task-Zyklus von zwei Grundzyklen ausgeführt. Diese Schwankung kann abhängig von der Problemstellung unerwünscht sein und lässt sich auf zwei Arten lösen. Die erste Lösung sind die vorgeschalteten bzw. nachgeschalteten „Puffer"-Bausteine, die das Prozessabbild zu den gleichen Zeitpunkten in jeder Phase des Tasks speichern. Die zweite Lösung ist die Zuordnung der Tasks zu der Menge A der phasenausgerichteten Tasks. Beide Lösungen haben ihre Nachteile. Die erste Alternative fügt die Puffer-Bausteine und die Reihenfolgeabhängigkeiten hinzu. Die zweite Alternative schränkt den Scheduler ein. So ist das obere Problem unter den Einschränkungen der Phasenausrichtung des Tasks T_2 nicht lösbar.

Die vorgestellte einfache Formulierung reicht für viele Scheduling-Probleme bereits aus. Sie kann jedoch weiter verfeinert werden, um weitere Randbedingungen an die Tasks berücksichtigen zu können. Die im Beispiel angesprochene Phasenausrichtung ist eine solche Randbedingung. Die Erweiterungen der vorgestellten Problemdefinition werden in folgenden Abschnitten erläutert.

Reihenfolgebeziehungen unter Tasks Trotz der virtuellen Unabhängigkeit der selbstständigen Komponenten (SKs können auch auf unterscheiden Geräten verteilt sein), kann die Optimierung deren Ausführungsreihenfolge auf einem Gerät für schnellere Reaktionszeiten der Anwendung sorgen und somit zu einem zusätzlichen Optimierungsziel erklärt werden (im worst case ist es aber möglich, dass SKs auf unterschiedlichen Geräten in

ungünstigster Reihenfolge ablaufen).

Diese Beziehung kann als eine Matrix $P \in \{0,1\}^{|T|\times|T|}$ ausgedrückt werden, sodass:

$$P_{i,j} = 1 \Leftrightarrow t_i \text{ ist Vorgänger von } t_j.$$

Die Reihenfolgebeziehung kann zwischen Tasks bestehen. Es wird zusätzlich angenommen, dass nur Tasks mit gleicher Periode in einer solchen Beziehung stehen können, d. h. $P_{i,j} = 1 \Rightarrow per_i = per_j$. Das Optimierungsproblem kann durch folgende Nebenbedingungen durch die Matrix P ergänzt werden:

$$\sum_{\substack{d=1 \\ (c-1)\cdot per_i+1 \leq d \leq c\cdot per_i}}^{|C|} d \cdot x_{d,i} - d \cdot x_{d,j} \leq 0 \qquad \forall\, 1 \leq c \leq |C|, P_{i,j} = 1. \tag{6.5}$$

Die Bedingung stellt sicher, dass in jeder Periode per_t der Aufruf von t_i in einem früheren oder gleichen Grundzyklus wie der Aufruf von t_j stattfindet.

Phasenausrichtung und fixierte Phase Die Problemformulierung lässt dem Scheduler eine freie Wahl über die Phase des Jobs innerhalb der Periode des Tasks. Somit können die Abstände zwischen den Ausführungen einzelner Jobs variieren, was für viele Anwendungen, z. B. im Bereich der Regelungstechnik, unerwünscht ist. Die Freiheit des Schedulers in Bezug auf die Phase wird in zwei Stufen eingeschränkt.

Als erstes wird eine Untermenge $A \subseteq T$ (aligned) der Tasks mit Phasenausrichtung gebildet. Diese Tasks müssen in jeder Periode eine fixierte Phase besitzen, die allerdings vom Scheduler bestimmt werden kann. Eine weitere Einschränkung ist die Untermenge $F \subseteq A$ (fixed) der Tasks mit fixem Phasenparameter $phase_f \in [0, per_f - 1]$.

Beide Mengen werden durch folgende Nebenbedingungen mit dem Optimierungsproblem verknüpft:

$$x_{c,a} = x_{c+per_a,a} \qquad \forall \quad 1 \leq c \leq |C| - per_a, 1 \leq a \leq |A| \tag{6.6}$$

$$x_{1+phase_f,f} = 1 \qquad \forall \quad 1 \leq f \leq |F|. \tag{6.7}$$

Die Gleichungen 6.6 erzwingen die gleiche Phase innerhalb der Periode des Tasks. Die Gleichungen 6.7 setzen die Phase der ersten Periode fest. Durch die Tatsache, dass $F \subseteq A$ ist, propagieren die Gleichungen 6.6 diese Phase über weitere Perioden hinweg.

Release-Zeiten und Deadlines Eine der festen Phasenverschiebung ähnliche Einschränkung kann mit der Angabe der Release-Zeiten und Deadlines erreicht werden. Bei den zyklischen Tasks wurde bisher davon ausgegangen, dass die Ausführung zwischen dem ersten und dem letzten Grundzyklus innerhalb der Periode erfolgen kann. Diese Annahme kann man für bestimmte Tasks lockern und explizite Release-Zeit $rel_t \in [1, per_t]$ und $dead_t \in [1, per_t]$ mit $rel_t \leq dead_t$ angeben. Dazu wird eine Untermenge der Tasks $R \subseteq T$

gebildet, für die folgende Nebenbedingungen gelten sollen:

$$\sum_{\substack{d=1 \\ (c-1)\cdot per_r+1 \leq d \leq c\cdot per_r \\ d \leq (c-1)\cdot per_r + rel_r}}^{|C|} x_{d,r} = 0 \qquad \forall \quad 1 \leq c \leq |C|, 1 \leq r \leq |r| \tag{6.8}$$

$$\sum_{\substack{d=1 \\ (c-1)\cdot per_r+1 \leq d \leq c\cdot per_r \\ d \geq (c-1)\cdot per_r + 1 + dead_r}}^{|C|} x_{d,r} = 0 \qquad \forall \quad 1 \leq c \leq |C|, 1 \leq r \leq |r|. \tag{6.9}$$

Die Gleichungen 6.8 stellen sicher, dass in jeder Periode des Tasks r die Ausführung nicht vor dem Grundzyklus rel_r bezüglich des Periodenanfangs beginnt. Bedingungen 6.9 arbeiten auf eine ähnliche Weise: sie erzwingen, dass keine Ausführung nach dem Grundzyklus $dead_r$ in Bezug auf die Periode des Tasks stattfindet.

Im Gegensatz zu der Phasenausrichtung (Gleichungen 6.6 bzw. 6.7), bieten die Release-Zeiten und Deadlines mehr Flexibilität: Zum einen sind die Bedingungen sowohl für Phasen ausgerichtete Tasks, als auch für solche ohne Phasenausrichtung nutzbar. Zum anderen bieten die Gleichungen die Möglichkeit „Fenster“ zu definieren, in denen die Ausführung stattfinden soll. Somit ist die Gleichung 6.7 nur ein Sonderfall der obigen Gleichungen für ein Task r mit $rel_r = dead_r$, d. h. der vorgeschriebenen Ausführung in dem bestimmten Grundzyklus bzw. mit einer festen Phase.

Komplexe Gütefunktionen: Gleiche Auslastung der Grundzyklen Die Gütefunktion 6.1 hatte das Ziel, die einzelnen Jobs möglichst kompakt auf die Grundzyklen zu verteilen. Damit werden die Jobs so früh wie möglich vom Scheduler ausgeführt und die Grundzyklen eventuell ungleich ausgelastet. Dieses Verhalten kann für manche Szenarien von Vorteil sein, denn es werden eventuell große zusammenhängende freie Zeitabschnitte für die Ausführung sporadischer Tasks entstehen. Für andere Szenarien, gerade bei der Ausnutzung der Slackzeit, kommt es jedoch auf eine möglichst gleichmäßige Auslastung der Grundzyklen an. Eine solche Nutzung kann durch die Einführung einer quadratischen Gütefunktion zusammen mit einem Minimierungsproblem erreicht werden:

$$min\ J_2 = \sum_{c=1}^{|C|} \left((\sum_{t=1}^{|T|} x_{c,t} \cdot w_t) - \frac{\sum_{t=1}^{|T|} \sum_{d=1}^{|C|} (x_{d,t} \cdot w_t)}{|C|} \right)^2 . \tag{6.10}$$

Der quadratische Term ist die Abweichung der Zeitbelegung des aktuellen Grundzyklus c von der arithmetisch durchschnittlichen Belegung aller Grundzyklen. Die quadratische Abweichung wird über alle Grundzyklen aufsummiert. Das Minimum der Funktion erreicht den Wert null, wenn alle Zyklen exakt mit der durchschnittlichen Belegung ausgefüllt werden. Die Klasse des Optimierungsproblems verschiebt sich von (M)ILP zu Mixed Integer Quadratic Program (MIQP).

Komplexe Gütefunktionen: Fehlerfunktionen Eine weitere Erweiterung des Problems ist die Berücksichtigung einer Fehlerfunktion für einzelne Tasks, wie solche, die in Imprecise Computation Model (vgl. Abschnitt 5.1.5) eingeführt wurden. Dabei wird die zu der WCET zusätzlich kommende Ausführungszeit eines Tasks mit einem Fehler bewertet, der negativ mit dieser Zeit korreliert und nur von dieser Zeit abhängt. Das Ziel ist die Minimierung des

Gesamtfehlers des Schedules. Die Menge der Entscheidungsvariablen wird um Variablen $add_{c,t} \geq 0 \ \forall \ c \in C, t \in T$ vergrößert. Die Variablen beinhalten den Wert der über die WCET hinausragende Ausführungszeit, die dem Job t im Grundzyklus c zugeordnet wird. Die Gleichung 6.4 wird durch die neuen Variablen ergänzt:

$$\sum_{t=1}^{|T|} x_{c,t} \cdot w_t + \sum_{t=1}^{|T|} x_{c,t} \cdot add_{c,t} \leq cyctime \qquad \forall \quad 1 \leq c \leq |C|. \tag{6.11}$$

Somit wird sichergestellt, dass die zusätzliche Zeit die Dauer des Grundzyklus nicht übersteigt. Als letztes kann die neue Gütefunktion des Optimierungsproblems aufgestellt werden, die die exponentiell fallende Fehlerfunktion der einzelnen Jobs beinhaltet:

$$min \ J_3 = \sum_{t=1}^{|T|} \sum_{c=1}^{|C|} x_{t,c} \cdot pri_t \cdot e^{-add_{c,t}}. \tag{6.12}$$

Der Fehler sinkt exponentiell mit der Zunahme der zusätzlichen Ausführungszeit, die zusätzliche Ausführungszeit kann dabei beliebig lang sein (es gilt die Annahme von sich monoton-verbessernden Jobs). Die zusätzlich eingeführte Priorität des Tasks pri_t spielt die Rolle eines Gewichtes bei der Summierung der Fehler einzelner Tasks zum Gesamtfehler. Die Klasse des Optimierungsproblems ändert sich zu Mixed Integer Nonlinearly Constrained Program (MINLP).

Lineare und konkave Fehlerfunktionen können auf ähnliche Art und Weise berücksichtigt werden. Ebenso können obere Schranken bezüglich der zusätzlichen Ausführungszeit $add_{c,t}$ eingeführt werden, um neben den monotonen Funktionen auch die 0/1-Ausführung und die Mehrvarianten-Methode erfassen zu können.

Praktische Lösbarkeit des Schedulingproblems Das vorgestellte ILP-Problem wurde mithilfe der algebraischen Modellierungssprache GAMS modelliert. Eine beispielhafte Probleminstanz als GAMS-Datei ist im Anhang A zu finden. Ein Vorteil der Nutzung einer Modellierungssprache ist die Möglichkeit der Verwendung unterschiedlicher Solver, die die Probleminstanzen direkt vom GAMS-Framework erhalten. Die Notwendigkeit der Solver-Spezifischen Problemformulierung entfällt somit. Das Problem wurde für Testzwecke mit zwei Solvern, Gurobi und IBM CPLEX, gelöst. Ein weiterer Vorteil der GAMS-Formulierung ist die Möglichkeit der Nutzung der NEOS Infrastruktur [CMM98, Dol01, GM97] für Optimierungsprobleme. Der NEOS Server stellt die Rechenkapazität und die Lizenzen für GAMS und Solver zur Lösung der Optimierungsprobleme kostenlos zur Verfügung. Eine Probleminstanz kann einfach über eine Hypertext Transfer Protocol (HTTP) oder Extensible Markup Language Remote Procedure Call (XML-RPC) Schnittstelle hochgeladen werden. Die Lösung wird durch den Server, sobald verfügbar, asynchron zur Verfügung gestellt. Die Nutzung der NEOS Infrastruktur ist eine elegante Möglichkeit, Schedules in der Cloud (Schedule-as-a-Service) berechnen zu lassen (mehr zu diesem Thema ist in Abschnitt 6.3.3 zu finden) um die Problematik der Softwarelizenzierung und der Ressourcenverfügbarkeit zu umgehen.

Im Folgenden wird die Praktikabilität des vorgestellten Ansatzes, d. h. die Problemformulierung und deren Lösbarkeit durch den NEOS Server untersucht. Als Ziel für die „Praktikabilität“ wurde eine Obergrenze von wenigen Minuten (1000 Sekunden) Rechenzeit auf dem NEOS Server gesetzt.

Für die Untersuchung wurden zufällige Probleminstanzen mit einer variierenden Anzahl der Tasks generiert und an den NEOS Server übermittelt. Die Periode der Tasks per_t wurde zufällig aus der Wertemenge $\{2, 4, 8, 16, 32\}$ ausgewählt. Die WCET wurde zufällig aus dem Intervall $[10, 100]$ ausgewählt. Alle Tasks erhielten eine Phasenausrichtung und eine zufällige Deadline aus dem Bereich $[1, per_t]$ für jeden Task t. Die Dauer des Grundzyklus wurde mit $|T| \cdot 15$ angesetzt, wobei T die Menge der Tasks bezeichnet.

Das Lösen der zufälligen Instanzen mit 500 Tasks und der Gütefunktion 6.1 mit Gurobi war bei 50 Versuchen immer unter 130 Sekunden möglich. Der Mittelwert lag bei 95.6 Sekunden. Somit kann man von einer praktikablen Lösung der Probleme bis zu dieser Größe ausgehen.

Der Einsatz der Gütefunktion 6.10 lässt Zweifel an der Praktikabilität der Lösbarkeit aufkommen. Das Auswerten der 50 Zufallsinstanzen mit nur 9 Tasks ergab folgende Resultate: 6 Instanzen konnten nicht innerhalb der 1000 Sekunden gelöst werden. Bei den restlichen 44 Instanzen lag der Mittelwert bei 81.8 Sekunden. So ist festzustellen, dass die Streuung viel größer ist, denn die maximale Lösungsdauer belief sich auf ca. 950 Sekunden, die minimale auf nur ca. eine Sekunde. Aus diesen Gründen wird das vorgestellte Optimierungsproblem mit der Gütefunktion 6.10 nicht als praktisch lösbar bezeichnet.

Bei der dritten Gütefunktion 6.12 erfolgt das Lösen des Problems mit dem Solver namens BARON [TS05]. Die Auswertung der 50 Instanzen mit 150 zufälligen Tasks und der Gewichtung pri_t von 1 ergab einen Ausreißer, der innerhalb der 1000 Sekunden nicht gelöst wurde. Bei den restlichen Instanzen beträgt der Mittelwert 159.7 Sekunden. Die Lösbarkeit dieser Problemklasse wird daher als semi-praktikabel bezeichnet.

Für die folgende Evaluation wurde aus diesen Gründen der Weg der heuristischen Ergänzung der unter der Verwendung der „einfachen“ Gütefunktion 6.1 erstellten Problemlösung gewählt. Diese Lösung stellt bereits einen validen Schedule bezüglich der Einhaltung der Echtzeitschranken dar. Es werden allerdings nur die WCET-Laufzeiten der Jobs berücksichtigt, sodass noch „ungenutzte“ Zeitabschnitte innerhalb der Grundzyklen existieren. Abgesehen von der Reihenfolgezuordnung der einzelnen Jobs, muss somit eine zusätzliche Strategie zur Ausnutzung dieser Zeitabschnitte definiert werden, die eine suboptimale Lösung in Bezug auf das Ziel der globalen Fehlerminimierung ergibt. Die möglichen Heuristiken werden im nächsten Abschnitt vorgestellt.

Heuristische Berechnung der zusätzlichen Ausführungszeiten einzelner Komponenten für das offline Schedule

Nachdem die Zuteilung von Jobs unter der Berücksichtigung der Gütefunktion 6.1 zu Grundzyklen geschehen ist, sind die einzelnen Grundzyklen möglicherweise nicht vollständig ausgefüllt. Diese geplante Slackzeit kann nun auf die Jobs aufgeteilt werden. Dafür stehen mehrere Strategien zur Verfügung.

Die einfachste Möglichkeit ist die Nutzung einer Greedy-Strategie, bei der der Job des Tasks i mit der höchsten Priorität und dem Parameter $flex_i = 1$ die gesamte externe Slackzeit des Grundzyklus zugewiesen bekommt. Die restlichen Instanzen bekommen nur die WCET zugewiesen. Durch diese Benachteiligung der anderen Jobs ist dieser einfache Ansatz im Allgemeinen nicht geeignet.

Eine weitere Strategie aus dem Bereich des Imprecise Computation Modells (vgl. Abschnitt 5.1.5) würde die unterschiedlichen Nutzenfunktionen einzelner Komponenten berücksichtigen und ein Schedule erzeugen, welcher den gesamten Nutzen für jeden Grundzy-

klus maximiert. Das problematische an diesem Ansatz ist die Notwendigkeit der Definition der Nutzenfunktionen, die nicht-linear sein müssen (ähnlich zu der Gleichung 6.12), damit diese Strategie nicht zu einer Greedy-Strategie degradiert. Denn wäre der Zeitgradient der Nutzenfunktionen immer konstant und positiv, so würde die Komponente mit dem größten Gradient die gesamte Slackzeit zugeordnet bekommen.

Die dritte Möglichkeit ist die Anwendung des elastischen Modells (vgl. Abschnitt 5.2.1) das die einzelnen Jobs als Federn modelliert, deren Federkonstante proportional zu der Priorität des Jobs ist. Dieses Modell sieht vor, dass flexible Komponenten i, also solche mit $flex_i = 1$, sich ihrer Priorität entsprechend ausdehnen und die gesamte Slackzeit des Grundzyklus ausnutzen.

Das Modell kann nicht nur für die Kompression der Tasks verwendet werden, sondern auch für die Dekompression. Im Gegensatz zu dem elastischen Modell ist die Länge der Federn nicht der Utilization-Faktor des Tasks, sondern die tatsächlich zugeteilte Ausführungsdauer Δ^{curr}. Für die Berechnung der dekomprimierten Ausführungszeiten müssen nur noch die Prioritäten einzelner Tasks bzw. Jobs pri_i in Federkonstanten k_i umgewandelt werden. Diese Konstanten sind antiproportional zu der Priorität. Denn je höher der Wert der Federkonstante, desto weniger zusätzlicher Zeit wird dem Job vom Scheduler zugeteilt. Somit können die Federkonstanten wie folgt berechnet werden:

$$k_i = \frac{\sum_{j=1}^{|T|} pri_j}{pri_i}.$$

Die Ausführungsdauer Δ_i^{curr} des Jobs i ist somit wie folgt berechenbar:

$$\Delta_i^{curr} = w_i + \left(cyctime - \sum_{j=1}^{|T|} w_j \right) \frac{K_p}{k_i}, \tag{6.13}$$

mit $K_p = \frac{1}{\sum_{j=1}^{|T|} \frac{1}{k_j}}$. Die Indizes der Jobs beziehen sich auf den jeweiligen Grundzyklus innerhalb der Hyperperiode. Da der Term $cyctime - \sum_{j=1}^{|T|} w_j$ immer nichtnegativ ist, ist die Bedingung $\Delta_i^{curr} \geq w_i$ garantiert.

Eine Erweiterung des Elastic Models sieht eine untere Längenbeschränkung der Federlänge vor (in der graphischen Anschauung als Klammern dargestellt, vgl. Abbildung 5.2). In einer ähnlicher Weise kann auch die Dekompression nach oben durch den Parameter $upper_i$ beschränkt werden. Dies ist z. B. bei der 0/1 Ausführung oder der Mehrversionen-Methode der Fall (vgl. Abbildung 6.17). In diesem Fall ist der Job nicht in der Lage beliebig viel Rechenzeit sinnvoll zu verbrauchen. Bei Tasks mit $flex_i = 0$ stellt die WCET eine solche obere Schranke dar, also $upper_i = w_i$. Die oberen Schranken können auf die gleiche Weise wie die unteren behandelt werden: Die Gleichung 6.13 werden iterativ gelöst; übersteigt die Dauer Δ_i^{curr} die obere Schranke, wird der Job aus dem Grundzyklus „entfernt“ und dessen Dauer um w_i verkürzt. Das Lösen wird so lange wiederholt bis für alle Jobs $\Delta_i^{curr} \leq upper_i$ gilt. Metaphorisch kann die obere Schranke $upper_i$ als die Länge eines Seils interpretiert werden, das an den Enden der Feder befestigt ist (vgl. Abbildung 5.2).

6.3.3. Aktivitäten der online Phase

Der ausgewählte Ansatz des im Voraus berechneten offline Schedules hat den Nachteil der eingeschränkten Flexibilität (vgl. Abschnitt 2.1.2). Um diesen Nachteil zu reduzieren, umfasst der entwickelte Scheduling-Algorithmus neben den diskutierten Vorgängen der offline

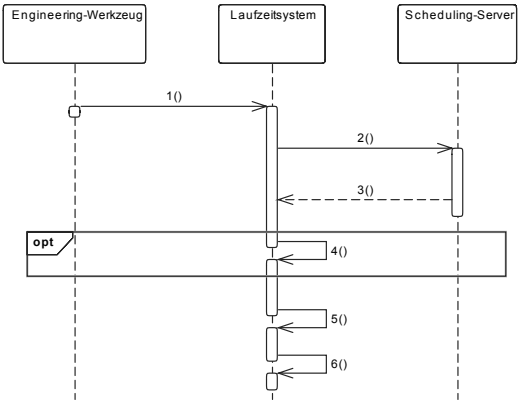


Abbildung 6.24.: Ablauf der „mode change“-Prozedur.

Phase, auch eine Reihe Laufzeit-Aktivitäten. Dazu gehört eine Anzahl bereits vorgestellter Techniken, die im Kontext des Gesamtsystems eingesetzt werden.

Acceptance Tests und Umstrukturierung des offline Schedules zur Laufzeit

Im Falle des starren offline Schedules für zyklische Systeme können Tasks nicht ohne eine Neuberechnung der Scheduling-Tabelle hinzugefügt werden. Dieser Grundsatz muss durch die ausgewählten System- und Scheduling-Paradigmen befolgt werden. Trotzdem muss das Laufzeitsystem für die Umstrukturierung der Scheduling-Tabelle nicht zwingend angehalten werden. Denn obwohl die Berechnung, d. h. die Lösung des Optimierungsproblems, gewisse Zeit in Anspruch nehmen kann, kann die Umstrukturierung an sich, d. h. das Umschalten zwischen zwei Scheduling-Tabellen in kürzester Zeit erfolgen. In der Literatur heißt dieser Ansatz „mode change“ [Foh93, BF11].

Die Berechnung des neuen Schedules erfolgt in der Zeit, in der der alte Schedule noch aktiv ist. In Abschnitt 6.3.2 wurde demonstriert, dass die Optimierungsprobleme auch „in der Cloud“ und somit nicht von dem in ihren Ressourcen eingeschränkten Laufzeitsystem gelöst werden können. Der Ablauf der Prozedur der Umstrukturierung läuft wie folgt ab (in Abbildung 6.24 als Sequenzdiagramm dargestellt):

1. Anstoßen der Änderung der aktuellen Scheduling-Tabelle,
2. Formulierung des neuen Tasksets und Übermittlung der Probleminstanz,
3. Lösen der Probleminstanz und Übermittlung der Lösung,
4. (*optional*) Verifikation der Lösung (Abbruch falls Lösung ungültig),
5. Laden der Ressourcen, z. B. des ausführbaren Codes der SK und die Erstellung des neuen Schedules aus der Problemlösung und
6. Umschalten des Schedules – „mode change“.

Im ersten Schritt werden dem Laufzeitsystem die Änderungswünsche mitgeteilt. Typischerweise kommen sie aus dem Engineering, z. B. bei dem Anlegen einer neuen SK. Das ist aber nicht das einzig denkbare Szenario. So kann die Änderungsanfrage von einem anderen System kommen, z. B. durch die Instanziierung eines Agenten als SK, oder durch das Laufzeitsystem an sich, z. B. wenn Abhängigkeiten zwischen den SKs festgestellt wurden und ein neuer Schedule diese Informationen berücksichtigen soll.

Der zweite Schritt kombiniert die Informationen aus der aktuellen Menge der Tasks mit der beantragten Änderung. Dabei müssen taskabhängige Parameter berücksichtigt werden. Ein Beispiel ist die Phase der phasenausgerichteten Tasks. Während bei der erstmaligen Erstellung eines Scheduler die Phase eines Tasks beliebig sein kann, muss bei Veränderung des Schedules sichergestellt werden, dass sich die Phase nicht bzw. nur in einem erlaubten Maße verändert (vgl. die Eigenschaften der Scheduling-Facette einer selbständigen Komponente in Abschnitt 6.1.5). Die Erstellung der neuen Anfrage findet im Zeitslot für die nicht echtzeitfähige Kommunikation statt.

Im dritten Schritt wird das Problem vom Scheduling-Server gelöst und an das Laufzeitsystem übermittelt. Als Beispiel für den Scheduling-Server kann der NEOS-Server betrachtet werden, der in Abschnitt 6.3.2 eingesetzt wurde.

Im nächsten Schritt wird die Lösung auf ihre Korrektheit überprüft. Wie bei jedem NP-hartem Problem kann die Korrektheit einer Lösung effizient überprüft werden, somit auch vom eingeschränkten Laufzeitsystem. Dieser Schritt kann übersprungen werden, falls Vertrauen zwischen dem Scheduling-Server und dem Laufzeitsystem besteht. Im anderen Fall sichert dieser Schritt insbesondere bei der Berechnung des Schedules in der Cloud das System zusätzlich. Der Vorteil der Neuberechnung ist der implizite Acceptance Test der neuen Menge der Tasks. Falls eine Lösung zu dem Optimierungsproblem des Abschnitts 6.3.2 existiert, so kann diese auch sicher ausgeführt werden.

Im fünften Schritt werden die benötigten Ressourcen geladen, ohne ausgeführt zu werden. Beispielsweise wird beim Hinzufügen einer neuen SK deren ausführbarer Code heruntergeladen. Auch das neue Schedule wird endgültig fertiggestellt.

Die Ausführung dieser Vorbereitungsschritte geschieht in der Slackzeit. Somit können für diese Schritte keine Echtzeitschranken definiert werden. Nach dem erfolgreichen Abschluss aller Vorbereitungen kann im letzten Schritt der neue Schedule aktiviert werden. Im besten Fall erfolgt die Aktivierung durch das Ändern eines einzigen Zeigers. Dieser Ansatz wird erfolgreich bei FASA (vgl. Abschnitt 3.3.3) eingesetzt und kann auch mit Rollback-Funktionen ergänzt werden [WO14].

Ausführung der ISSCs

Die Ausführung der eingebetteten ISSCs ist ein maßgeblicher Teil des online Schedulings, da während der Ausführung Laufzeitentscheidungen getroffen werden und als Folge die eventuell anfallende Slackzeit verbraucht wird. Der Komponentenscheduler teilt einem ISSC die für seine Ausführung bestimmte Laufzeit Δ^{curr} mit. Das Chart wird ausgehend von dieser Dauer und den beinhalteten Guards ausgewertet.

Die durch eine zu frühe Terminierung einzelner Aktionen innerhalb des Charts entstehende Slackzeit kann durch die „später“ folgenden Zustände und deren Aktionen verbraucht werden. Das gleiche Verhalten kann auch auf der Ebene der einzelnen Jobs beobachtet werden. Die innerhalb eines Grundzyklus folgenden Jobs können die nicht verbrauchte Rechenzeit der Vorgänger ausnutzen. Somit ist die erwartete zusätzliche Ausführungszeit für

später folgende Jobs tendenziell höher. Dies kann durch die Änderung der Gewichte der Tasks für das offline Optimierungsproblem berücksichtigt werden und als eine Maßnahme zu der Priorisierung einzelner Tasks genutzt werden.

Ausführung sporadischer Tasks

Die verfügbare Rechenzeit kann nicht nur für RA-Algorithmen, sondern auch für die Ausführung sporadischer Tasks, d. h. azyklischer Tasks, die erst zur Laufzeit erstellt werden, eingesetzt werden. Ein Beispiel für sporadische Tasks sind Transaktionen, die die Modelle in der Laufzeitumgebung auf vorhersagbare und zugesicherte Art und Weise verändern. Die Ausführung sporadischer Tasks im Kontext der offline Scheduling-Tabelle ist wahrscheinlich die wichtigste Maßnahme, um die Nachteile der fehlenden Flexibilität der offline Ansätze zu kompensieren.

Die Beschreibungssprache ISSC sieht eine Möglichkeit vor, solche Transaktionen zu beschreiben. Ein vorhersagbares Verhalten von Transaktionen kann durch die vom Nutzer definierten History-Breakpoints erreicht werden (vgl. Abschnitt 6.2.5, insbesondere Abbildung 6.17b). Die WCET eines solchen ISSCs ergibt sich als maximale Ausführungszeit der Abschnitte zwischen den History-Breakpoints.

Diese Vereinfachung ermöglicht ein effizientes online Scheduling. Falls ein sporadischer Task zur Ausführung angemeldet wird, überprüft der Scheduler, ob mindestens einer der Grundzyklen der Scheduling-Tabelle über genügend freie Zeit für die Ausführung des gesamten Tasks in der geplanten Slackzeit verfügt. Ist das der Fall, kann die Transaktion sicher ausgeführt werden. Die Ausführungszeiten der elastischen Jobs werden für die Dauer der Ausführung der Transaktion temporär reduziert. Diese Kompression kann entweder mithilfe des Elastic Models stattfinden oder mit einem anderen Verfahren, z. B. durch einen gleichanteiligen Abzug der Rechenzeit von der Laufzeit einzelner Jobs, erreicht werden.

Ist die Ausführung einer Transaktion auf diese vereinfachte Weise nicht möglich, so muss die Änderung des offline Schedules angestoßen werden. Die daraus resultierende temporäre Scheduling-Tabelle wird dann bis zu der Terminierung des sporadischen Tasks ausgeführt. Danach kann wieder zu dem ursprünglichen Schedule gewechselt werden.

Falls weder genügend statische Slackzeit vorhanden ist, noch eine Umstrukturierung des offline Schedules möglich ist, kann die Ausführung des sporadischen Tasks nach einer best-effort Strategie erfolgen. Der Scheduler räumt dabei den einzelnen RA-Komponenten nur die garantierte Zeit w ein und überlässt die komplette übrig gebliebene Slackzeit dem sporadischen Task. Ein Use-Case, der die Ausführung einer Transaktion nach dieser Strategie beinhaltet, wird in Abschnitt 7.5 vorgestellt. Bei dieser Strategie können keine Abschätzungen über die Dauer der Ausführung des sporadischen Tasks gemacht werden, da die verfügbare Slackzeit nicht notwendigerweise für die Ausführung ausreicht.

Die Nutzung eines Ansatzes des Predictably Flexible Real-Time Scheduling (vgl. Abschnitt 5.1.6) lässt die Nutzung komplexer Techniken aus diesem Bereich im eingeführten Modell zu. Dazu gehören z. B. die Slot Shifting Verfahren, die zur optimalen Verteilung der Rechenzeit einzelner Jobs genutzt werden können.

7. Evaluierung und Anwendungsszenarien

Das eingeführte RA-Rahmenwerk wurde zwecks Evaluierung und Machbarkeitsanalyse für das Laufzeitsystem ACPLT/RTE prototypisch implementiert. Dieses Kapitel beinhaltet eine kurze Beschreibung des Prototyps und des evaluierten Hardwaresystems.

Der wichtigere Beitrag besteht allerdings in den vier vorgestellten Use-Cases. Diese belegen die Einsatzmöglichkeiten des Rahmenwerks in verschiedenen Anwendungsszenarien aus der Domäne der Prozessleittechnik.

7.1. Prototypische Implementierung

Die Prozedurbeschreibungssprache ISSC und der Komponentenscheduler wurden gemäß der beschriebenen Spezifikationen in ACPLT/RTE umgesetzt. Die dafür benötigten OV-Modelle, die UML-Klassendiagrammen entsprechen, sind im Anhang B zu finden.

Die Realisierung von ISSC folgt dem Klassendiagramm in Abbildung 6.7. Das Modell wurde leichten implementierungsspezifischen Veränderungen unterworfen: Es wurde die Möglichkeit der Ausführung der Aktionen in den Breakpoints eingeführt, die für die Berechnung der Guard-Variablen benutzt werden.

Die Implementierung des Systemschedulers beinhaltet ein Objektmodell zur Darstellung des offline Schedules, dessen Aufbau exemplarisch in Objektdiagrammen der folgenden Use-Cases dargestellt wird. Zusätzlich zu der Darstellung des Schedules, leitet der Scheduler die Klassen für Funktionsbausteine und FBNs ab und erweitert die Signatur der Aufruffunktion einer POUs um den Parameter Δ^{curr}. Die Koexistenz zwischen dem bereits vorhandenen tabellarischen Scheduling des Laufzeitsystems und dem RA-Scheduling ist in beide Richtungen vorhanden. So kann der konventionelle Scheduler RA-Anwendungen durch Polymorphe aufrufen. In diesem Fall wird nur die zugesicherte Zeit w an die RA-Anwendung übergeben. Umgekehrt kann der RA-Scheduler konventionelle POUs ohne den Parameter Δ^{curr} aufrufen.

Zusätzlich zu der Implementierung des Schedulers und des ISSCs wurde eine Engineeringumgebung auf Basis des ACPLT/csHMI (Client Side Human Machine Interface) [JE13] Engineerings für SSCs aufgebaut. Da die Klassennamen der ISSC-Implementierung den Namen der ISSC-Klassen folgen, waren die benötigten Änderungen der bestehenden Engineering-Plattform minimal. Die Engineeringumgebung wurde für die Erstellung und Dokumentation der Use-Cases verwendet.

Für die Evaluation der prototypischen Implementierung wurde ein WAGO 758-875 Industrie-PC mit einer 1 GHz Intel Celeron M CPU und 256 MB RAM eingesetzt. Das System läuft unter 32-bit 2.6 Linux mit RT-Preempt Patch. Als Laufzeitsystem wurde ACPLT/RTE mit dem Versionstand „5f7c61c“ aus dem GitHub-Repository[1] verwendet.

[1] https://github.com/acplt/rte

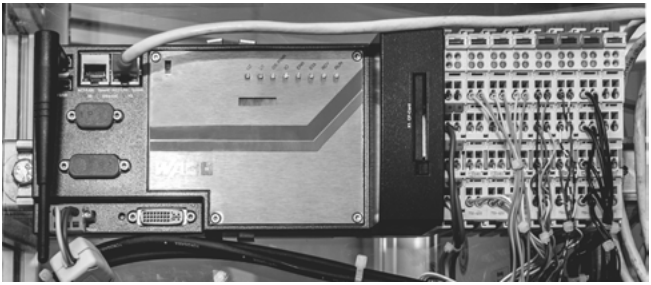


Abbildung 7.1.: WAGO 758-875 IPC eingebaut in einem Modul der Anlage M4P.AC.

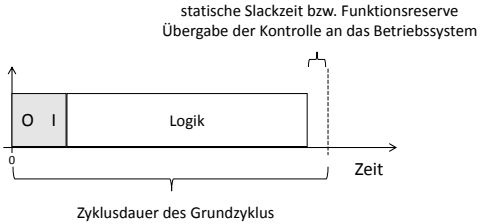


Abbildung 7.2.: Typischer Aufbau eines Grundzyklus der ACPLT/RTE Laufzeitumgebung.

Der typische Aufbau eines Grundzyklus des Laufzeitsystems ist in Abbildung 7.2 zu sehen. Im Unterschied zu dem in Abbildung 2.6 vorgestellten Zyklus einer SPS werden die Ein- und Ausgabe-Aktivitäten in einem Task zusammengefasst und am Anfang jedes Zyklus ausgeführt. Dieses Vorgehen hat Vorteile bei der Implementierung und verändert das Systemverhalten bis auf den ersten Zyklus nicht. Das Laufzeitsystem wird im Echtzeitbetrieb mit einer höheren Priorität als der Kernel des Betriebssystems ausgeführt und kann von ihm nicht unterbrochen werden. Damit das Betriebssystem trotzdem seine Aufgaben wahrnehmen kann, wird ihm ein vordefinierter Zeitschlitz am Ende des Grundzyklus des Laufzeitsystems gewährt (statische Slackzeit oder Funktionsreserve genannt).

Für die folgenden Use-Cases ist die Zykluszeit auf 100 ms und die Funktionsreserve auf 4 ms eingestellt. Dem Systemscheduler stehen somit 96 ms für die Aufteilung an ressourcenadaptive Tasks zur Verfügung. Dieser Wert gilt auch als eine harte Echtzeitschranke für das Laufzeitsystem. Zwecks der übersichtlichen Darstellung der Messergebnisse wurde die I/O-Aktivität für die folgenden Use-Cases abgeschaltet.

7.2. Use-Case 1: Nicht-echtzeitfähige Kommunikation

Eines der Hauptziele für die Motivation dieser Arbeit (vgl. Kapitel 1) war die Nutzung der Slackzeit für den Betrieb einer nicht-echtzeitfähigen Kommunikationsschnittstelle neben der Echtzeitfunktionalität des Laufzeitsystems. Die Vorstellung der möglichen Anwendungen des vorgestellten Frameworks für RA-Anwendungen wird mit diesem Use-Case eröffnet.

In der aktuellen Version existieren im ACPLT/RTE zwei dedizierte Tasks: ein Task für

die operative Bausteinlogik (der sogenannte „Urtask“) und ein Task für die Kommunikationstreiber. Der operative Task läuft normalerweise mit einer Zykluszeit von einer Sekunde und wird für die Ausführung von POUs und hierarchisch untergeordneten Tasks verwendet. Der Kommunikationstask wird mit einer Frequenz von einem kHz ausgeführt und stößt die Ausführung verschiedener untergeordneter Kommunikationssubsysteme an.

Die Implementierung der beiden zyklischen Tasks ist nahezu identisch. Dies ist in der Repräsentation der Vertreter der Kommunikationstreiber als Funktionsbausteine zwecks Introspektion begründet. Diese Treiber sind für eine minimale Laufzeit bzw. Unterbrechung ausgelegt. Es wird in der Regel pro Aufruf ein nach oben beschränkter Abschnitt des Transmission Control Protocol (TCP)-Stroms bearbeitet bzw. zu einem Zwischenpuffer hinzugefügt. Dadurch wird höchstens eine HTTP bzw. OPC UA Nachricht pro Aufruf bearbeitet. Darüber hinaus werden bestimmte Annahmen über die Längen bzw. die Komplexität der empfangenen Nachrichten angenommen, um sinnvolle Echtzeitschranken annehmen zu können.

Da die Ausführung der POUs die Einhaltung des Taktes für den Kommunikationstask eventuell „stören“ kann, wird die Überwachung seiner Zykluszeit auf der Systemebene deaktiviert bzw. die Ausführungszeitpunkt zur Laufzeit angepasst. Für den Nutzer besteht keine Möglichkeit die Ressourcenzuteilung an die Kommunikation zu beobachten bzw. zu ändern. Im Gegensatz dazu werden beide Tasks durch das RA-Rahmenwerk auf die gleiche Art und Weise aufgerufen. Die Zuteilung der Ressourcen ist durch ein explizites Scheduling durch den Komponentenscheduler transparent und einstellbar.

In diesem Use-Case übernimmt das RA-Framework die Ausführungssteuerung der Task-Ebene. Die Migration der existierenden Tasks kann vollkommen transparent für die nichtzeitsensitiven POUs erfolgen. Aus dieser Perspektive demonstriert der Use-Case auch die Einfachheit der Migration und die Co-Existenz des Rahmenwerks mit bewährter Software (Anforderung N3 in Kapitel 4).

Der Zyklus des Laufzeitsystems wird durch den Komponentenscheduler in zwei Abschnitte partitioniert: einen Abschnitt für die Programmlogik und einen für die Kommunikation. Für das Beispiel wurde die folgende Aufteilung gewählt: Für einen Zyklus von 100 ms gehen 95 ms an die Programmlogik und 1 ms an den RA-Kommunikationstask, 4 ms bleiben somit für die Funktionsreserve übrig, während der die Kontrolle zurück an das Betriebssystem übergeben wird.

Die Task-Konfiguration für den Komponentenscheduler ist in Abbildung 7.3 dargestellt. Der einzige wiederholte Zyklus besteht aus zwei Slots, denen jeweils ein Task zur Ausführung zugeordnet ist. Der erste Slot führt den Logiktask „/Tasks/UrTask“ direkt aus. Der zweite Slot führt das ISSC „/TechUnits/communication“ aus, welches das online Scheduling der Kommunikationsschnittstellen übernimmt.

Dieses ISSC ist in Abbildung 7.4 dargestellt. Es entspricht dem Muster der Meilenstein-Methode aus Abbildung 6.18a und besteht aus einem Initialzustand und einem Breakpoint. Die erste Aktion des Initialzustands führt den Kommunikationstask „/communication/RootComTask“ aus, die zweite Aktion initiiert einen kurzen Sleep-Befehl von 10 μs über den beinhalteten Baustein „sleep“ aus, damit das Betriebssystem die Daten aus dem Kommunikationsstack bereitstellen kann. Im Gegensatz zum abstrakten Modell des ISSCs, können in der Implementierung auch die Breakpoints Aktionen ausführen, z. B. die Überprüfung von Guards. In dem Use-Case stößt der Breakpoint die Ausführung des „guard“ Bausteins an, der dem Ausdruck „$(\Delta^{curr} - c) \geq 1\ ms$“ entspricht. Somit kann der Initialzustand nur bei einer verbleibenden Verarbeitungszeit von 1 ms wiederholt be-

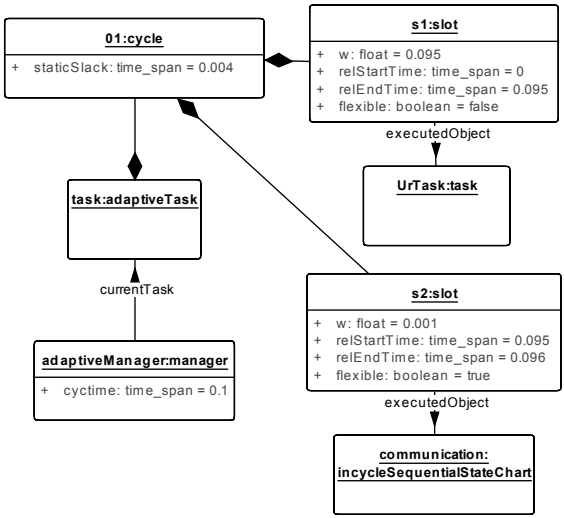


Abbildung 7.3.: Task-Konfiguration für den Use-Case.

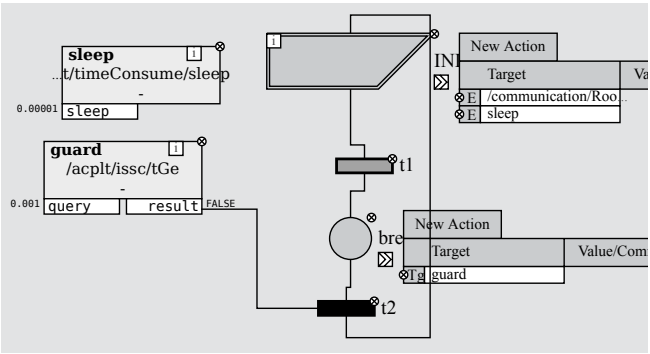


Abbildung 7.4.: ISSC für das online Scheduling des Kommunikationstasks.

treten werden. Der Wert von 1 ms ergab sich durch eine Abschätzung der WCET des Kommunikationstasks, die durch Messungen gestützt wurde.

Für die erste Messreihe wurde der schwankende Ressourcenverbrauch des Logiktasks durch einen Busy-Loop-Baustein simuliert, dessen Zeitverbrauch gleichverteilt im Intervall $[10, 95]$ ms lag. Dieser Baustein war der einzelne Eintrag in der „Urtask“ Taskliste.

Die Ergebnisse sind in Abbildung 7.5 dargestellt. Der Übersicht halber sind nur 100 Zyklen je 100 ms in der Abbildung zu sehen. Der ausgewählte Ausschnitt stellt eine Auswahl aus einer größeren Messreihe (ca. 12000 Zyklen) dar.

Das oberste Diagramm in Abbildung 7.5 illustriert die Aufteilung der gesamten Zykluszeit auf die beiden Tasks. Der hellgraue Teil entspricht dem Haupttask für die Logik („Urtask“) und der dunkelgraue Teil dem RA-Task für die Kommunikation bzw. dem ISSC. Es ist deutlich zu beobachten, wie der RA-Task die Schwankungen des Logiktasks ausgleicht und bei ca. 96 ms terminiert. Das oberste Diagramm zeichnet die reine Ausführungszeit der Tasks auf. Die Scheduling-Overheads wurden herausgerechnet.

Das zweite Diagramm von oben gibt Auskunft über die gesamte verbrauchte Zykluszeit. Da der Guard des ISSCs in Abbildung 7.4 auf 1 ms eingestellt ist, terminiert der Zyklus stets bei Werten zwischen 95 ms und 96 ms. Der Wert von 96 ms gilt dabei als eine harte Echtzeitschranke. Es ist zu sehen, dass diese nie erreicht wird (dies ist bei der gesamten Messreihe der Fall). Die Tatsache, dass die Zeit über 95 ms liegt, ist mit den Scheduling-Overheads zu erklären, die in diesem Diagramm miteinbezogen sind.

Das dritte Diagramm zeigt den Jitter des Beginns des Zyklus. Dieser Wert entspricht der „Verspätung“ des tatsächlichen Zyklusanfangs im Vergleich zu der geplanten Zeit. Das Jitter ist auf die Ungenauigkeit des Sleep-Befehls des Betriebssystems zurückzuführen. ACPLT/RTE implementiert keine Maßnahmen zur Kompensation des Jitters. Aus diesem Grund ist dieser immer positiv. Die absoluten Werte des Jitters befinden sich allerdings maximal bei ca. 40 μs, was den meisten Anwendungen des Laufzeitsystems genügt.

Das letzte Diagramm gibt Auskunft über den expliziten Scheduling-Overhead. Dieser setzt sich aus dem Overhead des internen Scheduler des ACPLT/RTE, dem Overhead des Komponentenschedulers und der Auswertung des ISSCs zusammen. Der maximale Overhead für das Scheduling liegt im Bereich von ca. 110 μs. Auch dieser Wert erscheint für Anwendungen in der Domäne der Prozessleittechnik akzeptabel.

In der zweiten Messreihe wurde der Netzwerkdurchsatz der RA-gesteuerten Kommunikation in Abhängigkeit von der Auslastung des Logiktasks gemessen. Im Unterschied zu der ersten Messreihe wurde die Dauer des Logiktasks nicht zufällig, sondern manuell auf Werte im Intervall $[0, 95]$ ms eingestellt. Für die Demonstration des Vorgehens werden bereits implementierte Kommunikationsschnittstellen des ACPLT/RTE Laufzeitsystems verwendet. Diese Schnittstellen umfassen ein auf HTTP basierendes Protokoll ACPLT/KS-HTTP, sowie das binäre OPC UA Protokoll. Für die letzte Kommunikationsschnittstelle wird eine offene OPC UA Implementierung open62541 [PGP+15] eingesetzt.

In der Messreihe wurde die Dauer einer Leseoperation beider Protokolle gemessen. Diese Operation entspricht einer GET Operation in HTTP bzw. dem Aufruf des Read Dienstes in OPC UA. Die Messung umfasst nur die Dauer des unmittelbaren Lesevorgangs und nicht den Aufbau der Kommunikation, z. B. den Verbindungsaufbau in OPC UA. Bei dem Informationsaustausch wurde der gleiche Payload übertragen. Es handelt sich dabei um eine ca. 5,5 KB große HTML-Seite.

Die Messungen des HTTP-Protokolls erfolgten mit dem Apache Benchmark Tool. Die Messungen der OPC UA Performance wurden mit einem auf dem open62541 Projekt ba-

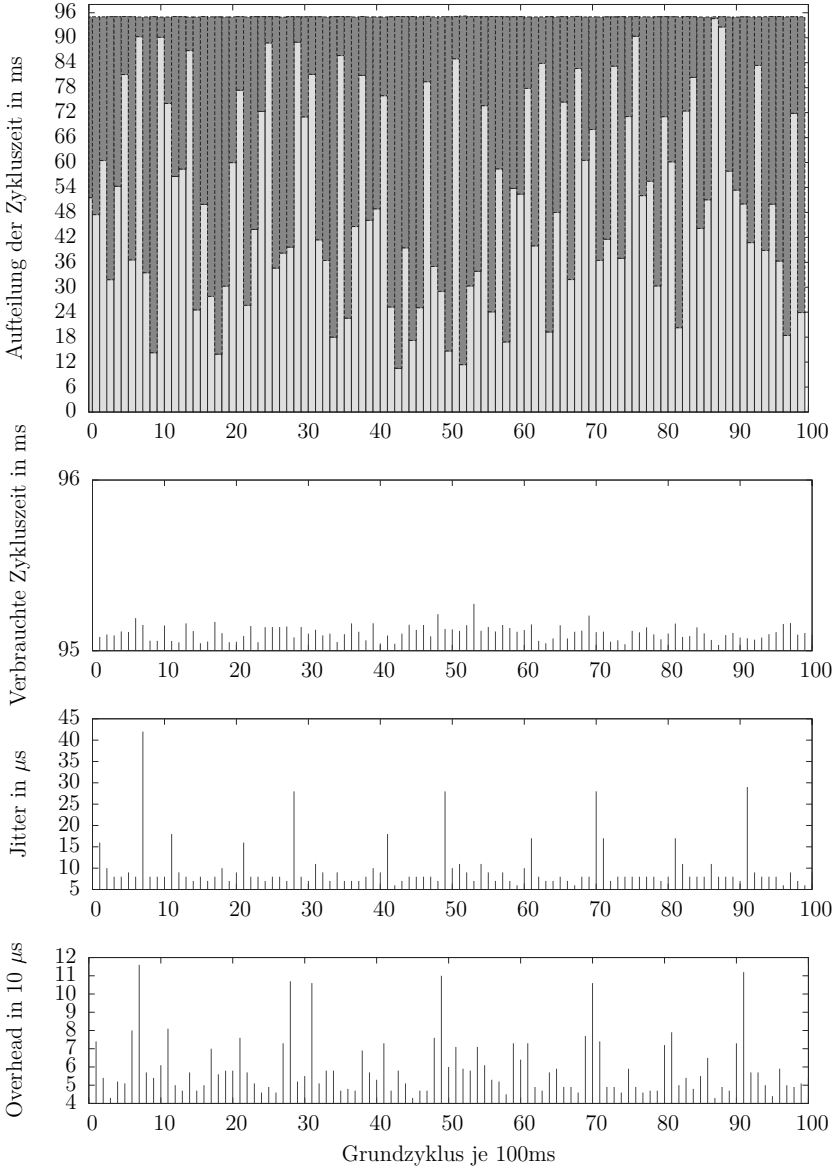


Abbildung 7.5.: Messungen der Aufteilung der Zykluszeit auf die einzelnen Tasks (von unten nach oben: Logiktask mit zufälliger Ausführungszeit in hellgrau, RA-Task für die Kommunikation in dunkelgrau), der verbrauchten Zeit innerhalb des gesamten Zyklus, des Jitters und des Scheduling-Overheads in einem repräsentativen Zeitraum von 100 Zyklen je 100 ms.

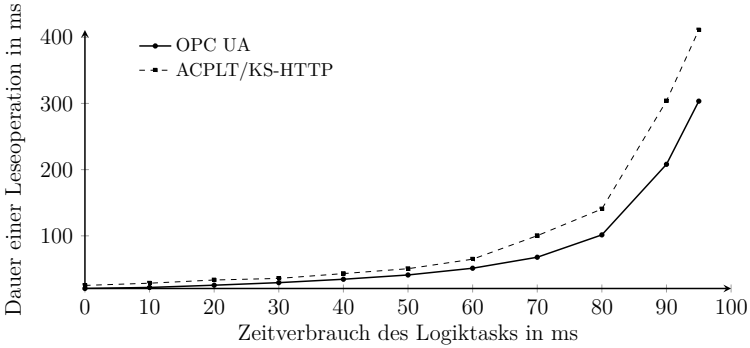


Abbildung 7.6.: Die Dauer einer Leseoperation für beide Protokolle in Abhängigkeit von der Auslastung des Logiktasks (Mittelwert je 1000 Anfragen).

sierenden Client durchgeführt. Bei den Tests kommt es somit nicht auf das absolute erreichbare Maximum des Netzwerkdurchsatzes, sondern auf die Darstellung der Skalierbarkeit des Durchsatzes in Abhängigkeit von der Auslastung des Logiktasks an. Aus diesem Grund wurden beide Clients zwecks besserer Reproduzierbarkeit im single-threaded Modus ausgeführt. Der Client-PC (mit einer Intel Core i5-2520M CPU und Linux 3.13) und die Hardwareplattform des Laufzeitsystems wurden mittels geswitschtem Fast Ethernet verbunden.

Die Ergebnisse der Messung sind in Abbildung 7.6 als Mittelwerte je 1000 Anfragen dargestellt. Auf der Abszisse ist der Zeitverbrauch des Logiktasks dargestellt. Diese Dauer entspricht der Höhe des hellgrauen Balkens im oberen Diagramm der Abbildung 7.5. Es ist eine lineare Zunahme der Dauer einer Leseoperation im Bereich von 0 bis 70 ms für beide Kommunikationsprotokolle zu beobachten. Für Werte größer als 70 ms kommt es zu einem signifikanten Anstieg der Latenz. Dieser ist insbesondere bei einer Auslastung des Urtasks von 95 ms sichtbar. In diesem Fall kann die Kommunikation nur eine (Teil-)Nachricht pro Grundzyklus des Laufzeitsystems verarbeiten. Trotz des relativ niedrigen Durchsatzes war die Kommunikation auch unter diesen Bedingungen jedoch stets möglich.

Beide Messreihen belegen die erwarteten Ergebnisse. Die Slackzeit eines Laufzeitsystems kann durch das online Scheduling effektiv für nicht-echtzeitfähige Kommunikation genutzt werden, ohne die Einhaltung der Echtzeitschranken zu gefährden. Der Durchsatz hängt dabei von der verfügbaren Slackzeit ab und sinkt bei besonders wenig verfügbarer Slackzeit signifikant.

7.3. Use-Case 2: Prozessbegleitende Simulation mit variabler Qualität

Die Einsatzgebiete der prozessbegleitenden Simulation umfassen die Überwachung von automatisierten Prozessen, das Schätzen der Systemzustände und die Erstellung kurzfristiger Prognosen des Systemverhaltens.

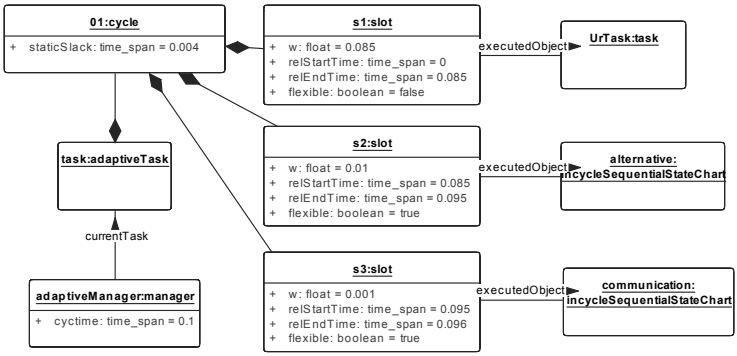


Abbildung 7.7.: Task-Konfiguration für den Use-Case.

Die eigentliche Simulation kann dabei entweder analytische oder heuristische Modelle verwenden, die auf Messdaten basieren. Hybride Verfahren als Kombination beider Ansätze sind auch möglich. Die meisten Simulationsverfahren beinhalten das Lösen von Differenzialgleichungen. Dieses erfolgt normalerweise numerisch mithilfe unterschiedlicher Lösungsverfahren, die die Lösung iterativ approximieren.

Die Laufzeit des Lösungsverfahrens hängt von deren Parametrierung ab, wie z. B. der gewünschten Genauigkeit der Approximation oder der Weite der diskretisierten Zeitschritte. Ein mögliches Einsatzgebiet von RA-Verfahren kann dabei eine dynamische Auswahl des eingesetzten Verfahrens bzw. des Parametersatzes sein. Die Genauigkeit der Simulation kann dabei in Abhängigkeit von der verfügbaren Rechenzeit in diskreten Schritten variiert werden.

Für diesen Use-Case wird eine einfache Auswahl zwischen zwei alternativen Verfahren implementiert, die dem Mehrversionen-Muster aus Abbildung 6.17b entspricht. Im Gegensatz zu dem ersten Use-Case in Abschnitt 7.2, ist es sinnvoll, die Ergebnisse des RA-Verfahrens innerhalb des aktuellen Zyklus operativ einzusetzen. Dabei können z. B. die Schätzungen des zukünftigen Systemverhaltens als eine zusätzliche Eingabe des Regelalgorithmus genutzt werden. Dieser Verarbeitungsschritt wurde in diesem Anwendungsfall der Einfachheit halber nicht explizit modelliert.

Die Struktur des Tasks ist in Abbildung 7.7 dargestellt. Ähnlich zu dem ersten Use-Case in Abschnitt 7.2 wird dabei die Schwankung des Logiktask im Intervall von $[10, 85]$ ms simuliert. Im zweiten Slot wird dabei das ISSC der Messwertvalidierung ausgeführt, dessen WCET 10 ms beträgt. Das Chart ist in Abbildung 7.8 explizit dargestellt. Die Engineering-Ansicht zeigt sowohl das ISSC als auch die ausgeführte Programmlogik. Die Ausführungslogik ist dabei einfach: Im Initialzustand wird mithilfe eines Guards überprüft, ob 20 ms für die Ausführung verfügbar sind. Ist das der Fall, so wird der Zustand „ver1“ betreten, sonst wird der Zustand „ver2“ ausgeführt. Die Aktionen der Zustände führen dann die FBDs „ver1_fc“ bzw. „ver2_fc“ aus. Die Ausführung der ersten Version dauert 20 ms, die Ausführung der zweiten Version 10 ms. Die Dauer der Ausführung wird durch parametrierte Bausteine simuliert. Die Synchronisation der Ausgabe der Simulationsverfahren erfolgt durch einen Multiplexer-Baustein „mux“, der von jeder Version ausgeführt

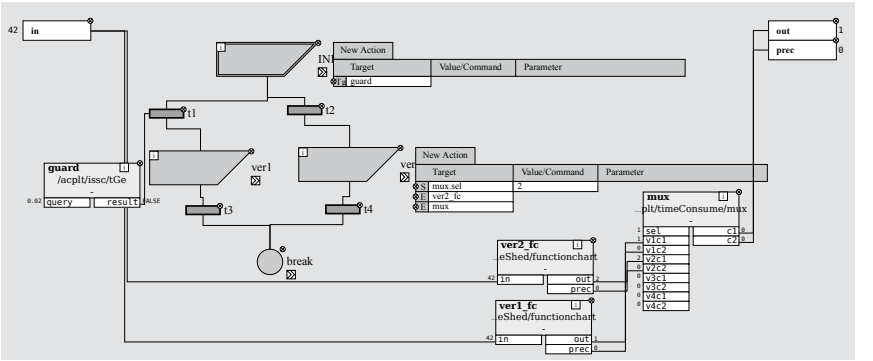


Abbildung 7.8.: ISSC für das online Scheduling der prozessbegleitender Simulation.

wird. Je nach ausgeführter Version, wird dabei die Ausgabe entsprechend umgeschaltet. Der Ausführungsrahmen des Charts entspricht dabei den Interfaces der FBDs der einzelnen Versionen. „Von außen“ ist somit die Nutzung des ressourcenadaptiven Verfahrens vollkommen transparent.

Die Ergebnisse der Messungen sind in Abbildung 7.9 dargestellt. Jede Spalte stellt dabei einen Zyklus des Laufzeitsystems dar. Jede Spalte ist in drei Teile aufgeteilt: Der untere hellgraue Teil entspricht der Dauer des Logiktasks, der mittlere graue Teil entspricht der Dauer des RA-Tasks für prozessbegleitende Simulation und der obere hellgraue Teil entspricht der Ausführungsdauer des Kommunikationstasks (vgl. Abschnitt 7.2). Es ist deutlich zu erkennen, wie die Dauer des mittleren Abschnitts zwischen 10 und 20 ms variiert, z. B. in Zyklen 99 und 100 der Abbildung 7.9. Der Jitter und die anderen Parameter entsprechen denen der ersten Messreihe (vgl. Abbildung 7.5) und sind aus diesem Grund nicht explizit abgebildet.

7.4. Use-Case 3: Mehrstufige Messwertvalidierung

Verfahren der Messwertvalidierung werden in der operativen Leittechnik eingesetzt, um die „Zuverlässigkeit“ eines Messwertes festzustellen und gegebenenfalls auf fehlerhafte Messungen proaktiv reagieren zu können. Dem tatsächlichen Messwert wird dabei ein Vertrauensindex zugeordnet, der seine Qualität beschreibt. Dieser Index kann in den Regelungsalgorithmus einfließen, beispielsweise durch das Umschalten der Stellgröße auf einen manuell eingestellten Wert [Uec05].

Die Verfahren der Validierung sind vielfältig und reichen von einfacher Signalanalyse bis zu komplexen modellgestützten Analyseverfahren. Im Regelfall werden die einzelnen Verfahren sukzessive ausgeführt, dabei wird der Vertrauensindex jedes Verfahrens an das nächste Verfahren weitergegeben und kann so berücksichtigt werden. Diese Konstellation wird als „mehrstufige Messwertvalidierung“ [Uec05] bezeichnet und ist schematisch in Abbildung 7.10 dargestellt.

Das Einsatzgebiet der Ressourcenorientierung umfasst die bedingte Ausführung der ein-

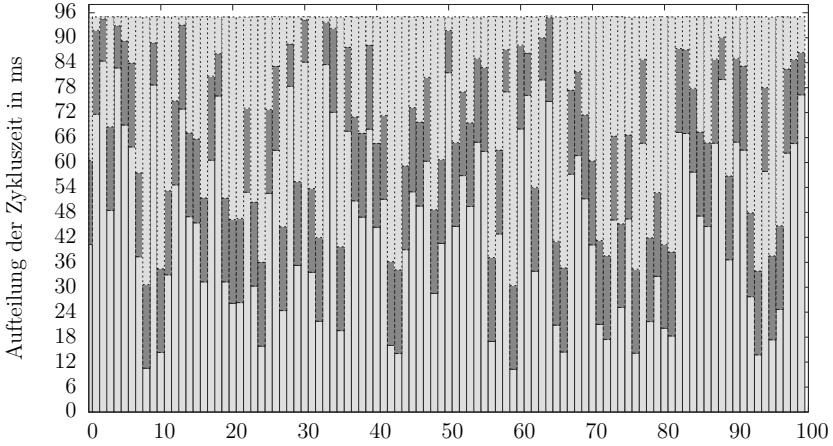


Abbildung 7.9.: Messungen der Aufteilung der Zykluszeit auf die einzelnen Tasks (von unten nach oben: Logiktask mit zufälliger Ausführungszeit in hellgrau, RA-Task für die alternative Auswahl des Simulationsverfahren in dunkelgrau, RA-Task für die Kommunikation in hellgrau) in einem repräsentativen Beobachtungszeitraum von 100 Zyklen je 100 ms.

zelnen Stufen der Validierung. Dabei kann beispielsweise die Ausführung der ersten Stufe zugesichert werden, während die folgenden Stufen nur bei ausreichend vorhandener Slackzeit ausgeführt werden.

Für den Use-Case wurde die gleiche Task-Konfiguration wie im letzten Use-Case verwendet (vgl. Abbildung 7.7). Die Ausführung der prozessbegleitenden Simulation wurde durch das ISSC in Abbildung 7.11 ersetzt. Das Chart folgt dem Muster der 0/1 Ausführung (vgl. Abbildung 6.17) und enthält die Logik einzelner Validierungsstufen und das Chart zu deren Steuerung. Die Zustände des Charts führen die FBDs „tier1_fc“ bis „tier3_fc“ sowie den Multiplexer für die Datenausgabe aus. Die Ausführung der Stufen wird mit je 10 ms simuliert. Durch die garantierte Ausführung der ersten Stufe beträgt die WCET des Charts 10 ms. Auf der Abbildung sind die Aktionen des dritten Zustands beispielhaft dargestellt. Die

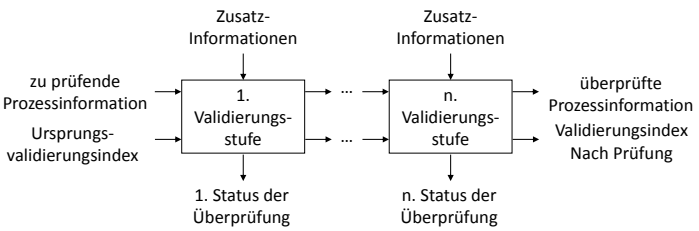


Abbildung 7.10.: Mehrstufige Messwertvalidierung [Uec05].

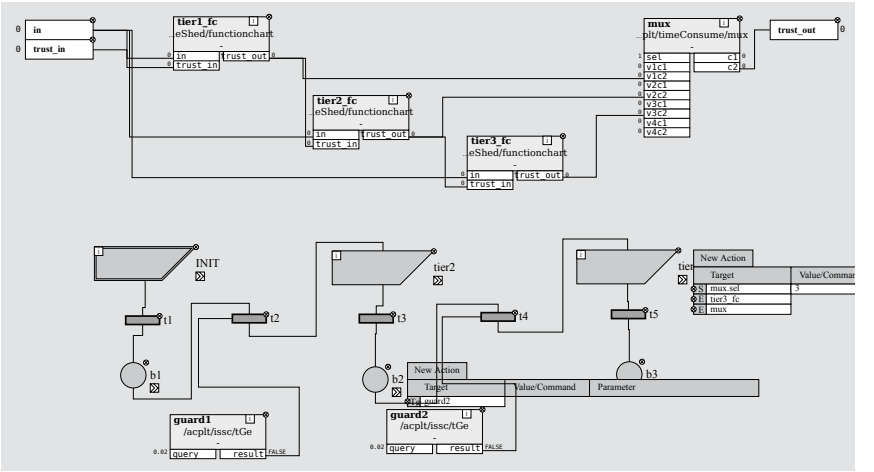


Abbildung 7.11.: ISSC für das online Scheduling der mehrstufigen Messwertvalidierung.

optionale Ausführung der Stufen wird durch die Breakpoints und entsprechende Guards gewährleistet. Die Breakpoints dürfen dabei nur verlassen werden, wenn die verfügbare Ausführungszeit mehr als 10 ms beträgt.

Die Messreihe der Ausführung ist in Abbildung 7.12 dargestellt. Ähnlich zu Abbildung 7.9 wird dabei die gemessene Ausführungszeit (dunkelgrau) der Messwertvalidierung in dunkelgrau von der schwankenden Ausführungszeit des Logiktasks unterhalb und der variablen Ausführungszeit des Kommunikationstasks oberhalb (beide in hellgrau) eingerahmt. Es ist klar erkennbar, wie die gemessene Ausführungszeit der Validierung diskret zwischen 10, 20 und 30 ms variiert, ohne die Echtzeitschranke von 96 ms zu verletzen. Damit ist die Anwendbarkeit der RA-Verfahren auch im Kontext dieser Anwendung bestätigt.

7.5. Use-Case 4: Transaktionskontrolle für regelbasiertes Engineering

Das regelbasierte Engineering wird vor allem im Kontext der AdA eingesetzt. Die Anwendungsfälle für das regelbasierte System sind vielfältig, beinhalten aber immer die Erzeugung bzw. die Modifikation der ausgeführten Programmlogik. Falls diese Logik zur Laufzeit des Systems erstellt bzw. verändert wird, muss sichergestellt sein, dass das System sich zu jedem Zeitpunkt in einem konsistenten Zustand befindet.

Normalerweise werden Engineering-Eingriffe über eine nicht-echtzeitfähige Kommunikationsschnittstelle übermittelt. Aus diesem Grund ist eine einmalige Übertragung der gesamten Änderung oft nicht möglich. Daher wird diese bei einem zyklischen Laufzeitsystem zwingend auf unterschiedliche Grundzyklen des Systems verteilt. Bei einem naiven Ansatz der direkten Logikänderung über die Kommunikationsschnittstelle besteht immer die Möglichkeit, dass semantisch zusammenhängende Änderungen in unterschiedlichen Zyklen

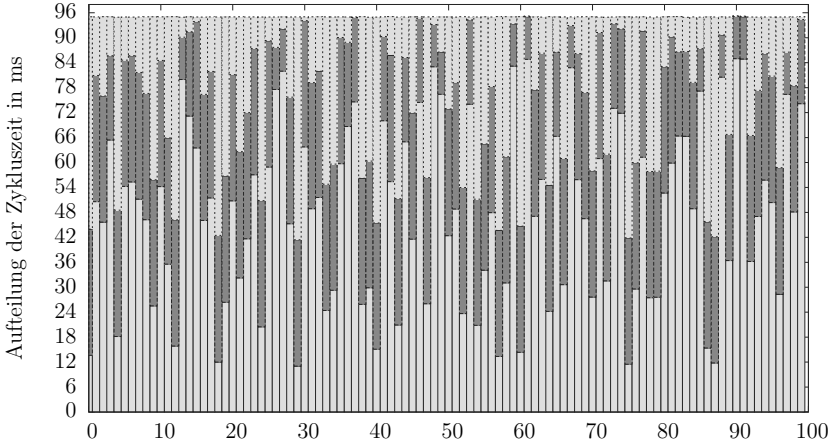


Abbildung 7.12.: Messungen der Aufteilung der Zykluszeit auf die einzelnen Tasks (von unten nach oben: Logiktask mit zufälliger Ausführungszeit in hellgrau, RA-Task für die mehrstufige Validierung in dunkelgrau, RA-Task für die Kommunikation in hellgrau) in einem repräsentativen Beobachtungszeitraum von 100 Zyklen je 100 ms.

wirksam werden und somit einen unerwünschten Systemzustand hervorrufen. Ein Beispiel für einen solchen Zustand ist eine partielle Aktivierung der Funktionsbausteine innerhalb eines FBNs. Ein anderes Beispiel ist eine nicht komplette Parametrierung bzw. Erstellung eines Objekts bzw. einer Komposition aus Objekten. Die besondere Bedeutung von Transaktionssicherheit während der Modellveränderungen wurde auch bereits im Kontext der verteilten Systeme angesprochen [Mer16].

Eine Möglichkeit der Zusicherung des konsistenten Systemzustands besteht darin, eine dedizierte Transaktion zu erstellen und sie unter Echtzeitbedingungen auszuführen. Die Erstellung unterliegt dabei keinen Echtzeitanforderungen und wird durch die nichtechtzeitfähige Dienste abgewickelt. Die Ausführung hingegen muss im Echtzeitbetrieb stattfinden. In diesem Use-Case wird die Nutzung des Transaktion-Musters aus Abbildung 6.18 vorgeschlagen, um eine solche Transaktion zu definieren und auszuführen. Das System muss eine Grundmenge an Operationen und deren Zeitabschätzungen, wie z. B. das Anlegen und das Löschen von Objekten bereitstellen, die von den Aktionen des ISSCs aufgerufen werden können.

Für die Zwecke der Evaluation wurde eine Task-Konfiguration aus dem vorhergehenden Use-Case in Abschnitt 7.4 verwendet. Das ausgeführte ISSC ist in Abbildung 7.13 dargestellt. Durch die Verwendung eines Zähler-Bausteins „add1“ wird der Initialzustand 50 mal betreten und stößt die Ausführung des „process“ Bausteins an, der 10 ms verbraucht und die Aktivität der Transition simuliert. In dem History-Breakpoint „b1“ wird die verbliebene Ausführungszeit durch den Guard überprüft. Nach der 50. Ausführung wird die Transition „t1“ deaktiviert, das Transaktion-Chart betritt den History-Breakpoint „b2“ und terminiert.

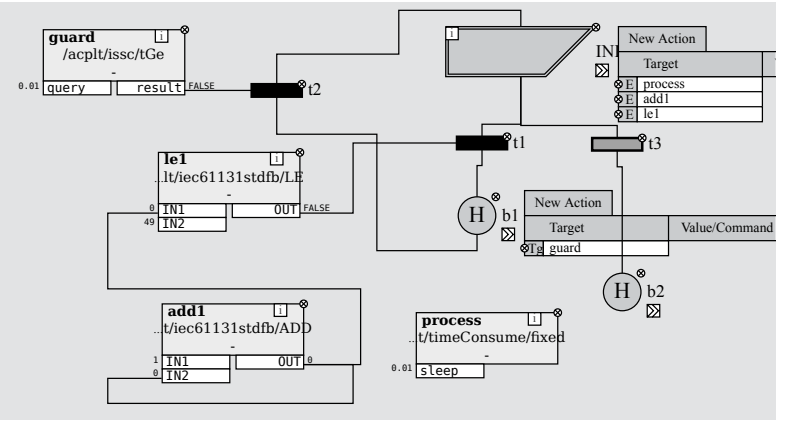


Abbildung 7.13.: ISSC für das online Scheduling der Transaktion.

Aus der Perspektive des Schedulings ist eine Transaktion ein sporadischer Task. Dieser kann in der online Phase des Schedulers ausgeführt werden (vgl. Abschnitt 6.3.3). Da die minimale WCET der Transaktion von 10 ms über der Funktionsreserve von 4 ms liegt, kann die Ausführung nur nach einer best-effort Strategie erfolgen. Diese Strategie kann anhand der folgenden Messreihe visualisiert werden.

Die Messreihe ist in Abbildung 7.14 dargestellt. Die Transaktion wurde durch den Kommunikationstask erstellt und liegt als ein Objektmodell im System vor. Die Ausführung der Transaktion wurde im Zyklus Nummer 50 angestoßen. Der Scheduler versucht ab diesem Zeitpunkt der Transaktion möglichst viel Slackzeit zuzusichern. Aus diesem Grund werden den einzelnen Tasks nur die minimale benötigte Zeit w eingeräumt (vgl. Abbildung 7.7). Diese Maßnahme betrifft die Messwertvalidierung und den Kommunikationstask. Diese werden nun stets mit Δ^{curr} von 10 ms bzw. 1 ms ausgeführt. Die Möglichkeit der Ausführung der Transaktion wird nach der Terminierung der regulären Tasks überprüft. Ob die Transaktion ausgeführt werden kann oder nicht, hängt von der Schwankung des Logiktasks ab. Es ist zu sehen, dass die Transaktion erstmalig im 50. Zyklus für 10 ms ausgeführt wurde. In Zyklen Nummer 51 und 52 war hingegen keine Zeit für die Ausführung der Transaktion vorhanden. Im Zyklus Nummer 66 war die Ausführung der Transaktion beendet. Ab diesem Zyklus teilt der Scheduler wieder die gesamte verfügbare Slackzeit den RA-Tasks zu.

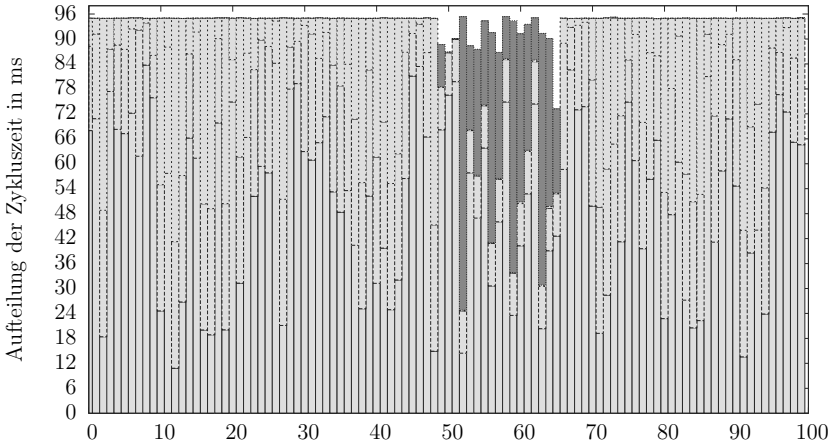


Abbildung 7.14.: Messungen der Aufteilung der Zykluszeit auf die einzelnen Tasks (von unten nach oben: Logiktask mit zufälliger Ausführungszeit RA-Task für die mehrstufige Validierung und RA-Task für die Kommunikation in hellgrau, sowie die Transaktion in dunkelgrau) im Zeitraum von 100 Zyklen je 100 ms.

8. Diskussion der Ergebnisse

In dieser Dissertation wurde ein Rahmenwerk für die Integration von ressourcenadaptiven Anwendungen in die Laufzeitsysteme der operativen Prozessleittechnik vorgestellt. Die Notwendigkeit des ressourcenadaptiven Einsatzes wurde aus den eingangs vorgestellten wachsenden Anforderungen an die Funktionalität der Komponenten industrieller Produktionssysteme und insbesondere an die Laufzeitsysteme abgeleitet. Diese zusätzlichen Anforderungen umfassen die Bereitstellung zusätzlicher Information bzw. die Ausführung zusätzlicher nicht-echtzeitfähiger Funktionalität durch die Laufzeitumgebung. Eine reguläre Ausführung dieser Funktionen ist häufig aus Gründen der festen Ressourcenzuteilung an die echtzeitkritischen Kern-Funktionen eines Laufzeitsystems nicht möglich. Ressourcenadaptive Anwendungen erlauben hingegen die Ausführung zusätzlicher Funktionalität in der Slackzeit, der Zeit, die wegen der Fluktuationen der Ausführungszeiten der Kern-Funktionen in den meisten Fällen ungenutzt bleibt. Das Konzept der Ressourcenadaptivität erlaubt somit eine effizientere Nutzung der vorhandenen Rechenkapazität des Laufzeitsystems bei gleichzeitiger Sicherstellung der garantierten Echtzeitanforderungen.

Die Idee der Inanspruchnahme der Slackzeit zyklischer Systeme ist nicht neu. So wird diese Zeit bereits im Bereich der Echtzeitsysteme, z. B. PikeOS [KF06], oder von Laufzeitsystemen der Automatisierungstechnik, z. B. FASA (vgl. Abschnitt 3.3.3), verwendet. Der erste Ansatz verteilt die übrig gebliebene Zeit an wartende Threads mit niedrigerer Priorität. Der zweite Ansatz nutzt die Slackzeit für Threads mit Verwaltungsaufgaben, z. B. Aufgaben der Umstrukturierung der Scheduling-Tabelle. Die implizite „Abgabe“ der ungenutzten Zeit an Tasks mit niedrigerer Priorität ist auch im Kontext der Implementierungen der IEC 61131-3 mit unterbrechendem Scheduling der Fall. Die Norm schreibt die Möglichkeit eines solches Verhaltens explizit vor. Der Nachteil des Multi-Threading Ansatzes ist die fehlende Sicherheit über die Ausführungsdauer der niedrig priorisierten Tasks. Somit kann es keine Garantie über den Zeitpunkt der Terminierung einer Funktion geben, dafür aber über die Eigenschaften des berechneten Ergebnisses.

Der Kernunterschied des vorgestellten Rahmenwerks zu diesen Ansätzen ist die vorhandene Möglichkeit der Synchronisation der ressourcenadaptiven Anwendung mit dem Zyklus des Laufzeitsystems. Das Ende des Zyklus steht als Zeitpunkt der Terminierung jeder Berechnung fest und die „Qualität“ des Ergebnisses ist variabel. Mit diesem neuen Ansatz sind deterministische operative Eingriffe der ressourcenadaptiven Logik in den Prozess möglich, wie in Use-Cases 2 und 3 in Abschnitten 7.3 bzw. 7.4 demonstriert wurde. Der „Einflussbereich“ der operativen Eingriffe ist insbesondere auf der Prozessleitebene sehr weit ausgeprägt. Es gehören nicht nur explizite Funktionen mit Manipulationen der I/O hinzu, sondern auch alle Daten, deren Änderung implizit auf das physische System wirken kann. Falls beispielsweise eine Modelländerung implizite Auswirkungen auf das physische System besitzt, z. B. über ein regelbasiertes System Verhaltensänderungen induziert, so muss der Zugriff auf dieses Modell ähnlichen Anforderungen wie direkter I/O Zugriff unterliegen. Eine bisherige Nutzung der Slackzeit für die Realisierung des operativen Eingriffs in den Prozess für Aufgaben der Automatisierungstechnik ist dem Autor unbekannt.

Es ist nochmals zu betonen, dass die Verfügbarkeit der Slackzeit im Allgemeinen nicht garantiert werden kann. Aus diesem Grund eignen sich ressourcenadaptive Anwendungen vor allem für nicht-echtzeitfähige Aufgaben bzw. Erweiterungen, wie z. B. nicht-echtzeitfähige Kommunikation, oder fakultative Verbesserungen der Kern-Logik des Laufzeitsystems, wie z. B. optionale zusätzliche Stufen eines Validierungsverfahrens für Messwerte. Auswertungen der Messdaten einer realen Forschungsanlage zeigen Laufzeitschwankungen im zweistelligen Prozentbereich, sodass durchaus umfangreiche Zusatzfunktionalitäten durch das Rahmenwerk eingebettet werden können.

Der Nachteil der nicht garantierbaren Verfügbarkeit der Slackzeit ist gleichzeitig ein Vorteil der Ressourcenadaptivität bezüglich der Integration mit bereits existierenden leittechnischen Anwendungen. Ressourcenadaptive Algorithmen können als ein optionales Add-on betrachtet werden und lassen die Semantik des existierenden Programms unverändert. Damit hat die Migration bzw. das Hinzufügen der ressourcenadaptiven Anwendungen keine Auswirkungen auf die echtzeitrelevante Funktionalität und erfordert keine erneute bzw. zusätzliche Verifikation der existierenden Software.

Dieser Vorteil der transparenten Koexistenz ist der eingeführten einheitlichen Laufzeitarchitektur geschuldet (vgl. Abschnitt 6.1). Diese Architektur folgt dem Prinzip des hierarchischen Schedulings und lässt somit die Kombination beliebiger Kontrollflusstypen zu. Auch die prototypische Implementierung des Rahmenwerks hat gezeigt, dass beliebige Kombinationen aus ressourcenadaptiven und nicht ressourcenadaptiven POUs in beide Richtungen möglich sind. So können zum einen ressourcenadaptive Anwendungen aus bereits existierenden POUs zusammengestellt werden, die unter bestimmten Voraussetzungen ausgeführt werden (vgl. Use-Cases 2 und 3 in Abschnitten 7.3 bzw. 7.4). Zum anderen können ressourcenadaptive Anwendungen von nicht zeitsensitiven Schedulern ausgeführt werden und entfalten bei solcher Ausführung nur ihren minimalen Funktionsempfang.

Eine domänenspezifische Sprache für die Beschreibung des ressourcenadaptiven Verhaltens der POUs mit Namen In-cycle Sequential State Chart (ISSC) wurde in Abschnitt 6.2 eingeführt. Diese Prozedurbeschreibungssprache lehnt sich an die existierenden Sprachen der Leittechnik an und erfüllt somit die Erwartungen der potentiellen Endnutzer. Neben dem Vorteil eines leichten Einstiegs in die neue Sprache, können auch die existierenden Engineering-Lösungen mit minimalen Anpassungen übernommen werden. Tatsächlich wurde eine Engineering-Umgebung der prototypischen Implementierung von einer existierenden Software abgeleitet. Trotz einer ähnlichen Syntax bleibt die Sensitivität in Bezug auf die verfügbare Ausführungszeit das wichtigste Unterscheidungsmerkmal der ISSCs. Dem Autor ist kein ähnliches Konzept im Bereich der Prozedurbeschreibung aus der Domäne der Automatisierungstechnik bekannt.

Die in Abschnitt 6.3 vorgestellte Referenzarchitektur des Komponentenschedulers bedient sich der breiten Auswahl der existierenden Scheduling-Konzepte wie Predictably Flexible Real-Time Scheduling und Elastic Model. Einige der beschriebenen Features dieser Referenzarchitektur, wie beispielsweise die Möglichkeit des Wechsels der offline Scheduling Tabelle, finden sich in anderen Laufzeitumgebungen, z. B. in FASA, wieder. Letztendlich wurde ein auf die Problemstellung zugeschnittenes Schedulingmodell entwickelt, das die Vorteile einer einfachen Implementierung mit den Garantien des strikten Determinismus und der praktischen Lösbarkeit der zugrundeliegenden Optimierungsprobleme vereint. Diese in der Konzeptionsphase anvisierten Vorteile wurden durch die prototypische Implementierung bestätigt und konnten durch die Auswertung empirischer Messdaten belegt werden.

Die drei entwickelten Bestandteile des Rahmenwerk-Konzepts erfüllen die eingangs anvisierten Ziele dieser Arbeit. Insgesamt ist mit dem Rahmenwerk und dessen Integration in die Laufzeitumgebung ACPLT/RTE ein abgeschlossenes Paket entstanden, dessen Einsatzmöglichkeiten mithilfe mehrerer Use-Cases demonstriert wurden. Die Anwendungsszenarien knüpfen dabei an die aktuellen bzw. kürzlich abgeschlossenen Forschungsarbeiten des Lehrstuhls für Prozessleittechnik an und zeigen damit die praktische Relevanz der Anwendung des ressourcenorientierten Ansatzes in der Domäne der Prozessleittechnik.

Im Folgenden werden die für diese Arbeit angenommenen Eigenschaften des Laufzeitsystems bezüglich des Schedulings kritisch überprüft:

- Die Annahme einer konstanten Zyklusdauer ist die grundlegende Voraussetzung für die Entstehung der Slackzeit. In Anwendungsgebieten der reinen Steuerungstechnik sind am Markt Systeme mit flexibler Zykluszeit verfügbar. Bei solchen Systemen wird ein neuer Zyklus direkt nach der Terminierung der Kontrolllogik gestartet. Die Begründung einer festen Zykluszeit ist im Bereich der Prozessautomatisierung mit dem Abtasttheorem, den Aufgaben der Regelung und der Forderung eines reproduzierbaren dynamischen Verhaltens verbunden. Aus diesen Gründen wird die Annahme, insbesondere in dieser Domäne, mit hoher Wahrscheinlichkeit weiterhin gültig sein.

- Die Annahme der Stabilität des Prozessabbildes ist in der IEC 61131-3 verankert und wird somit in absehbarer Zeit Bestand haben.

- Die Annahme des kooperativen Schedulings und somit der Nichtunterbrechbarkeit der Tasks ist in Bezug auf die IEC 61131-3 Umgebungen nur für manche Systeme gültig. Die Vorteile der Nichtunterbrechbarkeit bezüglich der strikten Zyklussynchronisation sind bereits erläutert worden. Für kommerzielle Systeme überwiegen oft aber die Nachteile der Fragilität, sodass unterbrechende Scheduler der Echtzeitsysteme eingesetzt werden. Von den entwickelten Bestandteilen des Rahmenwerks ist nur der Komponentenscheduler von der Annahme der Nichtunterbrechbarkeit betroffen. Die einheitliche Softwarearchitektur und die ISSCs sind auch im Kontext des unterbrechbaren Schedulers weiterhin anwendbar.

- Die Annahme eines zeitgesteuerten Systems ist mit Hinblick auf die Entwicklung der IEC 61499 zu überprüfen. Auch im Fall der wachsenden Anwenderakzeptanz der Norm, können Teile der vorgestellten Arbeit, insbesondere die einheitliche Laufzeitarchitektur, im Kontext der Ereignissteuerung Anwendung finden.

- Eine ähnliche Argumentation gilt für das angenommene Uniprozessor-System. Die Umsetzungsprobleme der Interprozesskommunikation und -synchronisation führen dazu, dass die Mehrkernsysteme aus der Softwareperspektive als mehrere Ressourcen dargestellt werden. Das Rahmenwerk ist somit auch in diesem Fall weiterhin anwendbar.

Abschließend werden Themen zukünftiger Arbeiten bzw. die Erweiterungen des vorgestellten Rahmenwerks besprochen: Eine feinere Modellierung der Guards der ISSCs im UPPAAL-Toolkit ist die erste mögliche Forschungsrichtung. Die Guards werden bis dato nur als externe Variablen betrachtet. Somit ist die Modellierung der gemischten Bedingungen (Guards, die nicht nur von der verfügbaren Ausführungszeit, sondern auch von den berechneten Ergebnissen einzelner Bausteine sowie Aktionen abhängen) erschwert bis

unmöglich. Eine Berücksichtigung der tatsächlichen Logik der Funktionsbausteine und der ISSCs würde zusätzlich die Anwendungsszenarien des Model-Checkers erweitern. Eine besondere Herausforderung bei der Erweiterung des Modells wird die richtige Balance zwischen der Aussagestärke und der Beschreibungskomplexität sein.

Eine weitere Verfeinerung der Modellierung kann darüber hinaus die genauere Erfassung der einzelnen POU-Laufzeiten sein. Eine Möglichkeit wäre dabei die stochastische Betrachtung der Ausführungszeiten, beispielsweise durch eine empirisch oder analytisch ermittelte Wahrscheinlichkeitsverteilung der erwarteten Ausführungszeit. Der Wechsel in die Domäne der stochastischen Analyse erfordert eine Untersuchung der verfügbaren Tools bzw. Möglichkeiten der Modellierung und Abgleich dieser mit den domänenspezifischen Anforderungen der Leittechnik.

Die automatische Berechnung der Guards der ISSCs beschränkt sich bislang auf die Zusicherung des „sicheren Betretens“ der einzelnen Zustände bzw. Zweige. Als Ergebnis einer besseren Modellierung des Zeitverhaltens des Systems bzw. der Anwendung könnte ein Algorithmus entstehen, der komplexe Bedingungen durch die Guards beschreiben kann. Somit wäre der erste Grundstein für ein Assistenzsystem für das ISSC-Engineering gelegt.

Die online Strategien des Systemschedulers können weiter ausgebaut werden. Ein Beispiel für eine solche Erweiterung wäre die Ausführung sporadischer Tasks. Die aktuelle Ausführungsstrategie des Prototyps lässt Raum für Verbesserung. In Abbildung 7.14 ist es deutlich sichtbar, dass diese Strategie die Slackzeit nicht vollständig ausnutzen kann. Die Tatsache wird insbesondere im letzten Zyklus der Transaktionsausführung deutlich. Eine bessere Strategie bedarf der Modellierung zusätzlicher Eigenschaften der Tasks. So kann beispielsweise gekennzeichnet werden, ob ein Task beliebig viel Slackzeit verbrauchen kann und ob dieser mehrfach innerhalb eines Grundzyklus ausgeführt werden darf.

Die effiziente Konstruktion der offline Tabellen für den Scheduler wirft weitere Forschungsfragen auf. Eine davon wäre die Suche nach geeigneten Heuristiken, die das effiziente Erstellen der Scheduling-Tabellen auf dem Laufzeitsystem ermöglichen. Die Analyse der Heuristiken muss neben der Laufzeit auch die Güte bzw. die Korrektheit der erstellten Lösungen in Betracht ziehen.

Anhänge

A. GAMS-Instanz für die offline Phase des Komponentenschedulers

Dieses Modell implementiert das Beispiel aus Abschnitt 6.3.2. Die Probleminstanz beinhaltet der Vollständigkeit halber den Aufbau für die komplette Problemformulierung aus diesem Abschnitt, der teilweise von der Instanz des Beispiels ungenutzt bleibt.

Listing 1: example.gms

```
*allow empty sets
$onempty

scalar cyctime /100/;

$if not set hypercycle $set hypercycle 6
scalar hypercycle /%hypercycle%/;

$if not set tasks $set tasks 3
scalar tasks /%tasks%/;

$eval timeInstants %tasks%*%hypercycle%
scalar timeInstants /%timeInstants%/;

SETS
 T tasks          /t1,t2,t3/
 A(T) phase-aligned tasks //
* A(T) phase-aligned tasks /t1,t2,t3/
 F(A) phase-fixed tasks //
 R(T) release-deadline //
 C hypercycle    /1*%hypercycle%/;

ALIAS(T,U);
ALIAS(C,D);

*periods are multiplied with job size

PARAMETERS
  w(T) wcet time of task t
  /       t1      90
          t2      90
          t3      30
  /
```

```
  prio(T) priority of the task (evaluated only by the exponential cost
     ↪ function)
  /       t1      1
          t2      1
          t3      1
  /
  period(T) period
  /       t1      6
          t2      2
          t3      3
  /
  phase(F) fixed phase
  /
  /
  release(R) release of task t in cases (first slot to be executed)
  /
  /
  deadline(R) deadline of task t in cases (last slot to be executed)
  /
  /
  pred(T,T) execution dependencies (only for tasks with same period)
  /
  /;

VARIABLES
    x(c,t)  true iff task t is scheduled in cycle c
*    add(c,t)  additional time for each task to run (only used for the
   ↪ exponential cost function)
    Z total cost
  BINARY VARIABLE x
  POSITIVE VARIABLE add ;

EQUATIONS
        PREDECESSOR(C,U,T)                      ordering
        FIXEDPHASE(F)                           release fixed phase
           ↪ task
    PHASEALIGN(C,A)                             phase alignment
    OVERALL(T)                  every task is executed a planned
       ↪ number of times
    ONCEAPERIOD(C,T)                    every task is executed once a
       ↪ period
    REL(C,R)                    respect release time
    DEAD(C,R)                   respect deadline
    OVERRUN(C)                  prevent overrun
    COST;
  PREDECESSOR(C,U,T) .. SUM(D$(pred(u,t) EQ 1 AND ord(d) GE (ord(c)-1)
     ↪ *period(t)+1 and ord(d) LE ord(c)*period(t)), ord(d)*x(d,u) -
     ↪ ord(d)*x(d,t)) =L= 0;
  FIXEDPHASE(F) .. SUM(C$(ord(c) EQ phase(f)+1), x(c,f)) =E= 1;
  PHASEALIGN(C,A) .. SUM(D$(ord(d) GE (ord(c)-1)+1 AND ord(d) LE ord(c
```

```
      ↪ ) AND ord(d)+period(a) LE hypercycle), x(d,a)) =E= SUM(D$(ord(
      ↪ d) GE (ord(c)-1)+1 and ord(d) LE ord(c) AND ord(d)+period(a)
      ↪ LE hypercycle), x(d+period(a),a));
  OVERALL(T) ..     SUM(D, x(d,t)) =E=  hypercycle/period(t);
  ONCEAPERIOD(C,T) ..     SUM(D$(ord(d) GE (ord(c)-1)*period(t)+1 and
      ↪ ord(d) LE ord(c)*period(t)), x(d,t)) =L=  1;
  REL(C,R) ..             SUM(D$(ord(d) GE (ord(c)-1)*period(r)+1 and
      ↪ ord(d) LE (ord(c)*period(r)) and ord(d) LT (ord(c)-1)*period(r
      ↪ )+1+(release(r)-1)), X(d,r)) =E= 0;
  DEAD(C,R) ..            SUM(D$(ord(d) GE (ord(c)-1)*period(r)+1 and
      ↪ ord(d) LE (ord(c)*period(r)) and ord(d) GE (ord(c)-1)*period(r
      ↪ )+1+deadline(r)), X(d,r)) =E= 0;
  OVERRUN(C) .. SUM(T, x(c,t)*w(t)) =L= cyctime;
* this overrun equation system should only be used for the exponential
    ↪  cost function
*  OVERRUN(C) .. SUM(T, x(c,t)*w(t)) + SUM(T, x(c,t)*add(c,t)) =L=
    ↪ cyctime;
  COST ..    Z =E= SUM((T,C), ord(c)*x(c,t));
*  COST .. Z =E= SUM(C, power[[SUM(T, x(c,t)*w(t))] - [SUM((T,D), (x(d
    ↪ ,t)*w(t))/Card(C))],2]);
*  COST ..    Z =E= SUM((T,C), x(c,t)*prio(t)*system.exp((-1) * add(c,t
    ↪ )));
MODEL SCHED /ALL/;
  SOLVE SCHED USING MIP MINIMIZING Z;
*SOLVE SCHED USING MINLP MINIMIZING Z;
*SOLVE SCHED USING MIQCP MINIMIZING Z;
```

B. ACPLT/OV Modelldateien für die Referenzimplementierung

Folgende OV-Modelle definieren die Klassen und die Assoziationen des zeitsensitiven hierarchischen Schedulers (Bibliothek „adaptiveShed“) und der ISSCs (Bibliothek „issc“).

Listing 2: adaptiveShed.ovm

```
#include "ov.ovm"
#include "fb.ovm"

LIBRARY adaptiveShed
        VERSION   = "V0.1";
        AUTHOR    = "Sten Gruener";
        COPYRIGHT = "";
        COMMENT   = "";

    CLASS cycle : CLASS ov/domain
        IS_INSTANTIABLE;
        VARIABLES
            staticSlack: TIME_SPAN HAS_GET_ACCESSOR FLAGS = "o";
        END_VARIABLES;
```

```
    OPERATIONS
    END_OPERATIONS;
END_CLASS;

    /** A structure to hold the contents of the executed Task */
CLASS task : CLASS ov/domain
    IS_INSTANTIABLE;
    COMMENT = "Adaptive Task";
    VARIABLES
            pActiveCycle: C_TYPE <OV_INSTPTR_adaptiveShed_cycle>
               ↪ FLAGS = "hi" COMMENT = "pointer to the active
               ↪ cycle";
    END_VARIABLES;
    OPERATIONS
            startup: C_FUNCTION <OV_FNC_STARTUP>;
            constructor: C_FUNCTION <OV_FNC_CONSTRUCTOR>;
    END_OPERATIONS;
END_CLASS;

/** A singleton to represent scheduler interface **/
CLASS manager : CLASS ov/object
    IS_INSTANTIABLE;
    COMMENT = "Adaptive Task manager";
    VARIABLES
        activeTask : STRING HAS_SET_ACCESSOR HAS_GET_ACCESSOR
           ↪ FLAGS = "i" COMMENT = "currently executed task";
        pActiveTask: C_TYPE <OV_INSTPTR_adaptiveShed_task> FLAGS =
           ↪  "hi" COMMENT = "pointer to the active task";
        activeTransaction: STRING HAS_SET_ACCESSOR
           ↪ HAS_GET_ACCESSOR FLAGS = "i" COMMENT = "transaction
           ↪ to execute";
        activeTransactionW: TIME_SPAN HAS_SET_ACCESSOR
           ↪ HAS_GET_ACCESSOR FLAGS = "i" COMMENT = "guaranteed
           ↪ time for a transaction to execute";
        pActiveTransaction: C_TYPE <OV_INSTPTR_ov_object> FLAGS =
           ↪ "hi" COMMENT = "pointer to the executed transaction
           ↪ ";
        nextTask: STRING HAS_SET_ACCESSOR HAS_GET_ACCESSOR FLAGS =
           ↪  "i" COMMENT = "task to swtich to asap";
        cyctime: TIME_SPAN HAS_SET_ACCESSOR HAS_GET_ACCESSOR FLAGS
           ↪  = "i" COMMENT = "cycle time";
        proctime: TIME HAS_SET_ACCESSOR HAS_GET_ACCESSOR FLAGS = "
           ↪ i" COMMENT = "next execution time";
        detachUrTask: BOOL HAS_SET_ACCESSOR HAS_GET_ACCESSOR FLAGS
           ↪  = "i" COMMENT = "trigger to detach UrTask";
        detachRootCommTask: BOOL HAS_SET_ACCESSOR HAS_GET_ACCESSOR
           ↪  FLAGS = "i" COMMENT = "trigger to detach
           ↪ RootCommTask";
    END_VARIABLES;
    OPERATIONS
```

```
            constructor: C_FUNCTION <OV_FNC_CONSTRUCTOR>;
            startup: C_FUNCTION <OV_FNC_STARTUP>; //registers the
                ↪ execution with ov
            shutdown: C_FUNCTION <OV_FNC_SHUTDOWN>;
            destructor: C_FUNCTION <OV_FNC_DESTRUCTOR>;
        END_OPERATIONS;
    END_CLASS;

    CLASS slot : CLASS ov/domain
        IS_INSTANTIABLE;
        VARIABLES
            executedObject: STRING HAS_SET_ACCESSOR HAS_GET_ACCESSOR
                ↪ FLAGS = "i" COMMENT = "object to execute e.g. an fb/
                ↪ task";
            pExecutedObject: C_TYPE <OV_INSTPTR_ov_object> FLAGS = "hi
                ↪ " COMMENT = "pointer to the executed object";
            w: TIME_SPAN HAS_ACCESSORS FLAGS = "i";
            relStartTime: TIME_SPAN HAS_ACCESSORS FLAGS = "i";
            relEndTime: TIME_SPAN HAS_ACCESSORS FLAGS = "i";
            flexible: BOOL HAS_ACCESSORS FLAGS = "i" INITIALVALUE=TRUE
                ↪ ;
        END_VARIABLES;
        OPERATIONS
            startup: C_FUNCTION <OV_FNC_STARTUP>;
        END_OPERATIONS;
    END_CLASS;

    CLASS functionblock : CLASS fb/functionblock
        OPERATIONS
            timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>
                ↪ IS_ABSTRACT;
            timedExecute: C_FUNCTION <AS_FNC_TIMEDEXECUTE>;
            typemethod: C_FUNCTION <FB_FNC_TYPEMETHOD>;
        END_OPERATIONS;
    END_CLASS;

    CLASS functionchart : CLASS adaptiveShed/functionblock
        IS_INSTANTIABLE;
        FLAGS = "i";
        COMMENT = "function chart";
        PARTS
            intask: CLASS fb/task;
        END_PARTS;
        OPERATIONS
            typemethod: C_FUNCTION <FB_FNC_TYPEMETHOD>;
            timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
            getport: C_FUNCTION <FB_FNC_GETPORT>;
            setport: C_FUNCTION <FB_FNC_SETPORT>;
        END_OPERATIONS;
    END_CLASS;
```

```
    CLASS demoFb : CLASS adaptiveShed/functionblock
        IS_INSTANTIABLE;
        VARIABLES
        END_VARIABLES;
        OPERATIONS
            timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        END_OPERATIONS;
    END_CLASS;

        ASSOCIATION variables: ONE_TO_MANY
        PARENT timedfunchart: CLASS adaptiveShed/functionchart;
        CHILD  ports: CLASS fb/port;
    END_ASSOCIATION;
END_LIBRARY;
```

Listing 3: adaptiveShed.ovf

```
#ifndef adaptiveShed_OVF_INCLUDED
#define adaptiveShed_OVF_INCLUDED

typedef OV_DLLFNCEXPORT void AS_FNC_TIMEDTYPEMETHOD(
    OV_INSTPTR_adaptiveShed_functionblock pfb,
    OV_TIME                              *pltc,
    OV_TIME_SPAN                         dCurr
);

typedef OV_DLLFNCEXPORT void AS_FNC_TIMEDEXECUTE(
    OV_INSTPTR_adaptiveShed_functionblock  pfb,
    OV_TIME                              *pltc,
    OV_TIME_SPAN                         dCurr
);

#endif
```

Listing 4: issc.ovm

```
#include "ov.ovm"
#include "adaptiveShed.ovm"
#include "ksbase.ovm"

LIBRARY issc
        VERSION   = "V0.1";
        AUTHOR    = "Sten Gruener";
        COPYRIGHT = "";
        COMMENT   = "";

        CLASS incycleSequentialStateChart : CLASS adaptiveShed/
            ↪ functionchart
                IS_INSTANTIABLE;
                PARTS
```

```
        transConds: CLASS ov/domain;
        taskActiveStep: CLASS fb/task;     //only the active step
            ↪ is linked here
    END_PARTS;
         OPERATIONS
        constructor: C_FUNCTION  <OV_FNC_CONSTRUCTOR>;
                    timedTypemethod: C_FUNCTION <
                        ↪ AS_FNC_TIMEDTYPEMETHOD>;
         END_OPERATIONS;
    END_CLASS;

    CLASS abstractStep : CLASS adaptiveShed/functionblock
    COMMENT = "abstractStep";
    VARIABLES
    END_VARIABLES;
    PARTS
       entry    : CLASS fb/task;
    END_PARTS;
    OPERATIONS
       constructor: C_FUNCTION  <OV_FNC_CONSTRUCTOR>;
       timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
    END_OPERATIONS;
END_CLASS;

    CLASS step : CLASS issc/abstractStep
    IS_INSTANTIABLE;
    COMMENT = "step";
    VARIABLES
       internalRole: UINT HAS_GET_ACCESSOR IS_DERIVED FLAGS = "n"
          ↪ COMMENT = "internal role (0 start, 1 normal, 999 end)
          ↪ ";
    END_VARIABLES;
        OPERATIONS
        END_OPERATIONS;
      END_CLASS;

 CLASS tGe : CLASS adaptiveShed/functionblock
    IS_INSTANTIABLE;
    COMMENT = "Returns true iff the remaining execution time is
       ↪ greater or equal to the queried one";
    VARIABLES
        query: TIME_SPAN HAS_ACCESSORS FLAGS = "i";
        result: BOOL HAS_ACCESSORS FLAGS = "o";
    END_VARIABLES;
    OPERATIONS
            constructor: C_FUNCTION  <OV_FNC_CONSTRUCTOR>;
        timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
    END_OPERATIONS;
END_CLASS;
```

```
    CLASS breakpoint : CLASS issc/abstractStep
   IS_INSTANTIABLE;
   COMMENT = "breakpoint";
   VARIABLES
   END_VARIABLES;
   OPERATIONS
   END_OPERATIONS;
 END_CLASS;

CLASS historyBreakpoint : CLASS issc/breakpoint
   IS_INSTANTIABLE;
   COMMENT = "history breakpoint";
   VARIABLES
   END_VARIABLES;
   OPERATIONS
   END_OPERATIONS;
 END_CLASS;

      CLASS transition: CLASS adaptiveShed/functionblock
        IS_INSTANTIABLE;
        COMMENT = "transition";
        VARIABLES
          result: BOOL FLAGS = "i" COMMENT = "The result of the
              ↪ executed transition";
          visuallayoutPrev: STRING FLAGS = "p" COMMENT = "visual
              ↪ layout information for HMI";
          visuallayoutNext: STRING FLAGS = "p" COMMENT = "visual
              ↪ layout information for HMI";
        END_VARIABLES;
        OPERATIONS
      constructor:   C_FUNCTION <OV_FNC_CONSTRUCTOR>;
      timedTypemethod:   C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        END_OPERATIONS;
      END_CLASS;

     CLASS actionBlock : CLASS adaptiveShed/functionblock
        COMMENT = "action block";
        VARIABLES
          w          : TIME_SPAN HAS_ACCESSORS FLAGS = "p"
              ↪ COMMENT = "WCET of the action";
        END_VARIABLES;
        OPERATIONS
          constructor: C_FUNCTION <OV_FNC_CONSTRUCTOR>;
       timedTypemethod: C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        END_OPERATIONS;
      END_CLASS;

      CLASS setVariable : CLASS issc/actionBlock
        IS_INSTANTIABLE;
        COMMENT = "action block for set variable action";
```

```
        VARIABLES
          variable : STRING         HAS_SET_ACCESSOR FLAGS = "p"
            ↪    COMMENT = "The variable name (relative to the
            ↪ ISSC or absolute path)";
          value              : ANY          HAS_SET_ACCESSOR FLAGS
            ↪  = "i"    COMMENT = "The value to be set";
        END_VARIABLES;
        OPERATIONS
          timedTypemethod:   C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        END_OPERATIONS;
      END_CLASS;

    CLASS execute : CLASS issc/actionBlock
      IS_INSTANTIABLE;
      COMMENT = "action block executing a function block";
      VARIABLES
        executedObject: STRING HAS_SET_ACCESSOR FLAGS = "p" COMMENT
          ↪ = "The object name (relative to the ISSC or absolute
          ↪ path)";
        pExecutedObject: C_TYPE <OV_INSTPTR_ov_object> FLAGS = "hi"
          ↪ COMMENT = "pointer to the executed object";
      END_VARIABLES;
      OPERATIONS
        timedTypemethod:   C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        startup: C_FUNCTION <OV_FNC_STARTUP>;
      END_OPERATIONS;
    END_CLASS;

    CLASS executeTg : CLASS issc/actionBlock
      IS_INSTANTIABLE;
      COMMENT = "action block executing a time guard";
      VARIABLES
        executedObject  : STRING HAS_SET_ACCESSOR FLAGS = "p"
          ↪ COMMENT = "The object name (relative to the ISSC or
          ↪ absolute path)";
        pExecutedObject : C_TYPE <OV_INSTPTR_ov_object> FLAGS = "hi"
          ↪  COMMENT = "pointer to the executed object";
      END_VARIABLES;
      OPERATIONS
        timedTypemethod :   C_FUNCTION <AS_FNC_TIMEDTYPEMETHOD>;
        startup      : C_FUNCTION <OV_FNC_STARTUP>;
      END_OPERATIONS;
    END_CLASS;

        /* Connections from steps to next transitions. */
        ASSOCIATION nextTransitions : ONE_TO_MANY
          IS_LOCAL;
          PARENT prevStep  : CLASS issc/abstractStep;
          CHILD  nextTrans : CLASS issc/transition;
        END_ASSOCIATION;
```

```
        /* Connections from transitions to next steps. */
        ASSOCIATION previousTransitions : ONE_TO_MANY
          IS_LOCAL;
          PARENT nextStep  : CLASS issc/abstractStep;
          CHILD  prevTrans : CLASS issc/transition;
        END_ASSOCIATION;
END_LIBRARY;
```

Literaturverzeichnis

[3S-15a] 3S-Smart Software Solutions: *CODESYS Control RTE V3 Handbuch*. https://www.codesys.com/download/download-center.html. Version: Juli 2015, Abruf: 01.03.2016 3.3.1

[3S-15b] 3S-Smart Software Solutions: *CODESYS Runtime (Brochure)*. https://www.codesys.com/products/codesys-runtime.html. Version: November 2015, Abruf: 01.03.2016 3.3.1

[Abe06] Abel, Dirk: *Rapid control prototyping : Methoden und Anwendungen ; mit 16 Tabellen*. Springer, 2006. – ISBN 978–3–540–29524–2 6.2.1

[AD94] Alur, Rajeev ; Dill, David L.: A theory of timed automata. In: *Theoretical computer science* 126 (1994), Nr. 2, S. 183–235 2.1.4

[Alb03] Albrecht, Harald: *Fortschrittberichte VDI : Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 975: *On Meta-Modeling for Communication in Operational Process Control Engineering*. Düsseldorf : VDI-Verlag, 2003. – ISBN 3183975084 2.2.7, 3.3.4

[AMM04] *Kapitel* Periodic Reward-Based Scheduling and Its Application to Power-Aware Real-Time Systems. In: Aydin, Hakan ; Melhem, Rami ; Mosse, Daniel: Chapman and Hall/CRC, 2004 (Chapman & Hall/CRC Computer & Information Science Series). – ISBN 978–1–58488–397–5 5.1.3, 5.1.5

[App90] Appel, Andrew W.: A runtime system. In: *Lisp and Symbolic Computation* 3 (1990), Nr. 4, S. 343–380 2.2.1

[Art12] Artigues, Christian: *Scheduling and (Integer) Linear Programming*. Nantes : Master Class CPAIOR 2012, 2012 6.3.2

[ASS97] Ahrens, Wolfgang ; Scheurlen, Hans-Joachim ; Spohr, Gerd-Ulrich: *Informations-orientierte Leittechnik*. Oldenbourg, 1997. – ISBN 3486234749 2.1.1

[ATV08] Abed, Nazha ; Tripakis, Stavros ; Vincent, Jean-Marc: Resource-aware verification using randomized exploration of large state spaces. In: *Lecture Notes in Computer Science* 5156 (2008), S. 214–231 5.1.3

[BA02] Buttazzo, Giorgio ; Abeni, Luca: Adaptive Workload Management Through Elastic Scheduling. In: *Real-Time Syst.* 23 (2002), Nr. 1/2, 7–24. http://dx.doi.org/10.1023/A:1015342318358. – DOI 10.1023/A:1015342318358. – ISSN 0922–6443 5.2.1, 5.2.3

[Bau04] Bauer, Nanette: *Formale Analyse von Sequential Function Charts*. 2004. – ISBN 3–8322–2298–7 3.4.1

[BBE+11] Bini, Enrico ; Buttazzo, Giorgio ; Eker, Johan ; Schorr, Stefan ; Guerra, Raphael ; Fohler, Gerhard ; Arzen, Karl-Erik ; Romero, Vanessa ; Scordino, Claudio: Resource Management on Multicore Systems: The ACTORS Approach. In: *Micro, IEEE* 31 (2011), Mai, Nr. 3, S. 72–81. http://dx.doi.org/10.1109/MM.2011.1. – DOI 10.1109/MM.2011.1. – ISSN 0272–1732 5.1.3

[BBF09] Blair, Gordon ; Bencomo, Nelly ; France, Robert B.: Models@run.time. In: *Computer* 42 (2009), Oktober, Nr. 10, S. 22–27. http://dx.doi.org/10.1109/MC.2009.326. – DOI 10.1109/MC.2009.326. – ISSN 0018–9162 2.2.2

[BBY13] Buttazzo, Giorgio C. ; Bertogna, Marko ; Yao, Gang: Limited Preemptive Scheduling for Real-Time Systems. A Survey. In: *Industrial Informatics, IEEE Transactions on* 9 (2013), Februar, Nr. 1, S. 3–15. http://dx.doi.org/10.1109/TII.2012.2188805. – DOI 10.1109/TII.2012.2188805. – ISSN 1551–3203 2.1.2

[BCK98] Bates, I. D. ; Chester, E. G. ; Kinniment, D. J.: A case study in the automatic programming of a PLC based control system using Statemate statecharts. In: *Control '98. UKACC International Conference on (Conf. Publ. No. 455)* Bd. 1, 1998. – ISSN 0537–9989, S. 832–837 vol.1 3.4.1

[BDL04] Behrmann, Gerd ; David, Alexandre ; Larsen, Kim G.: A Tutorial on Uppaal. In: Bernardo, Marco (Hrsg.) ; Corradini, Flavio (Hrsg.): *Formal Methods for the Design of Real-Time Systems* Bd. 3185. Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–23068–7, S. 200–236 2.1.4

[BF11] Baruah, Sanjoy ; Fohler, Gerhard: Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, 2011. – ISSN 1052–8725, S. 3–12 6.3.3

[BH09] Benra, Juliane T. (Hrsg.) ; Halang, Wolfgang A. (Hrsg.): *Software-Entwicklung für Echtzeitsysteme*. Springer Science + Business Media, 2009. http://dx.doi.org/10.1007/978-3-642-01596-0. http://dx.doi.org/10.1007/978-3-642-01596-0 2.1.2

[BMR+96] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael ; Stal, Michael: *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996. – ISBN 0471958697 2.2.1

[BPR01] Bellifemine, Fabio ; Poggi, Agostino ; Rimassa, Giovanni: JADE: a FIPA2000 compliant agent development environment. In: *Proceedings of the fifth international conference on Autonomous agents* ACM, 2001, S. 216–217 2.1.2

[But11] Buttazzo, Giorgio: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)*. Springer, 2011. – ISBN 1461406757 2.1.2, 2.1.2, 5.1.5, 5.2.1, 5.2

[BZXN02] Brennan, Robert W. ; Zhang, Xiaokun ; Xu, Yuefei ; Norrie, Douglas H.: A Reconfigurable Concurrent Function Block Model and Its Implementation in Real-time Java. In: *Integr. Comput.-Aided Eng.* 9 (2002), Nr. 3, 263–279. http://dl.acm.org/citation.cfm?id=1275687.1275693. – ISSN 1069–2509 3.2.4

[CMM98] Czyzyk, Joseph ; Mesnier, Michael P. ; Moré, Jorge J.: The NEOS Server. In: *Computing in Science and Engineering* 5 (1998), Nr. 3, S. 68–75. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/99.714603. – DOI http://doi.ieeecomputersociety.org/10.1109/99.714603. – ISSN 1070–9924 6.3.2

[CNYM12] Chung, Lawrence ; Nixon, Brian A. ; Yu, Eric ; Mylopoulos, John: *Non-functional requirements in software engineering*. Bd. 5. Springer Science & Business Media, 2012 4.2.2

[DB88] Dean, Thomas L. ; Boddy, Mark S.: An Analysis of Time-Dependent Planning. In: *AAAI* Bd. 88, 1988, S. 49–54 5.1.2

[DDV14] Dai, Wenbin ; Dubinin, Victor N. ; Vyatkin, Valeriy: Migration From PLC to IEC 61499 Using Semantic Web Technologies. In: *IEEE Transactions on Systems, Man, and Cybernetics* 44 (2014), März, Nr. 3, S. 277–291. http://dx.doi.org/10.1109/TSMCC.2013.2264671. – DOI 10.1109/TSMCC.2013.2264671. – ISSN 2168–2216 2.2.6, 6.1.1, 6.1.2

[DIN14] DIN Deutsches Institut für Normung e. V.: *DIN SPEC 40912: Kernmodelle – Beschreibung und Beispiele*. 2014-11-00. 2014 3.2.1

[DIN16] DIN Deutsches Institut für Normung e. V.: *DIN SPEC 91345: Referenzarchitekturmodell Industrie 4.0 (RAMI4.0)*. 2016-04. 2016 1.1

[DMES14] Diedrich, Christian ; Meyer, Matthias ; Evertz, Lars ; Schäfer, Wilhelm: Dienste in der Automatisierungstechnik : Automatisierungsgeräte werden I40-Komponenten. In: *Atp-Edition : automatisierungstechnische Praxis* 12 (2014), 24-35. http://publications.rwth-aachen.de/record/459738. – ISSN 0178–2320 3.2.1

[Dol01] Dolan, Elizabeth D.: NEOS Server 4.0 Administrative Guide / Technical Memorandum ANL/MCS-TM-250, Argonne National Laboratory, Argonne, IL. 2001. – Forschungsbericht 6.3.2

[DSSG+08] De Souza, Luciana Moreira S. ; Spiess, Patrik ; Guinard, Dominique ; Köhler, Moritz ; Karnouskos, Stamatis ; Savio, Domnic: SOCRADES: A web service based shop floor integration infrastructure. In: *The internet of things*. Springer, 2008, S. 50–67 3.2.1

[ECW92] Estivill-Castro, Vladmir ; Wood, Derick: A survey of adaptive sorting algorithms. In: *ACM Computing Surveys (CSUR)* 24 (1992), Nr. 4, S. 441–476 5.1.1

[EE13] Evertz, Lars ; Epple, Ulrich: Laying a basis for service systems in process control. In: *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, 2013. – ISSN 1946–0740, S. 1–8 3.2.1

[Epp00] Epple, Ulrich: Agentensysteme in der Leittechnik. In: *atp Automatisierungstechnische Praxis* 42 (2000), Nr. 8, S. 42–51 3.2.2

[Epp12] Epple, Ulrich: Increasing flexibility and functionality in industrial process control: The helpful usage of models, services and cybernetic principles. In: *Systems, Signals and Devices (SSD), 2012 9th International Multi-Conference on*, 2012, S. 1–4 4.2.1

[Epp13] Epple, Ulrich: Agentenmodelle in der Anlagenautomation. In: Göhner, Peter (Hrsg.): *Agentensysteme in der Automatisierungstechnik*. Springer Berlin Heidelberg, 2013 (Xpert.press). – ISBN 978–3–642–31767–5, S. 95–110 3.2.2

[FDGV13] Ferranti, Ettore ; Domova, Veronika ; Gooijer, Thijmen de ; Vulgarakis, Aneta: 4DIAC integration into the FASA project: a success story of increased maintainability and modularity. In: *4th 4DIAC User's Workshop held in conjunction with the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2013)*, 2013 3.3.3

[FDT+15] Fay, Alexander ; Diedrich, Christian ; Thron, Mario ; Andre, Scholz ; Schmidt, Philipp ; Ladiges, Jan ; Holm, Thomas: Wie bekommt Industrie 4.0 Bedeutung? In: *Atp-Edition* 57 (2015), Nr. 7/8, S. 31–43. – ISSN 0178–2320 6.1.2

[Fel01] Felleisen, Michael: *Prozessleittechnik für die Verfahrensindustrie*. München : Oldenbourg, 2001. – ISBN 978–3486270129 2.1, 4.1, 4.1, 4.2.2

[FLR+13] Feldmann, Stefan ; Loskyll, Matthias ; Rosch, Susanne ; Schlick, Jochen ; Zuhlke, Detlef ; Vogel-Heuser, Birgit: Increasing agility in engineering and runtime of automated manufacturing systems. In: *Industrial Technology (ICIT), 2013 IEEE International Conference on* IEEE, 2013, S. 1303–1308 3.2.1, 3.2.3

[Foh93] Fohler, Gerhard: Changing Operational Modes in the Context of Pre Run-Time Scheduling. In: *IEICE Transactions on Information and Systems* Special Issue on Responsive Computer Systems (1993), November 6.3.3

[Foh12] Fohler, Gerhard ; Chakraborty, Samarjit (Hrsg.): *Advances in Real-Time Systems, Chapter Predictable Flexible Real-time Systems*. SPRINGER, 2012 2.1.2, 2.1.2, 2.1.2, 5.1.6, 6.3.2

[FT11] Frey, Georg ; Thramboulidis, Kleanthis: Einbindung der IEC 61131 in modellgetriebene Entwicklungsprozesse. In: *Tagungsband AUTOMATION 2011* (2011) 6.1.4

[FVHF+15] Fay, Alexander ; Vogel-Heuser, Birgit ; Frank, Timo ; Eckert, Karin ; Hadlich, Thomas ; Diedrich, Christian: Enhancing a model-based engineering approach for distributed manufacturing automation systems with characteristics and design patterns. In: *Journal of Systems and Software* 101 (2015), März, 221–235. http://dx.doi.org/10.1016/j.jss.2014.12.028. – DOI 10.1016/j.jss.2014.12.028 4.1

[GE13a] Grüner, Sten ; Epple, Ulrich: Konzeption eines ebenenübergreifenden Laufzeitsystems für die Automation. In: *Automation 2013 : 14. Branchentreff der Mess- und Automatisierungstechnik ; Kongresshaus Baden-Baden, 25. und 26. Juni 2013 / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik* Bd. 2209. Düsseldorf : VDI-Verl., 2013 (VDI-Berichte). – ISBN 978–3–18–092209–6, S. 7–10 3.1

[GE13b] Grüner, Sten ; Epple, Ulrich: Paradigms for unified runtime systems in industrial automation. In: *Control Conference (ECC), 2013 European*, 2013, S. 3925–3930 2.2.7, 3.1, 3.3.4, 3.4.1, 6.1, 6.1.2, 6.2, 6.1.4, 6.1.6

[GE14] Grüner, Sten ; Epple, Ulrich: Regelbasiertes Engineering mit Graphabfragen. In: *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme X, Schloss Dagstuhl, Germany, März 05-07, 2014*. München : fortiss GmbH, 2014, S. 105–111 3.1

[GE15] Grüner, Sten ; Epple, Ulrich: Real-time properties for design of heterogeneous industrial automation systems. In: *Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE*, 2015, S. 004500–004505 3.1, 6.1.5

[GE16] Grüner, Sten ; Epple, Ulrich: Adaptive Laufzeiteigenschaften von Anwendungen in der Automation : Anforderungen und Nutzungsperspektiven. In: *Automation 2016 : 17. Branchentreff der Mess- und Automatisierungstechnik : secure & reliable in the digital world : Baden-Baden, 07. und 08. Juni 2016 / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*. Düsseldorf : VDI Verlag GmbH, Jun 2016 3.1

[GGH12] Guo, Yike ; Ghanem, M. ; Han, Rui: Does the Cloud need new algorithms? An introduction to elastic algorithms. In: *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, 2012, S. 66–73 5.1, 5.1.3, 5.1.4

[GKE12] Grüner, Sten ; Kampert, David ; Epple, Ulrich: A Model-Based Implementation of Function Block Diagram. In: *Tagungsband / Dagstuhl-Workshop MBEES (MBEES 2012): modellbasierte Entwicklung eingebetteter Systeme VIII; Model-Based Development of Embedded Systems; 6.02.2012 - 8.02.2012*. München : fortiss, 2012, S. 81–90 3.1

[Gli07] Glinz, Martin: On non-functional requirements. In: *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International* IEEE, 2007, S. 21–26 4.2.1, 4.2.2

[GM97] Gropp, William ; Moré, Jorge: Optimization environments and the NEOS server. In: *Approximation theory and optimization* (1997), S. 167–182 6.3.2

[GPP15] Grüner, Sten ; Pfrommer, Julius ; Palm, Florian: A RESTful extension of OPC UA. In: *Factory Communication Systems (WFCS), 2015 IEEE World Conference on*, 2015, S. 1–4 3.1

[GPP16] Grüner, Sten ; Pfrommer, Julius ; Palm, Florian: RESTful Industrial Communication with OPC UA. In: *IEEE Transactions on Industrial Informatics* PP (2016), Nr. 99, S. 1–1. `http://dx.doi.org/10.1109/TII.2016.2530404`. – DOI 10.1109/TII.2016.2530404. – ISSN 1551–3203 3.1

[Gra96] Grass, Joshua: Reasoning About Computational Resource Allocation. In: *Crossroads* 3 (1996), September, Nr. 1, 16–20. `http://dx.doi.org/10.1145/332148.332154`. – DOI 10.1145/332148.332154. – ISSN 1528–4972 5.1.2

[Gro09] Gronback, Richard C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1. Addison-Wesley Professional, 2009. – ISBN 0321534077, 9780321534071 6.2.3

[GWE14] Grüner, Sten ; Weber, Peter ; Epple, Ulrich: Rule-Based Engineering Using Declarative Graph Database Queries. In: *Proceedings of IEEE 12th International Conference on Industrial Informatics (INDIN)*. Piscataway, NJ : IEEE, 2014. – ISBN 978–1–4799–4906–9, S. 274–279 2.2.4, 3.1, 3.2.3, 3.2.5, 3.3.4

[GY06] Gaber, Mohamed M. ; Yu, Philip S.: A Framework for Resource-aware Knowledge Discovery in Data Streams: A Holistic Approach with Its Application to Clustering. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2006 (SAC '06). – ISBN 1–59593–108–2, 649–656 5.1.3

[Har87] Harel, David: Statecharts: A visual formalism for complex systems. In: *Science of computer programming* 8 (1987), Nr. 3, S. 231–274 3.4.1, 6.2.3

[IEC01] *IEC 61131-5: Programmable controllers – Part 5: Communications*. International Electrotechnical Commission, November 2001 2.2.3, 2.2.5

[IEC03a] *IEC 61131-1, Programmable controllers – Part 1: General information*. International Electrotechnical Commission, Juli 2003 2.2.3, 2.5

[IEC03b] *IEC 61131-8, Ed. 2: Programmable controllers – Part 8: Guidelines for the application and implementation of programming languages*. International Electrotechnical Commission, September 2003 2.2.5, 4.2.2

[IEC10] *IEC 62541: OPC Unified Architecture Part 1-10, Release 1.0*. International Electrotechnical Commission, Februar 2010 1.1, 3.1, 3.3.4

[IEC11] *IEC 61131-3, Ed. 3: Programmable controllers – Part 3: Programming languages*. International Electrotechnical Commission, August 2011 2.2.1, 2.2.4, 2.7, 2.2.5, 6.2.1

[IEC12] *IEC 61499-1: Function blocks – Part 1: Architecture*. International Electrotechnical Commission, November 2012 2.2.6, 2.8

[IEC13] *IEC 60848: GRAFCET specification language for sequential function charts*. International Electrotechnical Commission, Februar 2013 3.4.1

[ISO05] *ISO/IEC 25000 - Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE*. International Organization for Standardization / International Electrotechnical Commission, 2005 4.2.2

[JE13] Jeromin, Holger ; Epple, Ulrich: Modellbasiertes und technologieneutrales HMI für eingebettete Komponenten. In: Giese, Holger (Hrsg.) ; Huhn, Michaela (Hrsg.) ; Phillips, Jan (Hrsg.) ; Schätz, Bernhard (Hrsg.): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, April 24-26, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, fortiss GmbH, München, 2013, 80–89 7.1

[Jor11] Jorewitz, Reiner: *Fortschrittberichte VDI : Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 1201: *Eine strukturelle Beschreibungsmethodik zur automatisierten Erzeugung von Prozessbewertungen in der operativen Prozessleittechnik*. Düsseldorf : VDI-Verlag, 2011. – ISBN 978–3–18–520108–0 4.2.2

[JS05] Jammes, François ; Smit, Haram: Service-oriented paradigms in industrial automation. In: *Industrial Informatics, IEEE Transactions on* 1 (2005), Februar, Nr. 1, S. 62–70. http://dx.doi.org/10.1109/TII.2005.844419. – DOI 10.1109/TII.2005.844419. – ISSN 1551–3203 3.2.1

[JSM91] Jeffay, Kevin ; Stanat, Donald F. ; Martel, Charles U.: On non-preemptive scheduling of period and sporadic tasks. In: *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, 1991, S. 129–139 6.3.2

[JT00] John, Karl-Heinz ; Tiegelkamp, Michael: Stärken der IEC 61131-3. In: *SPS-Programmierung mit IEC 61131-3*. Springer Berlin Heidelberg, 2000 (VDI-Buch). – ISBN 978–3–662–09993–3, S. 293–298 4.2.2

[JT09] John, Karl-Heinz ; Tiegelkamp, Michael: *SPS-Programmierung mit IEC 61131-3: Konzepte und Programmiersprachen, Anforderungen an Programmiersysteme, Entscheidungshilfen (VDI-Buch) (German Edition)*. Springer, 2009. – ISBN 3642002684 2.2.4, 2.2.5, 6.1.6

[Kal12] Kallrath, Josef: *Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis: Mit Fallstudien aus Chemie, Energiewirtschaft, Papierindustrie, Metallgewerbe, Produktion und Logistik (German Edition)*. Springer Vieweg, 2012. – ISBN 3658006897 2.1.5

[KF06] Kaiser, Robert ; Fuchsen, Rudolf: *Verfahren zur Verteilung von Rechenzeit in einem Rechnersystem*. August 2006. – DE Patent App. DE200,410,054,571 8

[KHAJ05] Koumpis, Konstantinos ; Hanna, Lesley ; Andersson, Mats ; Johansson, Magnus: Wireless industrial control and monitoring beyond cable replacement. In: *2nd Profibus International Conference*, 2005 4.1

[KM05] Kopec, Hellmut ; Maier, Uwe: Kritische Anmerkungen zu heutigen PLS. In: *atp Automatisierungstechnische Praxis* 47 (2005), Nr. 3, S. 24–28 4.2.2

[KM09] Koppermann, Claus ; Möckel, Bernd: Qualitäts- und Projektmanagement. In: Früh, Karl F. (Hrsg.) ; Maier, Uwe (Hrsg.) ; Schaudel, Dieter (Hrsg.): *Handbuch der Prozessautomatisierung*. Oldenbourg Industrieverlag, 2009. – ISBN 383563142X 2.4

[Kom06] Komoda, Norihisa: Service Oriented Architecture (SOA) in Industrial Systems. In: *Industrial Informatics, IEEE International Conference on*, 2006, S. 1–5 3.2.1

[Kop93] Kopetz, Hermann: Should responsive systems be event-triggered or time-triggered? In: *IEICE Transactions on Information and Systems* 76 (1993), Nr. 11, S. 1325–1332 2.1.2

[Kop11a] Kopetz, Hermann: *Real-Time Systems*. Springer US, 2011. `http://dx.doi.org/10.1007/978-1-4419-8237-7`. `http://dx.doi.org/10.1007/978-1-4419-8237-7` 2.1.2, 2.1.2, 2.1.2, 2.1.2, 4.1, 4.1, 5.1.2

[Kop11b] *Kapitel* 3. In: Kopetz, Hermann: *Real-Time Systems*. Springer US, 2011 6.2.1

[KPKP06] Kolovos, Dimitrios S. ; Paige, Richard F. ; Kelly, Tim ; Polack, Fiona A.: Requirements for domain-specific languages. In: *Proceedings of the First ECOOP Workshop on Domain-Specific Program Development*, 2006 6.2.3

[KQE11] Krausser, Tina ; Quiros, Gustavo ; Epple, Ulrich: An IEC-61131-based rule system for integrated automation engineering: Concept and case study. In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, 2011, S. 539–544 2.2.4, 3.2.5

[KRS12] Kowalewski, Stefan ; Rumpe, Bernhard ; Stollenwerk, André: *VDI-Berichte*. Bd. 2171: *Cyber-Physical Systems - eine Herausforderung an die Automatisierungstechnik?* Düsseldorf : VDI-Verl., 2012. – 113–116 S. `http://publications.rwth-aachen.de/record/123658`. – ISBN 978–3–18–092171–6. – Datenträger: 1 CD-ROM 2.1.3, 4.2.1

[Kru95] Kruchten, Philippe: The 4+1 View Model of Architecture. In: *IEEE Softw.* 12 (1995), November, Nr. 6, 42–50. http://dx.doi.org/10.1109/52.469759. – DOI 10.1109/52.469759. – ISSN 0740–7459 6.1.4

[KS95] Koren, Gilad ; Shasha, Dennis: Skip-Over: algorithms and complexity for overloaded systems that allow skips. In: *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, 1995. – ISSN 1052–8725, S. 110–117 5.2.3

[KWH13] Kagermann, Henning (Hrsg.) ; Wahlster, Wolfgang (Hrsg.) ; Helbig, Johannes (Hrsg.): *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0*. Berlin : Forschungsunion im Stifterverband für die Deutsche Wirtschaft e.V., 2013 3.2.1, 4.1

[Lee06] Lee, Edward A.: Cyber-physical systems - are computing foundations adequate. In: *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap* Bd. 2 Citeseer, 2006 2.1.3

[LG99] Lauber, Rudolf ; Göhner, Peter: *Prozessautomatisierung 1: Automatisierungssysteme und -strukturen, Computer- und Bussysteme für die Anlagen- und Produktautomatisierung, Echtzeitprogrammierung und Echtzeitbetriebssysteme, Zuverlässigkeits- und Sicherheitstechnik*. Springer, 1999. – ISBN 354065318X 2.1.1, 2.1.1

[LHFL14a] Ladiges, Jan ; Haubeck, Christopher ; Fay, Alexander ; Lamersdorf, Winfried: Evolution Management of Production Facilities by Semi-Automated Requirement Verification. In: *at - Automatisierungstechnik* 62 (2014), Januar, Nr. 11. http://dx.doi.org/10.1515/auto-2014-1100. – DOI 10.1515/auto–2014–1100 2.2.8

[LHFL14b] Ladiges, Jan ; Haubeck, Christopher ; Fay, Alexander ; Lamersdorf, Winfried: Semi-automated decision making support for undocumented evolutionary changes. In: *16. Workshop Software-Reengineering und -Evolution der GI-Fachgruppe Software-Reengineering (SRE)* (2014) 2.2.8

[Li12] Li, Fang: Specification of the Requirements to Support Information Technology-Cycles in the Machine and Plant Manufacturing Industry. In: Borangiu, Theodor (Hrsg.): *14th IFAC Symposium on Information Control Problems in Manufacturing*, Elsevier BV, Mai 2012 4.1

[LL73] Liu, C. L. ; Layland, James W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In: *J. ACM* 20 (1973), Nr. 1, 46–61. http://dx.doi.org/10.1145/321738.321743. – DOI 10.1145/321738.321743. – ISSN 0004–5411 2.1.2

[LLB+94] Liu, Jane W. S. ; Lin, Kwei-Jay ; Bettati, Riccardo ; Hull, David ; Yu, Albert: Use of Imprecise Computation to Enhance Dependability of Real-Time Systems. Version: 1994. http://dx.doi.org/10.1007/978-0-585-27316-7_6. In: Koob, Gary M. (Hrsg.) ; Lau, Clifford G. (Hrsg.): *Foundations of Dependable Computing* Bd. 284. Springer US, 1994.

– DOI 10.1007/978–0–585–27316–7_6. – ISBN 978–0–7923–9485–3, S. 157–182 5.1.5, 5.1, 5.1.5

[LLS+91] Liu, Jane W. ; Lin, Kwei-Jay ; Shih, Wei-Kuan ; Yu, Albert Chiang-shi ; Chung, Jen-Yao ; Zhao, Wei: Algorithms for Scheduling Imprecise Computations. In: *Computer* 24 (1991), Mai, Nr. 5, S. 58–68. http://dx.doi.org/10.1109/2.76287. – DOI 10.1109/2.76287. – ISSN 0018–9162 5.1.5

[LMP+05] Levis, Philip ; Madden, Sam ; Polastre, Joseph ; Szewczyk, Robert ; Whitehouse, Kamin ; Woo, Alec ; Gay, David ; Hill, Jason ; Welsh, Matt ; Brewer, Eric u. a.: TinyOS: An operating system for sensor networks. In: *Ambient intelligence*. Springer, 2005, S. 115–148 2.1.2

[LZVM11] Lepuschitz, Wilfried ; Zoitl, Alois ; Vallee, Mathieu ; Merdan, Munir: Toward self-reconfiguration of manufacturing systems using automation agents. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 41 (2011), Nr. 1, S. 52–69 3.2.4

[Mal09] Mall, Rajib: *Real-time systems: theory and practice*. Pearson Education India, 2009 6.1.5

[MBS+11] Mersch, Henning ; Behnen, Daniel ; Schmitz, Dominik ; Epple, Ulrich ; Brecher, Christian ; Jarke, Matthias: Gemeinsamkeiten und Unterschiede der Prozess- und Fertigungstechnik. In: *at - Automatisierungstechnik* 59 (2011), Januar, Nr. 1. http://dx.doi.org/10.1524/auto.2011.0891. – DOI 10.1524/auto.2011.0891 2.1.1, 2.1

[ME12] Mersch, Henning ; Epple, Ulrich: Concepts of service-orientation for process control engineering. In: *Systems, Signals and Devices (SSD), 2012 9th International Multi-Conference on*, 2012, S. 1–6 3.2.1

[Mea55] Mealy, George H.: A Method for Synthesizing Sequential Circuits. In: *Bell System Technical Journal* 34 (1955), Nr. 5, 1045–1079. http://dx.doi.org/10.1002/j.1538-7305.1955.tb03788.x. – DOI 10.1002/j.1538–7305.1955.tb03788.x. – ISSN 1538–7305 3.4.3

[Mer16] Mersch, Henning: *Fortschrittberichte VDI : Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 1245: *Deterministische, dynamische Systemstrukturen in der Automatisierungstechnik*. Düsseldorf : VDI-Verlag, 2016. – ISBN 3185245084 3.3.4, 7.5

[Mey02] Meyer, Dirk: *Fortschrittberichte VDI : Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 940: *Objektverwaltungskonzept für die operative Prozessleittechnik*. Düsseldorf : VDI-Verlag, 2002. – ISBN 3183940086 2.1.1, 2.2.7, 3.3.4

[MFFR02] Marti, Pau ; Fuertes, Josep M. ; Fohler, Gerhard ; Ramamritham, Krithi: Improving quality-of-control using flexible timing constraints: metric and scheduling. In: *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 2002. – ISSN 1052–8725, S. 91–100 5.2.2

[Mon14] Monostori, László: Cyber-physical production systems: Roots, expectations and R&D challenges. In: *Procedia CIRP* 17 (2014), S. 9–13 2.1.3

[Moo56] Moore, Edward F.: Gedanken-experiments on sequential machines. In: *Automata studies* 34 (1956), S. 129–153 3.4.3

[NAM02] *NE 58: Abwicklung von qualifizierungspflichtigen PLT-Projekten*. Normenarbeitsgemeinschaft für Mess- und Regeltechnik in der Chemischen Industrie, 2002 2.4, 2.2.8, 2.2.8

[NAM03] *NA 35: Abwicklung von PLT-Projekten*. Normenarbeitsgemeinschaft für Mess- und Regeltechnik in der Chemischen Industrie, 2003 2.2.8

[NAM12] *NE 141: Schnittstelle zwischen Batch- und MES-Systemen*. Normenarbeitsgemeinschaft für Mess- und Regeltechnik in der Chemischen Industrie, 2012 3.2.1

[NAM14] *NE 152: Regelgütemanagement: Überwachung und Optimierung der Basisregelung von Produktionsanlagen*. Normenarbeitsgemeinschaft für Mess- und Regeltechnik in der Chemischen Industrie, Dezember 2014 4.1

[NK12] Newman, Peter ; Kotonya, Gerald: A Runtime Resource-aware Architecture for Service-oriented Embedded Systems. In: *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, 2012, S. 61–70 5.1.3

[OAS06] OASIS: Organization for the Advancement of Structured Information Standards: *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006 3.2.1

[PCC+11] Polo, Jordà ; Castillo, Claris ; Carrera, David ; Becerra, Yolanda ; Whalley, Ian ; Steinder, Malgorzata ; Torres, Jordi ; Ayguadé, Eduard: Resource-aware Adaptive Scheduling for Mapreduce Clusters. In: *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*. Berlin, Heidelberg : Springer-Verlag, 2011 (Middleware'11). – ISBN 978–3–642–25820–6, 187–207 5.1.3

[PGP+15] Palm, Florian ; Grüner, Sten ; Pfrommer, Julius ; Graube, Markus ; Urbas, Leon: Open source as enabler for OPC UA in industrial automation. In: *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*, 2015, S. 1–6 3.1, 7.2

[PLC16] PLCopen and OPC Foundation: *OPC-UA Client FUNCTION BLOCKS for IEC61131-3 Version 1.1*. `http://www.plcopen.org/pages/tc4_communication/forms/conf-opcua.htm`. Version: März 2016, Abruf: 18.03.2016 6.1.3

[Pol85] Polke, Martin: Informationsgehalt technischer Prozesse. In: *atp Automatisierungstechnische Praxis* 27 (1985), Nr. 4, S. 161–171 2.1

[Pol94] Polke, Martin (Hrsg.): *Prozeßleittechnik*. Oldenbourg, 1994. – ISBN 3486225499 2.1.1, 2.1, 2.1.1, 2.2

[PSA+15] Pfrommer, Julius ; Stogl, Denis ; Aleksandrov, Kiril ; Navarro, Stefan E. ; Hein, Björn ; Beyerer, Jürgen: Plug & Produce by Modelling Skills and Service-Oriented Orchestration of Reconfigurable Manufacturing Systems. In: *at Automatisierungstechnik* 10 (2015), Nr. 63 3.2.1

[PSGS04] Poladian, Vahe ; Sousa, Joao P. ; Garlan, David ; Shaw, Mary: Dynamic configuration of resource-aware services. In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on* IEEE, 2004, S. 604–613 5.1.3

[PSVHM15] Priego, Rafael ; Schütz, Daniel ; Vogel-Heuser, Birgit ; Marcos, Marga: Reconfiguration architecture for updates of automation systems during operation. In: *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*, 2015, S. 1–8 3.2.5

[RJB04] Rumbaugh, James ; Jacobson, Ivar ; Booch, Grady: *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. – ISBN 0321245628 3.4.1

[Sch16] Schüller, Andreas: *Fortschrittberichte VDI : Reihe 8, Meß-, Steuerungs- und Regelungstechnik*. Bd. Nr. 1254: *Ein Referenzmodell zur Beschreibung allgemeiner Prozeduren im leittechnischen Umfeld*. Düsseldorf : VDI Verlag GmbH, 2016. – XIV, 148 Seiten : Diagramme S. `http://publications.rwth-aachen.de/record/686692`. – ISBN 978–3–18–525408–6. – Als Manuskript gedruckt. - Weitere Reihe: Lehrstuhl für Prozessleittechnik der RWTH Aachen; Dissertation, RWTH Aachen University, 2016 3.4

[SEE09] Schlütter, Markus ; Epple, Ulrich ; Edelmann, Thomas: *ARGESIM report*. Bd. 35: *On Service-Orientation as a New Approach for Automation Environments*. Vienna : ARGESIM, ARGE Simulation News, Vienna Univ. of Technology, 2009. – 2426–2431 S. `http://publications.rwth-aachen.de/record/115981`. – ISBN 978–3–901608–35–3 3.2.1

[SMS+06] Strasser, T. ; Müller, I. ; Sünder, C. ; Hummer, O. ; Uhrmann, H.: Modeling of Reconfiguration Control Applications based on the IEC 61499 Reference Model for Industrial Process Measurement and Control Systems. In: *Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on*, 2006, S. 127–132 3.2.4

[SOG14] Stattelmann, Stefan ; Oriol, Manuel ; Gamer, Thomas: Execution Time Analysis for Industrial Control Applications. In: Buhnova, Barbora (Hrsg.) ; Happe, Lucia (Hrsg.) ; Kofron, Jan (Hrsg.): *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2014, Grenoble, France, 12th April 2014*. Bd. 147, 2014 (EPTCS), 16–31 6.2.6

[SSF13] Schumacher, Frank ; Schröck, Sebastian ; Fay, Alexander: Tool support for an automatic transformation of GRAFCET specifications into IEC 61131-3 control code. In: *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, 2013. – ISSN 1946–0740, S. 1–4 3.4.1

[SVE07] Stahl, Thomas ; Völter, Markus ; Efftinge, Arno Haase: *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. Heidelberg : Dpunkt-Verl, 2007. – ISBN 3898644480 3.2.3

[SVH13] Schütz, Daniel ; Vogel-Heuser, Birgit: Werkzeugunterstützung für die Entwicklung von SPS-basierten Softwareagenten zur Erhöhung der Verfügbarkeit. In: Göhner, Peter (Hrsg.): *Agentensysteme in der Automatisierungstechnik*. Springer Berlin Heidelberg, 2013 (Xpert.press). – ISBN 978–3–642–31767–5, S. 291–303 2.2.4

[SWH+08] Sünder, Christoph ; Wenger, Monika ; Hanni, Christian ; Gosetti, Ivo ; Steininger, Heinrich ; Fritsche, Josef: Transformation of existing IEC 61131-3 automation projects into control logic according to IEC 61499. In: *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008*, 2008, S. 369–376 6.1.1

[SZ11] Schimmel, Andreas ; Zoitl, Alois: Distributed online change for IEC 61499. In: *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, 2011. – ISSN 1946–0740, S. 1–7 3.2.4

[SZE10] Strasser, Thomas ; Zoitl, Alois ; Ebenhofer, Gerhard: 4DIAC - Ein Open Source Framework für verteilte industrielle Automatisierungs- und Steuerungssysteme. In: *Informatik 2010: Service Science - Neue Perspektiven für die Informatik, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Band 1, 27.09. - 1.10.2010, Leipzig*, 2010, 435–440 3.3.2

[TF11] Thramboulidis, Kleanthis ; Frey, Georg: Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation. In: *Journal of Software Engineering and Applications* 04 (2011), Nr. 04, S. 217–226. http://dx.doi.org/10.4236/jsea.2011.44024. – DOI 10.4236/jsea.2011.44024. – ISBN 978–1–4577–0017–0 3.2.3

[TFB13] Theis, Jens ; Fohler, Gerhard ; Baruah, Sanjoy: Schedule Table Generation for Time-Triggered Mixed Criticality Systems. In: *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, 2013 6.1.6

[Thr13] Thramboulidis, Kleanthis: IEC 61499 vs. 61131: A Comparison Based on Misperceptions. In: *JSEA* 06 (2013), Nr. 08, 405–415. http://dx.doi.org/10.4236/jsea.2013.68050. – DOI 10.4236/jsea.2013.68050 2.2.6

[TM09a] Tauchnitz, Thomas ; Maier, Uwe: Prozessleitsysteme (PLS). In: Früh, Karl F. (Hrsg.) ; Maier, Uwe (Hrsg.) ; Schaudel, Dieter (Hrsg.): *Handbuch der Prozessautomatisierung*. Oldenbourg Industrieverlag, 2009. – ISBN 383563142X 2.2, 2.2.4, 2.2.7, 4.2.2, 4.2.2

[TM09b] Tauchnitz, Thomas ; Maier, Uwe: Speicherprogrammierbare Steuerungen (SPS). In: Früh, Karl F. (Hrsg.) ; Maier, Uwe (Hrsg.) ; Schaudel, Dieter (Hrsg.): *Handbuch der Prozessautomatisierung*. Oldenbourg Industrieverlag, 2009. – ISBN 383563142X 2.2.3, 2.2.3

[TS05] Tawarmalani, Mohit ; Sahinidis, Nikolaos V.: A polyhedral branch-and-cut approach to global optimization. In: *Mathematical Programming* 103 (2005), Nr. 2, S. 225–249 6.3.2

[TVK01] Tommila, Teemu ; Ventä, Olli ; Koskinen, Kari: Next generation industrial automation–needs and opportunities. In: *Automation Technology Review* 2001 (2001), S. 34–41 3.2.1

[Uec05] Uecker, Felix: *Fortschritt-Berichte VDI: Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 1075: *Konzept zur Prozessdatenvalidierung für die Prozessleittechnik*. Düsseldorf : VDI-Verlag, 2005. – ISBN 978–3–18–507508–5 7.4, 7.4, 7.10

[VDI95] *VDI/VDE 3696: Herstellerneutrale Konfigurierung von Prozeßleitsystemen – Blatt 2: Standard-Funktionsbausteine*. Verein Deutscher Ingenieure, Oktober 1995 1.1

[VDI10] *VDI/VDE 2653 Blatt 1 Agentensysteme in der Automatisierungstechnik - Grundlagen*. Verein Deutscher Ingenieure, Juni 2010 3.2.2

[VDI15a] *VDI/VDE 3699: Prozessführung mit Bildschirmen – Blatt 1: Begriffe*. Verein Deutscher Ingenieure, April 2015 4.1

[VDI15b] VDI/VDE-Gesselschaft Mess- und Automatisierungstechnik: *Statusreport: Industrie 4.0 - Technical Assets*. `https://www.vdi.de/artikel/industrie-40-technische-assets-und-abgestimmte-begriffe/`. Version: November 2015, Abruf: 10.03.2016 2.2.8

[VFM03] Velasco, Manel ; Fuertes, Josep M. ; Marti, Pau: The self triggered task model for real-time control systems. In: *24th IEEE Real-Time Systems Symposium (work in progress*, 2003, S. 67–70 5.2.2

[VHDB13] Vogel-Heuser, Birgit ; Diedrich, Christian ; Broy, Manfred: Anforderungen an CPS aus Sicht der Automatisierungstechnik. In: *at – Automatisierungstechnik* 61 (2013), Januar, Nr. 10. `http://dx.doi.org/10.1515/auto.2013.0061`. – DOI 10.1515/auto.2013.0061 4.1, 4.1, 4.1, 4.2.1, 4.2.2

[VHDF+14] Vogel-Heuser, Birgit ; Diedrich, Christian ; Fay, Alexander ; Jeschke, Sabine ; Kowalewski, Stefan ; Wollschlaeger, Martin ; Göhner, Peter: Challenges for Software Engineering in Automation. In: *JSEA* 07 (2014), Nr. 05, S. 440–451. `http://dx.doi.org/10.4236/jsea.2014.75041`. – DOI 10.4236/jsea.2014.75041 4.1, 4.2.1

[VhKW09] Vogel-heuser, Birgit ; Kegel, Gunther ; Wucherer, Klaus: Global information architecture for industrial automation. In: *atp edition-Automatisierungstechnische Praxis* 51 (2009), Nr. 01-02, S. 108–115 2.2.8

[VHLL15] Vogel-Heuser, Birgit ; Lee, Jay ; Leitão, Paulo: Agents enabling cyber-physical production systems. In: *at - Automatisierungstechnik* 63 (2015), Januar, Nr. 10. http://dx.doi.org/10.1515/auto-2014-1153. – DOI 10.1515/auto–2014–1153 3.2.2

[VMSS07] Voigt, K.-I. ; Monsees, H. ; Schorr, S. ; Saatmann, M.: Flexibilität und Adaptivität – Verständnis und Ausprägung. In: Günthner, Willibald A. (Hrsg.): *Neue Wege in der Automobillogistik*. Springer Berlin Heidelberg, 2007 (VDI-Buch). – ISBN 978–3–540–72404–9, S. 39–59 2.1.2

[VPN09] Veiga, G ; Pires, JN ; Nilsson, Klas: Experiments with service-oriented architectures for industrial robotic cells programming. In: *Robotics and Computer-Integrated Manufacturing* 25 (2009), Nr. 4, S. 746–755 3.2.1

[Vya11] Vyatkin, Valeriy: IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. In: *Industrial Informatics, IEEE Transactions on* 7 (2011), Nr. 4, S. 768–781 2.2.6

[Vya13] Vyatkin, Valeriy: Software Engineering in Industrial Automation: State-of-the-Art Review. In: *Industrial Informatics, IEEE Transactions on* 9 (2013), August, Nr. 3, S. 1234–1249. http://dx.doi.org/10.1109/TII.2013.2258165. – DOI 10.1109/TII.2013.2258165. – ISSN 1551–3203 3.2.3

[WE15] Wagner, Constantin ; Epple, Ulrich: Sprechende Kommandos als Grundlage moderner Prozessführungsschnittstellen. In: *[AUTOMATION 2015 : 11.-12.Juni 2015, Baden-Baden / VDI]*. Düsseldorf : VDI-Verl., Juni 2015 3.4.3

[WEE+08] Wilhelm, Reinhard ; Engblom, Jakob ; Ermedahl, Andreas ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem ; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenström, Per: The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools. In: *ACM Trans. Embed. Comput. Syst.* 7 (2008), Mai, Nr. 3, 36:1–36:53. http://dx.doi.org/10.1145/1347375.1347389. – DOI 10.1145/1347375.1347389. – ISSN 1539–9087 6.2.6

[Wer09] Werner, Bernhard: Object-oriented extensions for IEC 61131-3. In: *IEEE Industrial Electronics Magazine* 3 (2009), Dezember, Nr. 4, S. 36–39. http://dx.doi.org/10.1109/MIE.2009.934795. – DOI 10.1109/MIE.2009.934795. – ISSN 1932–4529 3.3.1

[WGKO15] Wahler, Michael ; Gamer, Thomas ; Kumar, Atul ; Oriol, Manuel: FASA: A software architecture and runtime framework for flexible distributed automation systems. In: *Journal of Systems Architecture* 61 (2015), Nr. 2, 82 - 111. http://dx.doi.org/http://dx.doi.org/10.1016/j.sysarc.

2015.01.002. – DOI http://dx.doi.org/10.1016/j.sysarc.2015.01.002. – ISSN 1383–7621 3.3.3

[Wit12] Witsch, Daniel: *Modellgetriebene Entwicklung von Steuerungssoftware auf Basis der UML unter Berücksichtigung der domänenspezifischen Anforderungen des Maschinen- und Anlagenbaus*. Göttingen : Sierke, 2012. – ISBN 978–3–86844–545–9 3.4.2, 6.2.3

[WJ95] Wooldridge, Michael ; Jennings, Nicholas R.: Intelligent agents: Theory and practice. In: *The knowledge engineering review* 10 (1995), Nr. 02, S. 115–152 3.2.2

[WKS+16] Wagner, Constantin ; Kampert, David ; Schüller, Andreas ; Palm, Florian ; Grüner, Sten ; Epple, Ulrich: Model based synthesis of automation functionality. In: *at - Automatisierungstechnik* 64 (2016), Nr. 3, 168-185. http://dx.doi.org/10.1515/auto-2015-0094. – ISSN 0178–2312 3.3.4, 3.4.3

[WO14] Wahler, Michael ; Oriol, Manuel: Disruption-free software updates in automation systems. In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, 2014, S. 1–8 3.2.4, 3.3.3, 6.3.3

[Woo09] Wooldridge, Michael: *An introduction to multiagent systems*. John Wiley & Sons, 2009 3.2.2

[WRKO11] Wahler, Michael ; Richter, Stefan ; Kumar, Sumit ; Oriol, Manuel: Non-disruptive large-scale component updates for real-time controllers. In: *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, 2011, S. 174–178 3.2.4, 3.3.3

[WRKVH10] Witsch, D. ; Ricken, M. ; Kormann, B. ; Vogel-Heuser, B.: PLC-statecharts: An approach to integrate umlstatecharts in open-loop control engineering. In: *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, 2010, S. 915–920 3.4.1, 3.4.2, 3.4.3, 6.2.4

[WVH11] Witsch, Daniel ; Vogel-Heuser, Birgit: PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking. In: Bittanti, Sergio (Hrsg.): *Proceedings of the 18th IFAC World Congress*, Elsevier BV, August 2011 3.2.3, 6.2.3, 6.2.3, 6.2.4

[WZSS09] Wenger, Monika ; Zoitl, Alois ; Sünder, Christoph ; Steininger, Heinrich: Semantic correct transformation of IEC 61131-3 models into the IEC 61499 standard. In: *IEEE Conference on Emerging Technologies Factory Automation, 2009. ETFA 2009*, 2009. – ISSN 1946–0759, S. 1–7 6.1.1

[YGE13] Yu, Liyong ; Grüner, Sten ; Epple, Ulrich: An Engineerable Procedure Description Method for Industrial Automation. In: *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA 2013) : Cagliari, Italy, 10 - 13 September 2013*. Piscataway, NJ : IEEE, 2013. – ISBN 978–1–4799–0862–2, S. 8 S. 3.1, 3.4.1, 3.4.3, 6.2.3, 6.2.3, 6.2.4

[YQGE12] Yu, Liyong ; Quirós, Gustavo ; Grüner, Sten ; Epple, Ulrich: SFC-based Process Description for Complex Automation Functionalities. In: *EKA 2012 : Entwurf komplexer Automatisierungssysteme; Beschreibungsmittel, Methoden,Werkzeuge und Anwendungen ; 12. Fachtagung mit Tutorium ; 08. Mai bis 10. Mai 2012 Fachtagung, Magdeburg, Denkfabrik im Wissenschaftshafen / Institut für Automation und Kommunikation e.V. Magdeburg, Ulrich Jumar*. Magdeburg : ifak, Institut für Automation und Kommunikation e.V., 2012. – ISBN 978–3–940961–72–3, 13-20 3.1, 6.2.3

[YSE14] Yu, Liyong ; Schüller, Andreas ; Epple, Ulrich: On the engineering design for systematic integration of agent-orientation in industrial automation. In: *ISA Transactions* 53 (2014), September, Nr. 5, 1404–1409. http://dx.doi.org/10.1016/j.isatra.2013.12.029. – DOI 10.1016/j.isatra.2013.12.029 3.2.2

[Yu16] Yu, Liyong: *Fortschritt-Berichte VDI: Reihe 8, Mess-, Steuerungs- und Regelungstechnik*. Bd. 1248: *A Reference Model for the Integration of Agent Orientation in the Operative Environment of Automation Systems*. Düsseldorf : VDI-Verlag, 2016. – ISBN 978–3–18–524808–5 3.2.2, 3.4, 3.4.1, 3.4.3, 3.4.3, 6.1.1, 6.2.3, 6.2.3, 6.2.3

[YV13] Yan, Jeffrey ; Vyatkin, Valeriy: Extension of reconfigurability provisions in IEC 61499. In: *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, 2013. – ISSN 1946–0740, S. 1–7 3.2.4

[ZGSS07] Zoitl, Alois ; Grabmair, Gunnar ; Smodic, Rene ; Strasser, Thomas: An Execution Environment for Real-Time Constrained Control Software based on IEC 61499. In: *Industrial Informatics, 2007 5th IEEE International Conference on* Bd. 2, 2007. – ISSN 1935–4576, S. 1071–1076 3.3.2

[Zil96] Zilberstein, Shlomo: Using anytime algorithms in intelligent systems. In: *AI magazine* 17 (1996), Nr. 3, S. 73 5.1.2

[Zoi08] Zoitl, Alois: *Real-Time Execution for IEC 61499*. International Society of Automation, 2008. – ISBN 978–1934394274 6.1.6

[ZSSB09] Zoitl, Alois ; Strasser, Thomas ; Sünder, Christoph ; Baier, Thomas: Is IEC 61499 in harmony with IEC 61131-3? In: *Industrial Electronics Magazine, IEEE* 3 (2009), Dezember, Nr. 4, S. 49–55. http://dx.doi.org/10.1109/MIE.2009.934797. – DOI 10.1109/MIE.2009.934797. – ISSN 1932–4529 2.2.6, 6.1.7

Diese Dissertation enthält 51 Abbildungen, 2 Tabellen und 177 Literaturquellen.

Online-Buchshop für Ingenieure

BUCHSHOP

Online-Shops

Fachliteratur und mehr - jetzt bequem online recherchieren & bestellen unter: www.vdi-nachrichten.com/Der-Shop-im-Ueberblick

Täglich aktualisiert: Neuerscheinungen VDI-Schriftenreihen

Im Buchshop von vdi-nachrichten.com finden Ingenieure und Techniker ein speziell auf sie zugeschnittenes, umfassendes Literaturangebot.

Mit der komfortablen Schnellsuche werden Sie in den VDI-Schriftenreihen und im Verzeichnis lieferbarer Bücher unter 1.000.000 Titeln garantiert fündig.

Im Buchshop stehen für Sie bereit:

VDI-Berichte und die Reihe **Kunststofftechnik:**

Berichte nationaler und internationaler technischer Fachtagungen der VDI-Fachgliederungen

Fortschritt-Berichte VDI:

Dissertationen, Habilitationen und Forschungsberichte aus sämtlichen ingenieurwissenschaftlichen Fachrichtungen

Newsletter „Neuerscheinungen":

Kostenfreie Infos zu aktuellen Titeln der VDI-Schriftenreihen bequem per E-Mail

Autoren-Service:

Umfassende Betreuung bei der Veröffentlichung Ihrer Arbeit in der Reihe Fortschritt-Berichte VDI

Buch- und Medien-Service:

Beschaffung aller am Markt verfügbaren Zeitschriften, Zeitungen, Fortsetzungsreihen, Handbücher, Technische Regelwerke, elektronische Medien und vieles mehr – einzeln oder im Abo und mit weltweitem Lieferservice

Die Reihen der Fortschritt-Berichte VDI:

1 Konstruktionstechnik/Maschinenelemente
2 Fertigungstechnik
3 Verfahrenstechnik
4 Bauingenieurwesen
5 Grund- und Werkstoffe/Kunststoffe
6 Energietechnik
7 Strömungstechnik
8 Mess-, Steuerungs- und Regelungstechnik
9 Elektronik/Mikro- und Nanotechnik
10 Informatik/Kommunikation
11 Schwingungstechnik
12 Verkehrstechnik/Fahrzeugtechnik
13 Fördertechnik/Logistik
14 Landtechnik/Lebensmitteltechnik
15 Umwelttechnik
16 Technik und Wirtschaft
17 Biotechnik/Medizintechnik
18 Mechanik/Bruchmechanik
19 Wärmetechnik/Kältetechnik
20 Rechnerunterstützte Verfahren (CAD, CAM, CAE CAQ, CIM ...)
21 Elektrotechnik
22 Mensch-Maschine-Systeme
23 Technische Gebäudeausrüstung

ISBN 978-3-18-525708-7