LEHRSTUHL FÜR REALZEIT-COMPUTERSYSTEME
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. G. FÄRBER

Modellierung und effiziente Implementierung eingebetteter Realzeitsysteme

Martin Orehek

Dissertation

Lehrstuhl für Realzeit-Computersysteme

Modellierung und effiziente Implementierung eingebetteter Realzeitsysteme

Martin Orehek

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor Ingenieurs (Dr.-Ing.)

genehmigte Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing./Univ. Tokio, M. Buss
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr.-Ing. Georg Färber
	2. Univ.-Prof. Dr. rer. nat., Dr. rer. nat. habil. Manfred Broy

Die Dissertation wurde am 24.06.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 23.10.2003 angenommen.

Vorwort

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München im Teilprojekt *„Architektur und Entwurf hybrider Realzeitsysteme"* des Forschungsverbundes Software-Engineering FORSOFT II der Bayerischen Forschungsstiftung.

Ein besonderer Dank gilt meinem Doktorvater, Herrn Prof. Dr.-Ing. G. Färber, für das in mich gesetzte Vertrauen und die mir gewährten Freiheiten. Die mit ihm geführten fachlichen Diskussionen trugen wesentlich zum Gelingen dieser Arbeit bei. Bedanken möchte ich mich auch bei Prof. Dr. M. Broy für das Interesse an meiner Arbeit und die Übernahme des Zweitberichts.

Ganz herzlich bedanken möchte mich bei meinem Kollegen Benito Liccardi, der mir durch seine Fachkenntnisse ein geschätzter Diskussionspartner war. Im weiteren gilt mein Dank Philipp Harms, Alexander Münnich und Christian Robl für die fachliche Unterstützung und die Korrektur meiner Arbeit. Ich möchte mich bei allen Mitarbeitern des Lehrstuhls für Realzeit-Computersysteme für die gute Arbeitsatmosphäre und die zahlreichen interessanten Diskussionen bedanken.

Nicht zuletzt schulde ich großen Dank meiner Familie und meiner Lebensgefährtin Andrea, die mir immer Beistand geleistet und mich in allen Phasen uneingeschränkt unterstützt haben.

München, im Juni 2003

Zusammenfassung

Eingebettete Realzeitsysteme gewinnen für die Entwicklung neuer Produkte und Gesamtsysteme immer mehr an Bedeutung. Dabei wird ein Großteil der neuen Funktionalität durch die darin enthaltene Software realisiert. Dies ist zum einen auf den Wunsch nach mehr Flexibilität zurückzuführen und zum anderen notwendig, um die steigende Komplexität der gewünschten Funktionen und Dienste zu beherrschen.

Bei der Entwicklung von Software, speziell bei eingebetteten Systemen, zeichnet sich ein klarer Trend in Richtung modell-basierter Ansätze ab. Durch den dabei erreichten höheren Abstraktionsgrad wird die Entwicklung komplexer und umfangreicher Systeme erleichtert. Dank der Verwendung ausführbarer Spezifikations-Sprachen kann eine frühe Evaluierung des Designs sowohl die Entwicklungszeit verkürzen als auch die Qualität des Endergebnisses gegenüber konventionellen Entwurfsmethodiken verbessern.

Die Entwicklung geht sogar so weit, dass aus dem modellierten System neben einer Simulation auch automatisch eine realzeitfähige Implementierung für eingebettete Controller generiert wird. Eine geeignete Entwurfs-Methodik und entsprechende Werkzeugunterstützung ist bei dieser Art der Software-Entwicklung der Schlüssel zum Erfolg.

In dieser Arbeit wird ein Verfahren für eine weitgehend automatisierte Umsetzung hybrider graphischer Spezifikationen auf eine echtzeitfähige Implementierung vorgestellt. Dazu wurde eine neue Software-Architektur erarbeitet, die neben der Berücksichtigung von zustandsbasierten und signalflussorientierten Laufzeitmodellen auch deren Abbildung sowohl auf zyklische als auch auf ereignisgesteuerte SW-Teile unterstützt. Damit kann eine effiziente, ressourcenschonende Umsetzung auf Zielsysteme mit beschränkter Rechenleistung durchgeführt werden. Zur Integration der vorgestellten Methoden in einem modell-basierten Entwurfsprozess wurde ein Vorgehensmodell für die Phasen *Implementierung* und *Realzeit-Analyse* entwickelt.

Grundelement der neuen SW-Architektur ist die Zusammenfassung einer Nachrichten-Warteschlange mit einem Server-Thread. Berechnungswünsche im System werden durch Ereignisse, die entweder von außen kommen oder intern bei der Verarbeitung entstehen, an die verschiedenen Server-Threads übergeben. Das Besondere ist, dass die statischen Prioritäten der Ereignisse nicht nur zum Sortieren der Warteschlangen verwendet, sondern auch während der Verarbeitung dem jeweiligen Server-Thread vererbt werden. Dadurch wird der Tatsache Rechnung getragen, dass Ereignisse im System Berechnungswünsche wiederspiegeln und sie somit auch die Reihenfolge der Bearbeitungen mit ihren jeweiligen Prioritäten vorgeben.

Damit trotz des *run-to-completion* Paradigmas der Server-Threads bei der Bearbeitung von Ereignissen ein berechenbares Zeitverhalten der SW-Architektur erreicht wird, wurden unterschiedliche Vererbungsstrategien (*Basic-Priority-Inheritence* und *Preemption-Threshold*) zwischen den Prioritäten der Nachrichten in den Queues und den jeweiligen Server-Thread Prioritäten entwickelt und im unterlagerten Laufzeitsystem umgesetzt.

Zur Berechnung des zeitlichen worst-case Verhaltens wurden für die entwickelte SW-Architektur und die untersuchten Vererbungsstrategien alle notwendigen mathematischen Formeln aufgestellt. Diese ermöglichen eine weitgehend automatisierte Realzeit-Analyse für eine konkrete Umsetzung eines graphischen Modells.

Zur Beschreibung der Kommunikationsstrukturen zwischen Aktionen in einem zu analysierenden System wurde der *Aktions-Präzedenz-Graph (APG)* eingeführt. Um dabei eine möglichst detaillierte Beschreibung der jeweils vorliegenden Zusammenhänge zu ermöglichen, wurden zwei unterschiedliche Abstraktions-Ebenen von *APG* definiert. Der *allgemeine APG* ermöglicht

neben der Darstellung paralleler Pfade auch die Definition exklusiver, sich ausschließender Kommunikationsbeziehungen. Bei der worst-case Reaktionszeitberechnung für eine konkrete Aktion wird dieser *allgemeine APG* dann mittels definierter Transformationsvorschriften in den jeweiligen *konkreten APG* überführt. Der *konkrete APG* erlaubt die automatisierte Berechnung der worst-case Reaktionszeit der betrachteten Aktion auf ein definiertes Ereignis.

Die Reaktionszeit-Berechnung baut auf die worst-case Ausführungszeiten aller im System vorhandenen Aktionen auf. Im Rahmen der vorgestellten Arbeit wurde zur einfachen, automatisierten Gewinnung von Ausführungszeiten eine Co-Simulation umgesetzt, die konkrete Ausführungszeitmessungen bei der realen Laufzeitumgebung unter realistischen Bedingungen durchführt. Die gemessenen Werte spiegeln somit realistische Ausführungszeiten wieder, berücksichtigen aber keine worst-case Pfade der Software. Ein weiterer Vorteil der Co-Simulation ist der Funktionstest der realzeitfähigen Umsetzung des graphisch spezifizierten Designs auf dem Zielsystem, ohne dass der einbettende Prozess benötigt wird. Dadurch kann die Korrektheit der automatisierten Abbildung in der Implementierungs-Phase auf parallele Threads und priorisierte Aktionen im Realzeit-Betriebssystem verifiziert werden.

Eine weitere zur Durchführung der vorgestellten Realzeitanalyse notwendige Information betrifft die Rechenzeitanforderung aus dem einbettenden System. Hierzu wurde die von Gresser [117] eingeführten Ereignis-Tupel verwendet, die eine Beschreibung beliebiger Ereignisfolgen ermöglichen. Letztere ergeben sich entweder aus der Spezifikation des Systems oder aus einer entsprechenden Analyse des einbettenden Prozesses.

Die vorgestellten Konzepte wurden zur Validierung in eine *state-of-the-art* Werkzeugkette (Simulink/Stateflow) integriert und anhand konkreter Anwendungsbeispiele evaluiert. Dabei konnten die Anwendbarkeit und die Genauigkeit der Methoden bewiesen werden.

Inhaltsverzeichnis

Vorwort ..... **i**

Zusammenfassung ..... **iii**

Inhaltsverzeichnis ..... **v**

1 Einleitung **1**

1.1 Bedeutung und Auswirkungen eingebetteter Systeme 1

1.2 Modell-basierte Software-Entwicklung 2

1.3 Ziel und Aufbau dieser Arbeit 5

2 Grundlagen und Stand der Technik **7**

2.1 Modellierung eingebetteter Realzeitsysteme 7

2.2 Implementierungsvarianten von Software für Realzeitsysteme 9

2.3 Werkzeugunterstützung: Matlab, Simulink/Stateflow 10

2.3.1 Kontinuierliche Systeme mit Simulink 12

2.3.2 Ereignisgesteuerte Systeme mit Stateflow 13

2.3.3 Umsetzung graphischer Spezifikationen 16

2.3.4 Probleme bei der derzeitigen Implementierung 20

2.4 Andere CASE-Werkzeuge für eingebettete Systeme 23

3 SW-Architektur hybrider graphischer Spezifikationen **25**

3.1 Anforderungen 25

3.1.1 Kopplung unterschiedlicher Laufzeitmodelle 25

3.1.2 Möglichkeit eines Realzeitnachweises 26

3.1.3 Übereinstimmung Simulation und Implementierung 26

3.2 Konzept der SW-Architektur hybrider Systeme 28

3.2.1 Statische Sicht der Software-Architektur 29

3.2.2 Dynamisches Verhalten 31

3.3 Laufzeitsystem 37

4 Realzeitnachweis **39**

4.1 Taskmodell 39

4.1.1 Messagequeue und Thread 39

4.1.2 Messages und Aktionen 39

4.1.3 Aktions-Präzedenz-Graph (APG) 40

4.1.4 Systembelastung durch das einbettende System 54
4.2 Formeln für den Realzeitnachweis 55
4.2.1 Blockierzeiten 57
4.2.2 Startzeitpunkte 58
4.2.3 Endzeitpunkte 59
4.2.4 Reaktionszeiten 60

5 Umsetzung bei Simulink/Stateflow 61
5.1 Software-Architektur 61
5.1.1 Graphische Modellierung 61
5.1.2 RT-Implementierung (Codegenerierung) 66
5.1.3 Simulation 68
5.1.4 Divergenz zwischen Simulation und Implementierung 69
5.1.5 Richtlinien für den Entwickler 69
5.1.6 Simulation eines RT-Kernels (*TrueTime*-Simulator) 70
5.2 RT-Nachweis 71
5.2.1 Parametergewinnung 72
5.2.2 Implementierung des RT-Nachweises 75
5.2.3 Konkretes Anwendungs-Beispiel 77

6 Realisierung der Laufzeitsystemunterstützung 83
6.1 Anforderungen und Wahl eines RTOS 83
6.2 Notwendige Anpassungen und Implementierung 84
6.2.1 Basic Priority Inheritence (BPI) Protokoll 84
6.2.2 Preemption Threshold (PT) Protokoll 86

7 Anwendungsbeispiele 89
7.1 Hochregallager (HRL) 89
7.1.1 Systembeschreibung 89
7.1.2 Modell-basierter Entwurf 90
7.1.3 Implementierung und Realzeit-Analyse 94
7.1.4 Ergebnisse und Erfahrungen 104
7.2 Hydrostatische Vorderachs-Antriebs-Steuerung 104
7.2.1 Systembeschreibung 105
7.2.2 Modell-basierter Entwurf 105
7.2.3 Implementierung und Realzeit-Analyse 108
7.2.4 Ergebnisse und Erfahrungen 114

Ausblick 115

Literatur 117

A Anhang: Erweiterung der Block-Bibliothek mit m- und tlc-Files 125

A.1 Erweiterungen zur Modellierung der SW-Architektur 125

A.2 rtems_task_rcv_msg_build.m 126

A.3 rtems_send_msg.tlc 127

A.4 Block-Bibliothek zur Realisierung der Co-Simulation 129

B Anhang: Automatisiertes Parsen des graphischen Modells 131

B.1 parse_for_APG.m 131

B.2 build_APG.m 141

C Anhang: Realzeit-Analyse Skripten 145

C.1 responsetime_BPI.m 145

C.2 responsetime_PT.m 151

D Anhang: Realzeitbetriebsystem RTEMS-4.5.0 155

D.1 Erweiterungen des RTOS für Prioritätsvererbungs-Strategien 155

Kernel Funktionen 155

POSIX-API Funktionen 167

D.2 RTOS-Overhead (RTEMS-4.5.0 auf MPC555) 169

Interrupt-Verarbeitungs-Overhead 169

Task-Switch-Overhead 170

1 Einleitung

Eingebettete Realzeitsysteme sind Computersysteme, die eine externe Umwelt überwachen, sie kontrollieren oder mit ihr interagieren. Sie sind meist in einem umfangreichen System eingebettet und bestehen aus einem Software-Teil und einer darunter liegenden Hardware (Mikrocontroller), die über Ein-Ausgabe-Schnittstellen (Sensoren/Aktoren) mit den physikalischen oder biologischen Objekten (Umwelt) verbunden ist.

1.1 Bedeutung und Auswirkungen eingebetteter Systeme

Die Bedeutung eingebetteter Systeme ist unumstritten[1]. Sie sind allgegenwärtig und ihr Marktvolumen (Stückzahlen) ist ca. 100 mal so groß wie der Desktop-Markt [1]. Heute erreicht kaum ein Produkt den Markt, ohne ein eingebettetes System zu beinhalten und mehr als 99% aller weltweit produzierten Prozessoren werden in eingebetteten Systemen eingesetzt [2].

Die Anzahl eingebetteter Systeme in einem Produkt reicht von eins bis zehn in Verbraucher-Produkten und bis zu hundert in großen professionellen Systemen. Ein durchschnittlicher Haushalt verwendet heute ungefähr 50 eingebettete Systeme, wobei ihre Anzahl in dieser Dekade um mindestens eine Größenordnung ansteigen wird [1].

Der stark zunehmende Einsatz von eingebetteten Systemen in neuen Produkten und Dienstleistungen eröffnet enorme Möglichkeiten für jede Art von Unternehmen. Gleichzeitig stellt jedoch das schnelle Tempo der Verbreitung auch eine immense Bedrohung für die meisten Unternehmen dar.

Diese Entwicklung betrifft Unternehmen und Einrichtungen aus unterschiedlichen Sparten wie Landwirtschaft, Gesundheitswesen, Umweltschutz, Sicherheit, Mechanik, Kommunikations- , Automobil- und Medizin-Technik, Unterhaltungselektronik, usw. Sie alle müssen, um wettbewerbsfähig zu bleiben, rechtzeitig auf folgende technologische und markttechnische Herausforderungen reagieren:

- Wachsende Bedeutung von Flexibilität und Software
- Breite Vielfältigkeit und zunehmende Komplexität von Anwendungen
- Zunehmende interdisziplinäre Natur von Produkten und Service
- Schrumpfende *„time-to-market"*
- Zunehmende Anzahl nicht-funktionaler Randbedingungen
- Zunehmendes Maß an Integration und Vernetzung

Die aktuelle Situation ist vor allem für kleine und mittlere Unternehmen bedrohlich. Entsprechend den Prognosen aus der *Embedded Systems Roadmap 2002* [1] werden mehr als die Hälfte von ihnen in der nächsten Dekade verschwinden, sofern sie nicht Mittel und Wege finden, um das notwendige *Know-How* für die eingebetteten Systeme in ihren Produkten und Dienstleistungen aufzunehmen und umzusetzen.

Im Einklang mit dem Gesetz von Moore[2] wächst die Komplexität eingebetteter Systeme aufgrund der zunehmenden Anzahl mikroelektronischer Komponenten exponentiell an. Um gleichzeitig die zunehmende Forderung nach Flexibilität solcher Systeme zu erfüllen, kann mit

---

[1] Report Task force ICT-en-kennis (2001) [4]: The market for embedded systems will grow exponentially also in the next decade.... Every company faces the challenge of changing over to embedded and distributed systems to not get off track.

[2] Gordon Moore beobachtete 1965 eine exponentielle Entwicklung bei der Anzahl von Transistoren in integrierten Schaltkreisen, die sich bis heute bewahrheitet.

einem signifikanten Zuwachs an eingebetteter Software gerechnet werden. Dies führt dazu, dass in Zukunft die Kosten der Software einen dominanten Anteil der Entwicklungs-Kosten eingebetteter Systeme darstellen werden.

Wie in der Hardware-Entwicklung ist auch in der Software-Entwicklung zur Bewältigung der exponentiell wachsenden Komplexität die Einführung von neuen Abstraktions-Ebenen unerlässlich.

So kann seit der Veröffentlichung der *Unified Modelling Language* (*UML*) durch die *Object Management Group* (*OMG*) im Jahre 1997 ein klarer Trend bei der Entwicklung von Software-Systemen in Richtung modell-basierter Ansätze erkannt werden.

Mit dem *Unified Process* wurde 1999 eine allgemeine Software-Entwicklungsmethodik veröffentlicht, die durch Verwendung der *UML* eine allgemeine Art der modell-basierten Entwicklung darstellt. Eine konkrete Ausprägung davon ist unter anderem der *Rational Unified Process (RUP)*.

Die neueste verabschiedete Spezifikation der *OMG*, die *Model Driven Architecture* (*MDA*) im Sommer 2001 zeigt, dass der modell-basierte Ansatz weiter durch die Standardisierung vorangetrieben wird.

1.2 Modell-basierte Software-Entwicklung

Eine modell-basierte Software-Entwicklung zeichnet sich durch folgende Eigenschaften aus:

- Die graphische Spezifikations-Sprache rückt bei der Entwicklung in den Mittelpunkt. Sie ermöglicht einen höheren Abstraktionsgrad der funktionalen Beschreibung.
- Durch die Verwendung von ausführbaren Spezifikations-Sprachen ist bereits in den frühen Phasen des Designs eine Validierung durch Simulation der sogenannten „ausführbaren Spezifikation" möglich.
- Für eine derartige Simulation ist ein Minimum an Modellierung des einbettenden Prozesses erforderlich. Dies wiederum garantiert, dass das graphische Modell nicht nur gegenüber der Anforderungs-Spezifikation getestet wird, sondern das zu realisierende System im einbettenden Prozess simuliert werden kann. So können Fehler in der Spezifikation erkannt werden, bevor sie, zwar richtig umgesetzt, aber letztendlich unerwünscht, im Endsystem zu finden sind.
- Die iterative Verfeinerung bestimmter Modell-Teile ermöglicht eine evolutionäre Entwicklung des Systems und unterstützt eine Aufteilung komplexer Funktionen. Dadurch können nachträgliche Anforderungs-Änderungen leichter in den Design-Prozess einfließen.
- Das Konzept einer ausführbaren Spezifikation ermöglicht nicht nur die Simulation verschiedener Designvarianten. Durch die abstrakte, zielsystem-unabhängige Funktions-Beschreibung wird es möglich, das Design weitgehend automatisiert auf dedizierte Hardware-Systeme zu implementieren und zu testen (z.B. unterschiedliche Arten des Prototyping).

Durch diese Hauptmerkmale verspricht eine modell-basierte Entwicklung trotz steigender Komplexität der umzusetzenden Funktion höhere Produktivität und Qualität. Das Vorhandensein einer geeigneten Werkzeugunterstützung ist dabei ein essenzieller Bestandteil eines modell-basierten Entwurfsprozesses und des Konzepts einer ausführbaren Spezifikation.

Heute existieren bereits eine Vielzahl kommerzieller und im Rahmen von Forschungsprojekte entwickelte *Computer Aided Software Engineering* Werkzeuge (CASE-Tools). Eine Auswahl ist in Tabelle 1.1 wiedergegeben.

Tabelle 1.1: Auswahl von CASE-Tools zur Entwicklung eingebetteter Realzeit-Software

	Werkzeug	**Hersteller**
kommerziell	Matlab, Simulink/Stateflow	The MathWorks Inc.
	ASCET/SD	Etas GmbH
	Tau UML/SDL Suite	Telelogic AB
	Rational Rose-RT	Rational Software (IBM)
	Real-Time Studio	ARTiSAN Software Tools Inc.

	Werkzeug	**Institut / Universität**
universitär	Ptolomy II	Department of EECS, UC Berkeley
	Giotto	Department of EECS, UC Berkeley
	AutoFocus	Technische Universität München, Institut für Informatik – Lehrstuhl IV

Im Projekt *Architektur und Entwurf hybrider Realzeit-Systeme* (HRS) des Forschungsverbundes FORSOFT II[3] wurde ein speziell für eingebettete Realzeit-Systeme angepasster modell-basierter Entwurfsprozess erarbeitet [74][75][76].

Er berücksichtigt die verschiedenen Anforderungen beim Design des eingebetteten *Control System* von der ersten Phase der Entwicklung bis hin zur Realzeit-Analyse und letztendlichen Implementierung. *Control Systems* sind im Kontext dieser Arbeit sowohl kontinuierliche Regelungen als auch diskrete, ereignisorientierte Steuerungen bzw. eine Verkopplung beider Domänen (hybride Systeme).

Um auf existierende Werkzeuge aufzubauen und diese geeignet zu erweitern, wurde im Projekt HRS zur Modellierung und Simulation das Werkzeug Simulink/Stateflow von The MathWorks Inc. eingesetzt, das zur Zeit in der Industrie sehr weit verbreitet ist.

Die Grundstruktur des Vorgehensmodells ist in Abbildung 1.1 dargestellt und soll nachfolgend kurz beschrieben werden.

Die drei Phasen der Entwicklung sind:

- Umwelt-Spezifikation (*environment specification*)
- System-Design (*system design*)
- Automatisierte Implementierung (*automatic code generation*)

Das zentrale Element im fortlaufenden Entwurf ist eine graphische, ausführbare Spezifikation der zu entwickelnden Funktionalität. Dabei werden in den verschiedenen Phasen jeweils unterschiedliche Aspekte untersucht und umgesetzt.

In der ersten Phase, der Umwelt-Spezifikation, wird das einbettende System modelliert. Dabei werden erste Untersuchungen und Abschätzungen über das Verhalten der Umwelt gewonnen, aus denen sich funktionale und nicht-funktionale Anforderungen (z.B. Zeitbedingungen) an die zu realisierende Steuerung/Regelung ableiten lassen. Das in dieser Phase gewonnene Umgebungs-Modell erlaubt die Simulation der relevanten Zusammenhänge des einbettenden Prozesses und entspricht einem Strecken-Modell in der Regelungstechnik.

[3] Der Bayrische Forschungsverbund Software-Engineering (FORSOFT II) wurde von der Bayrischen Forschungsstiftung und Industriepartnern gefördert. www.forsoft.de

In der zweiten Phase wird das gewonnene Umgebungs-Modell vor allem zur Validierung der zu entwickelnden Funktionalität mittels Simulation eingesetzt. Dabei kann bereits sehr früh mit einem relativ ungenauen Modell des einbettenden Prozesses begonnen werden. Somit ist eine zeitlich entkoppelte Verfeinerung des Streckenmodells (Iterationen in Phase 1) und eine erste Entwicklung der gewünschten Funktionalität (Phase 2) möglich. Sukzessive können die entwickelten graphischen Spezifikationen des *Control Systems* anhand der neuen, verfeinerten Strecken-Modelle getestet werden. Oftmals werden so zuvor nur sehr abstrakt betrachtete Zusammenhänge durch mehrmalige Verfeinerungen (*Refinements*) immer komplexer bis ein adäquater Grad der Modellierung beider Seiten, sowohl des einbettenden Prozesses als auch des eingebetteten Systems, erreicht wird.

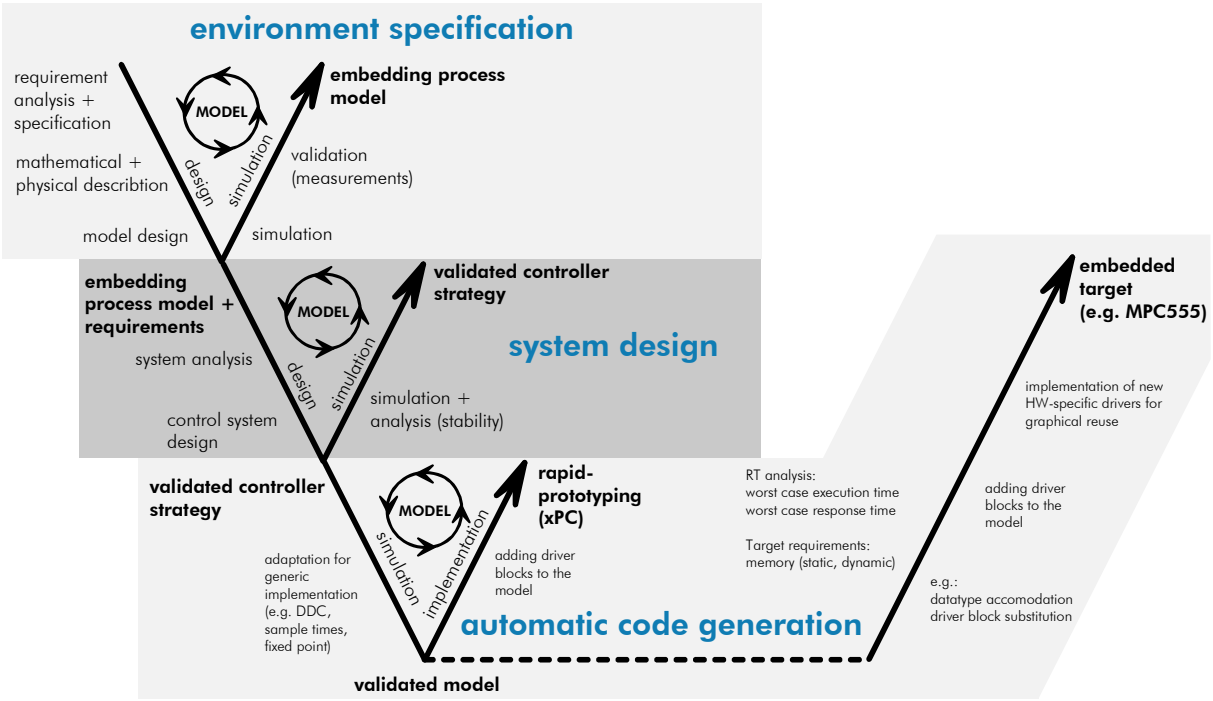

Abbildung 1.1: Modell-basierter Entwurfsprozess aus dem Teilprojekt HRS des Forschungsverbundes FORSOFT II

Entsprechen die Simulations-Ergebnisse für alle spezifizierten Szenarien den definierten Anforderungen, kann mit der ersten realen Umsetzung des graphischen Modells begonnen werden. Hierzu wird in der dritten Phase des vorgestellten Entwurfsprozesses zuerst die Abbildung der graphischen Spezifikation des validierten *Control Systems* auf ein Rapid-Prototyping System vorgenommen (Phase 3 links). Ziel dabei ist, die Funktion der entwickelten Algorithmen an einem realen System zu testen. Hierzu ist zum ersten Mal die reale Umgebung erforderlich, die z.B. den einbettenden Prozess in einem Laboraufbau nachbildet.

Durch Versuchsläufe in diesem Testaufbau kann nicht nur das simulierte Verhalten des zu entwickelnden SW-Teils validiert, sondern auch das Verhalten des einbettenden Prozesses mit den möglicherweise ebenfalls neuen physikalischen Komponenten erprobt werden. Die offline Auswertung der gewonnenen Daten gibt zudem wichtige Hinweise zur Überprüfung und Optimierung des physikalischen Modells aus der ersten Phase. Durch die Wahl eines Rapid-Prototyping Systems für die realzeitfähige Umsetzung der graphischen Spezifikation ist es möglich, sich in dieser ersten Implementierungs-Phase ausschließlich auf die funktionale Verifikation des Designs zusammen mit dem realen einbettenden Prozess zu konzentrieren. Die Anforderungen an eine geeignete Software-Architektur sind, durch die stark überdimensionierte Rechenleistung des Rapid-Prototyping Computersystems, in dieser ersten Implementierungs-Phase noch sehr gering.

Wird jedoch nach einer erfolgreichen Erprobung der graphischen Spezifikation zur letztendlichen Implementierung auf ein eingebettetes System übergegangen (Phase 3 rechts), gewinnt die Software-Architektur an Bedeutung. Durch den höheren Kostendruck und die gewünschte stärkere Einbettung der Controller-Hardware in das Produkt steht im letztendlichen Zielsystem eine geringere Rechenleistung zur Verfügung.

Dadurch steigt bei derselben zu realisierenden Funktionalität die Auslastung der Zielhardware. Ineffiziente Umsetzungen führen bald zu unüberwindbaren Problemen in der späten Implementierungsphase oder zu sporadischen Fehlfunktionen der Software, die nur sehr schwer durch langwierige Tests reproduziert, analysiert und beseitigt werden können.

Für die weitgehend automatisierte Implementierung graphischer Modelle auf eingebetteten Mikrocontrollern mit beschränkter Rechenleistung bei einer möglichst hohen erreichbaren Auslastung ist die Bereitstellung einer analysierbaren Software-Architektur unabdingbar.

Die Software-Architektur muss neben der semantiktreuen Abbildung der verwendeten Modellierungs-Sprachen auf eine echtzeitfähige Laufzeitumgebung auch die Möglichkeit der Realzeit-Analyse bieten. In Zukunft wird nämlich zur Senkung der Entwicklungskosten eine Verkürzung der heute sehr zeitintensiven Testphase des Endsystems notwendig sein. Ein vielversprechender Ansatz dabei ist der verstärkte Einsatz von Analyse-Techniken kombiniert mit gezielten Simulationen der modellierten Funktionalität [67][81][82][84].

1.3 Ziel und Aufbau dieser Arbeit

Ziel dieser Arbeit ist es somit, eine neue Software-Architektur zu entwickeln, die für eine automatisierte Abbildung graphischer Modelle auf ein eingebettetes Zielsystem optimiert ist. Die betrachteten Modelle sind im Kontext dieser Dissertation Regelungen und Steuerungen, die sowohl kontinuierliche Regelkreise als auch diskrete, ereignisorientierte Steuerungen beinhalten. Auch die Abbildung von Verkopplungen beider Domänen, sogenannte hybride Systeme [8] soll möglich sein.

Die SW-Architektur soll den Realzeitbedingungen Rechnung tragen und sich durch eine optimierte Auslastung der Ziel-Hardware auszeichnen. Ziel der Optimierung ist, eine hohe Auslastung unter Einhaltung aller geforderten Zeitbedingungen zu ermöglichen. Sie soll die Implementierung aufwendiger Modelle auf eingebetteten Controllern erlauben und durch einen geeigneten Realzeit-Nachweis die Einhaltung der kritischen Zeitbedingungen garantieren. Dadurch wird ein methodisches Vorgehen bei der Implementierung graphischer Modelle auf Systemen mit beschränkter Rechenkapazität möglich. Die erarbeiteten Konzepte sollen letztendlich durch ein Vorgehensmodell in einem modell-basierten Entwurfsprozess zur effizienten Umsetzung der Schritte *Implementierung* und *Realzeit-Analyse* integriert werden. Aus der Aufgabenstellung und den Zielen der Arbeit ergibt sich folgender Aufbau.

In Kapitel 2 werden Grundlagen zur Modellierung und Implementierung eingebetteter Realzeitsysteme vorgestellt und eine Auswahl von existierenden Werkzeuge beschrieben.

Kapitel 3 stellt die Anforderungen und Konzepte der neuen Software-Architektur vor. Die erarbeiteten Konzepte der SW-Architekturen werden sowohl aus der statischen als auch aus der dynamischen Sicht beschrieben. Das dynamische Verhalten ist besonders für die Analysierbarkeit konkreter Implementierungen wichtig und zur Vermeidung von Prioritäts-Inversion sind geeignete Strategien erforderlich, die ebenfalls in diesem Kapitel beschrieben werden.

In Kapitel 4 wird der Realzeitnachweis für die vorgestellte Software-Architektur präsentiert. Ausgehend von der Definition des verwendeten Taskmodells werden anschließend die mathematischen Zusammenhänge zur Berechnung der worst-case Reaktionszeiten einer konkreten Implementierung beschrieben.

Kapitel 5 beinhaltet die exemplarische Umsetzung der theoretischen Konzepte im Werkzeug Simulink/Stateflow. Hierbei wird zuerst die Modellierung der Software-Architektur auf graphischer Ebene beschrieben und anschließend die Implementierung des Realzeitnachweises präsentiert. Zur Integration des RT-Nachweises in einen modell-basierten Entwurfsprozess wird ein Vorgehensmodell vorgestellt und die notwendige Werkzeugunterstützung am Beispiel von Simulink/Stateflow beschrieben. Am Ende des Kapitels wird anhand eines konkreten Beispiels die Genauigkeit der Analyse demonstriert.

In Kapitel 6 werden die konkreten Anpassungen eines Realzeit-Betriebsystems (RTEMS-4.5.0) für die Umsetzung der beschriebenen Software-Architektur erläutert.

In Kapitel 7 wurde die Realisierung der beschriebenen Konzepte und das Vorgehensmodell anhand zweier Beispiele verifiziert. Im ersten Anwendungsbeispiel (Hochregallager) werden drei unterschiedliche Implementierungen derselben Funktionalität vorgestellt. Es wird dabei ein mögliches Entwicklungsszenario dargestellt, indem die konkreten Ergebnisse aus der RT-Analyse zu einem optimierten Software-Design führen. Das zweite Anwendungsbeispiel (hydrostatischer Vorderachs-Antrieb) zeigt die Anwendbarkeit der Methoden auch bei komplexeren Modellen.

Im Ausblick werden abschließend mögliche Erweiterungen der vorgestellten Ergebnisse beschrieben.

2 Grundlagen und Stand der Technik

2.1 Modellierung eingebetteter Realzeitsysteme

Ein *Modell* ist die *„Darstellung derjenigen allgemeinen und abstrakten Merkmale eines Forschungsgegenstandes, die für das Ziel der Forschung von Bedeutung sind" [Bertelsmann Universal Lexikon]*.

Die Modellierung eingebetteter Realzeitsysteme ist somit die Abstraktion der interessanten Sachverhalte eines zu entwickelnden Systems. Hierbei versteht man vor allem die graphische Modellierung der funktionalen Aspekte in einer sehr frühen Phase des Designs.

Früher wurden graphische Modelle vorwiegend in der Anforderungsanalyse oder Spezifikationsphase angewandt. Heute jedoch unterstützen weiterentwickelte Sprachen mit den entsprechenden Werkzeugen immer mehr den Entwicklungsprozess von Software-Systemen. Der Trend geht von einer funktionalen Verifikation in Form von Simulationen ausführbarer Sprachen bis hin zur Unterstützung einer automatisierten Implementierung eines realisierten graphischen Designs.

Über die Jahre hinweg hat sich für die Entwicklung von Software-Systemen eine große Anzahl domänenspezifischer, graphischer Modellierungs-Sprachen entwickelt, die jeweils unterschiedliche Vor- und Nachteile haben.

Die Wahl einer bestimmten Notation bei der Realisierung eines Modells kann eine beträchtliche Auswirkung auf Faktoren wie die Entwicklungszeit, die Korrektheit, die Effizienz und die Wartbarkeit der Endlösung haben. Die gewünschten Eigenschaften einer Notation, um für die Beschreibung von Software-Systemen in Frage zu kommen, sind:

- Genauigkeit und Eindeutigkeit, d.h. Formalität (Syntax und Semantik)
- Verständlichkeit und Anwendbarkeit
- Möglichkeit des Beschreibens und Beweisens gewünschter Systemeigenschaften
- Möglichkeit der formalen Manipulation bzw. Transformation und gleichzeitiges Beibehalten der spezifizierten Eigenschaften
- Ausführbarkeit (ausführbare Spezifikation)

Es gibt grundsätzlich *deklarative* und *operationale* Sprachen [29].

Bei der *Deklarativen-Form* oder auch *Beschreibungs-Form* wird direkt die gewünschte Eigenschaft des spezifizierten Systems beschrieben. Es wird somit das *„Was"* und nicht das *„Wie"* ausgedrückt. Sehr oft basieren Sprachen aus dieser Kategorie auf Mengentheorie, Algebra oder symbolischer Logik. Manche Sprachen können auch ausgeführt werden, um das implizite Verhalten wiederzugeben. Für gewöhnlich ist es mit dieser Art von Spezifikations-Sprachen einfacher, Eigenschaften zu beschreiben und sie zu beweisen, aber schwieriger, ein Design daraus zu realisieren. Beispiele für diese Form von Spezifikations-Sprachen sind *OBJ* von Gougen [34], *Z* von Spivey [35] und *Real-Time Logic* von Johanian und Mok [36].

Bei der *Operationalen-Form* oder auch *Befehls-Form* wird hingegen das Verhalten durch Algorithmen beschrieben. Spezifikationen sind direkt ausführbar und ermöglichen eine „einfache" Übersetzung in Computer-Prozeduren. Damit ist es möglich, die Spezifikation für bestimmte Szenarien zu simulieren.

Viele der Befehls-Formen wurden von Programmier-Sprachen abgeleitet, wie z.B. *Communicating Sequential Processes* von Hoare [38], *UNITY* von Candy und Misra [39] oder *CSR* von Gerber und Lee [40].

Eine sehr verbreitete aber eher informale Notation sind die *Data Flow Diagramms* von Ghezzi et al., Ward und Mellor. Sie ähnelt einer Funktionalen-Programmierung.

Eine weitere Klasse sehr beliebter operationaler Notationen ist zustandsbasiert, wie *Petri-Netze* [42], *Statecharts* von Harel [58], *Modecharts* von Jahanian und Mok [37], *ESL* von Ostroff und Wonham [43], *Input-Output Automata* von Lynch und Tuttle [44], *Communicating Real-Time State Machines* von Shaw [45], *µ-Charts* von Scholz [63] oder *SDL (Specification and Description Language)* [46].

Tabellarische Methoden, wie z.B. die Notationen, die im Projekt *SCR* von Heninger verwendet wurden, basieren auf Entscheidungs-Tabellen und sind ebenfalls hauptsächlich imperativ und zustandsbasiert.

Eine Anzahl prominenter Systeme vereint Zustandsautomaten mit objektorientierten Methoden, wie z.B. ROOM von Selic et. al. [10].

Mit der Veröffentlichung der *Unified Modelling Language (UML)* [47] 1997 durch *die Object Management Group (OMG)* wurde eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Software-Systeme standardisiert. Sie stellt eine Zusammenfassung von *„best practice"* Notationen und Diagrammen dar und findet in immer mehr Bereichen der Software-Entwicklung Anwendung.

Bei der funktionalen Beschreibung der hier betrachteten eingebetteten Realzeit-Systeme lässt sich grundsätzlich ein *signalfluss-orientiertes* und ein *zustands-orientiertes* Verhalten erkennen. Dabei wird je nach Komplexität der unterschiedlichen Domäne zwischen einem rein zustandsorientierten, einem rein signalflussorientierten oder einem sogenannten hybriden System unterschieden. Der Übergang zwischen den System-Kategorien ist dabei fließend, da mit „Kunstgriffen" durch signalfluss-basierte Nomenklaturen gewisse zustandsorientierte Anteile eines Systems beschrieben werden können und umgekehrt.

In Abbildung 2.1 ist die Einteilung eines Systems abhängig von der eigentlichen Komplexität der zwei orthogonalen Systemteile wiedergegeben.

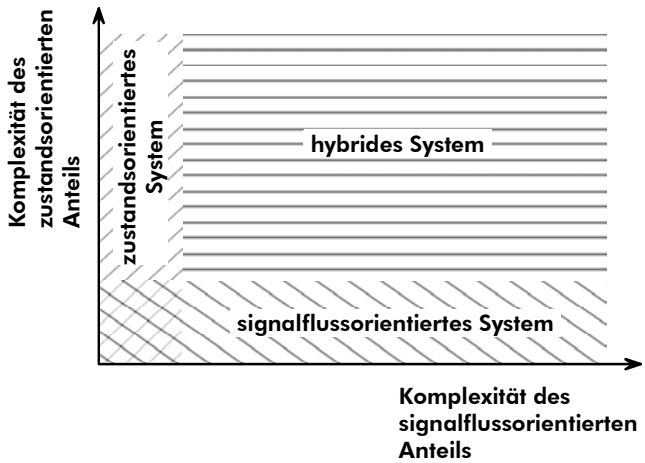

Abbildung 2.1: Kategorisierung von eingebetteten Realzeit-Systemen

Durch diese Einteilung ergibt sich die Beschränkung für die in dieser Arbeit betrachteten Spezifikationssprachen auf Signalflussgraphen für den signalflussorientierten Teil und auf Statecharts für den zustandsorientierten Teil. Für die Spezifikation hybrider Systeme wurde die Kombination beider Nomenklaturen eingesetzt.

Die Vorteile der graphischen Modellierung durch operationale Sprachen ist eine höhere Abstraktion der Funktion, eine Hierarchisierung und Partitionierung des Designs. Zudem kann eine frühe Validierung durch Simulation gewünschter Szenarien durchgeführt werden (virtuelles Testen). Durch die evolutionäre Entwicklung *(Refinements)* können Änderungen der Anforderungen leicht in die Entwicklung mit aufgenommen werden.

Herausforderungen bei der Umsetzung in ein reales Software-System ergeben sich jedoch durch die Divergenz zwischen dem funktionalen Design der Applikation und der Implementierung auf einer physikalischen Architektur.

Modellierungssprachen verwenden nämlich vereinfachende Annahmen für das Verhalten und das Zusammenarbeiten von Komponenten. So existiert zum Beispiel entsprechend der Semantik einer Notation eine unverzögerte, augenblickliche und perfekte Kommunikation zwischen unterschiedlichen Komponenten. Die Gleichzeitigkeit von Interaktionen mit der Umwelt ist ebenfalls in der Theorie möglich, aber in der Realität ist sie nur mit kaum vertretbarem Aufwand umsetzbar. Die Atomarität von Aktionen, d.h. das Verständnis von ununterbrechbaren Einheiten, ist ebenfalls nur in bestimmten Fällen zu realisieren.

Es fehlen heute geeignete Methoden zur „korrekten" Implementierung solcher high-level Konstrukte. Das Ziel dieser Arbeit ist es, durch geeignete Implementierungs-Methoden und einer speziellen Software-Architektur die Auswirkungen dieser vereinfachenden Annahmen bei den gewählten Notationen zu minimieren bzw. diese im Design zu berücksichtigen.

2.2 Implementierungsvarianten von Software für Realzeitsysteme

Grundsätzlich gibt es heute zwei Design-Paradigmen zur Entwicklung von Realzeitsystemen [5]. Zum einen das zyklische, Zeit-getriebene Ausführungsmodell *(Time-Driven Style)* und zum anderen das Ereignis-getriebene Bearbeitungsmodell *(Event-Driven Style)*.

Bei der *zyklischen Verarbeitung* wird ein sehr einfaches Laufzeitsystem angenommen. Zeit-Ereignisse, die zyklisch auftreten, stoßen periodisch die Bearbeitung verschiedener Aktionen an. Sind die Berechnungen abgeschlossen, wird auf ein neues Zeit-Ereignis gewartet. Besonders verbreitet ist diese Art der Verarbeitung bei der Software-technischen Umsetzung von kontinuierlichen Regleralgorithmen *(Direct Digital Control, DDC)*. Die Untersuchung des erhaltenen Realzeitverhaltens [96] [98] ist seit vielen Jahren etabliert.

Die unterschiedlichen Periodendauern in einem System ergeben sich dabei durch die Transformations-Vorschriften der im kontinuierlichen Zeitbereich entworfenen Regleralgorithmen auf eine zeitdiskrete Umsetzung [7]. Die geforderten Deadlines für die Berechnungen sind meistens auch die Periodendauern der Zeit-Ereignisse.

Bezüglich der Auslastung des Systems ist diese Art der Umsetzung auch für bestimmte Ereignisgetriggerte Systeme effizient, solange die durch den einbettenden Prozess vorgegebene maximale Reaktionszeit auf ein Ereignis mit der minimalen *Inter-Arrival-Time* zwischen unterschiedlichen Instanzen desselben Ereignisses übereinstimmt. Wird nämlich zyklisch das Eintreffen eines Ereignisses abgefragt (pollen), das zwischenzeitlich z.B. in der Hardware gepuffert wird, entspricht die Periodendauer des Testens der maximalen Verzögerung bis zum Start der geeigneten Reaktion der Software. Somit muss die Abtastrate auf alle Fälle kleiner sein als die geforderte Reaktionszeit. Trifft jedoch das Ereignis nur sehr selten ein (große *Inter-Arrival-Time*, z.B. Aufprall des Fahrzeuges, Ausnahmebehandlung eines Hardware-Systems etc.) ist die erzeugte Rechenlast, durch das zyklische Abfragen ob ein Ereignis eingetroffen ist, sehr viel höher als die reale durch das einbettende System beanspruchte Bearbeitung. Die realisierte Rechenlast ist *fiktiv* und nur durch die Wahl des Design-Paradigmas definiert. Ein Ereignisgetriebenes Bearbeitungsmodell könnte die spezifizierte Aufgabe viel effizienter umsetzen.

Ereignis-getriebene Bearbeitungsmodelle, oft auch als Interrupt-getriebene Laufzeitmodelle bekannt, wurden vorwiegend zur Behandlung von Systemen mit schwer vorhersagbaren, asynchronen Ereignissen entwickelt. Diese Art von Software ist grundsätzlich durch ereignisbearbeitende Software-Komponenten aufgebaut. Ein Ereignis triggert eine Bearbeitung durch eine solche Komponente, die bei Beendigung wieder einen Ruhe-Zustand einnimmt, bis erneut ein Ereignis eintrifft.

Da Ereignisse eintreffen können, während ein anderes in Bearbeitung ist, muss die Möglichkeit für die Pufferung von Ereignissen zur späteren Bearbeitung in der Architektur vorgesehen werden. Solche Systeme reagieren auf Ereignisse aus dem einbettenden Prozess und tun dies abhängig vom aktuellen internen Zustand, der gerade vorliegt. Ihr Verhalten wird deshalb meist durch zustandsorientierte Sprachen modelliert.

Das Interrupt-getriebene Laufzeitmodell implementiert somit am effektivsten Ereignis-gesteuerte Modellteile, da dort Reaktionen und Zustandsänderungen im System nur durch Ereignisse an den Systemgrenzen des einbettenden Prozesses verursacht werden können. Die Rechenleistung eines eingebetteten Systems ist somit optimal genutzt, da nicht „umsonst", d.h. ohne neue Daten gerechnet wird. Wichtig ist dieses Design-Paradigma vor allem in Verbindung mit *Soft-Real-Time* Tasks, die zusammen auf ein und demselben Hardware-System umgesetzt werden. Durch die Verwendung von *Quality-Of-Service* Parametern bei Soft-Real-Time Aufgaben kann die verfügbare Rechenleistung eingebetteter Hardware neben der Verarbeitung der asynchronen Rechenzeitanforderung optimal genutzt werden, um z.B. die Wertschöpfung für den Kunden zu erhöhen, ohne Mehrkosten für den Hersteller zu verursachen.

In Abbildung 2.2 sind die zwei Abstraktions-Ebenen bei der modell-basierten Entwicklung eingebetteter Software-Systeme dargestellt. Die bei der echtzeitfähigen Umsetzung graphischer Beschreibungen notwendige Abbildung ist durch Pfeile gekennzeichnet, wobei die durchgehenden Linien die bereits angedeutete bevorzugte Abbildung wiedergeben.

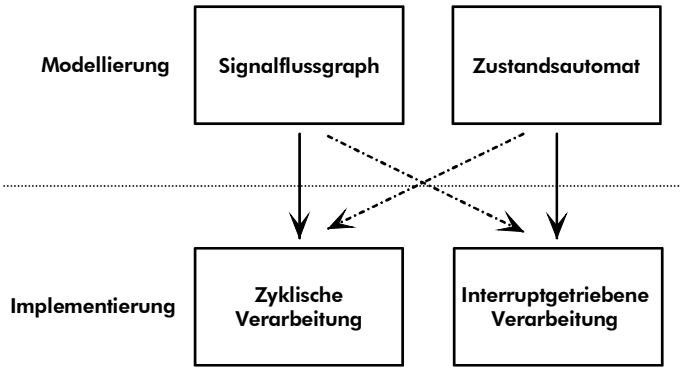

Abbildung 2.2: Abstraktions-Ebenen der modell-basierten Entwicklung eingebetteter Software-Systeme und die bevorzugte Abbildung bei der echtzeitfähigen Umsetzung.

2.3 Werkzeugunterstützung: Matlab, Simulink/Stateflow

Im Rahmen dieser Arbeit wurde zur Modellierung hybrider Systeme die integrierte Werkzeugkette Matlab, Simulink und Stateflow[4] eingesetzt. Sie bietet neben der Unterstützung der zwei gewählten Spezifikations-Notationen (Signalflussplan und Statecharts) ein erweiterbares Codegenerierungs-Konzept, womit die Umsetzung der erarbeiteten Konzepte zur verbesserten realzeitfähigen Implementierung hybrider Systeme erst möglich wurde. Ein weiterer Grund genau auf diese Werkzeugkette aufzubauen, ist die Tatsache, dass heute Simulink/Stateflow in

[4] Matlab: Version 6.0.0.88 (R12); Simulink: Version 4.0 (R12); Stateflow: Version 4.0.4 (R12);

vielen industriellen Bereichen zur Entwicklung moderner, komplexer Systeme eingesetzt wird und somit die Notwendigkeit für eine optimierte Software-Architektur gegeben ist.

Matlab ist ein umfangreiches Softwarepaket für numerische Mathematik. Der Name ist von MATrix LABoratory abgeleitet und weist auf den ursprünglichen Einsatzbereich, der Manipulation und Auswertung von Vektoren und Matrizen, hin. Es bildet das mathematische Rechenwerk für verschiedene Zusatzmodule, sogenannte Toolboxen, wie z.B. für Bildverarbeitung, Datenerfassung oder Statistik. Eine Sonderstellung nehmen die graphischen Front-Ends Simulink und Stateflow ein.

Simulink ermöglicht die Modellierung, Simulation und Analyse komplexer dynamischer Systeme in Form von Signalflussgraphen. Es hat sich vor allem bei der Entwicklung von Regelungssystemen etabliert. Neben den vielseitigen Simulationsmöglichkeiten werden auch regelungstechnische Analyseverfahren unterstützt. Block-Sets bilden die Grundbibliotheken von elementaren Blöcken, die für verschiedene Anwendungsgebiete (Signalverarbeitung, Leistungselektronik, Mechanik etc.) entwickelt wurden.

Stateflow wurde später in die Werkzeugkette integriert und dient der graphischen Beschreibung von ereignisgesteuerten Systemanteilen. Es können damit erweiterte diskrete Zustandsautomaten sehr intuitiv modelliert und simuliert werden.

Es ist möglich, abhängig von der Komplexität der jeweiligen Systemeigenschaften, zustands- oder signalflussorientiert (siehe Abbildung 2.1auf Seite 8), ausschließlich Stateflow oder Simulink für die Modellierung des Systems zu verwenden. Sind jedoch beide Systemanteile entsprechend aufwendig, liegt ein hybrides System vor und die Kopplung beider Domänen ist durch den integrierten Einsatz von Stateflow in Simulink möglich.

In den letzten Jahren wurde das SW-Paket um die Möglichkeit einer automatisierten, softwaretechnischen Implementierung der graphischen Modelle erweitert. Die derzeitigen Anstrengungen gehen in Richtung einer automatisierten Abbildung für eingebettete Zielsysteme und werden *Production-Code* Generierung genannt.

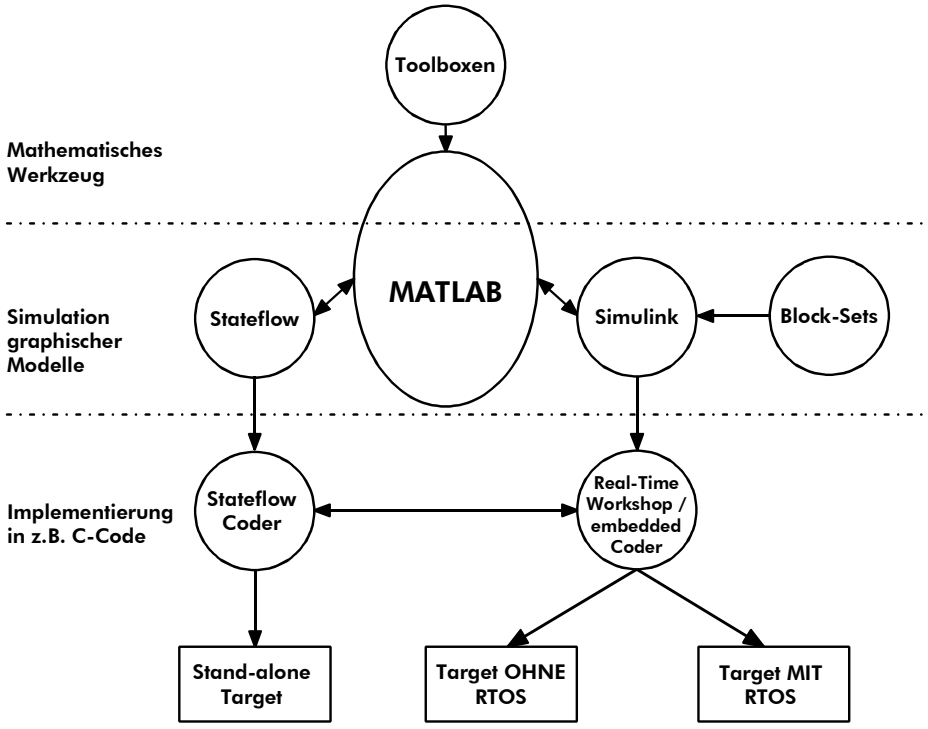

Abbildung 2.3 Übersicht Matlab, Simulink und Stateflow mit Codegenerierungs Pfad

In Abbildung 2.3 ist ein Überblick über die verschiedenen Module der Werkzeugkette und deren Zusammenspiel dargestellt. Matlab bildet das zentrale Glied mit den graphischen Front-

Ends Stateflow und Simulink. Für die Implementierung steht der Stateflow-Coder und der Real-Time-Workshop bzw. für die Gewinnung von *Production Code* zusätzlich der Embedded Coder zur Verfügung. Die bereitgestellten Zielsysteme sind in der Implementierungsebene dargestellt.

Im folgenden Abschnitt werden jeweils die Syntax und Semantik von Simulink und Stateflow beschrieben und anschließend auf derzeitige Implementierungen und deren Probleme eingegangen.

2.3.1 Kontinuierliche Systeme mit Simulink

Bei jeder Art der Modellierung wird versucht, ein System, sei es physikalischer oder anderer Natur (z.B. soziologisch etc.), in einer abstrakten Form zu beschreiben. Dabei ist es sehr wichtig, nur die für die Analyse relevanten Aspekte zu berücksichtigen.

In der Regelungstechnik hat sich der Blockschaltplan bzw. Signalflussplan als eine sehr anschauliche Möglichkeit der Systembeschreibung kontinuierlicher, linearer und nichtlinearer Systeme erwiesen [8]. Simulink stellt ein graphisches Interface dar, um genau solche Systeme modellieren, simulieren und analysieren zu können. Das Verhalten kann dabei als zeitkontinuierlich, zeitdiskret oder als Mischung beider Zeitformen modelliert werden. Anwendungsbeispiele sind elektrische, mechanische oder thermodynamische Systeme (siehe Abbildung 2.4).

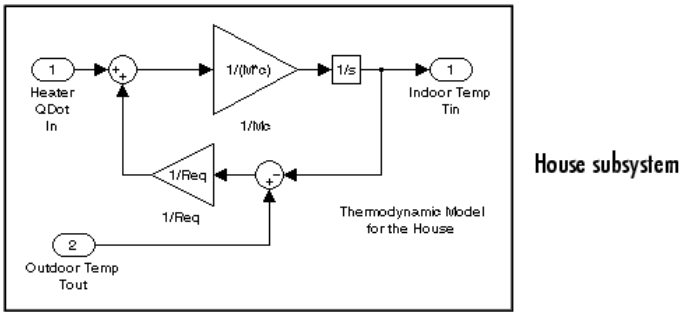

Abbildung 2.4: Beispiel eines Blockschaltplans in Simulink [71]

Im mathematischen Blockschaltplan werden einzelne Teilsysteme durch eine Reihe von Symbolen, den sogenannten elementaren Blöcken, repräsentiert. Diese sind untereinander mittels gerichteter Kanten, den Signalen, zu einem Gesamtsystem verbunden. Mathematische Blockschaltpläne verdeutlichen somit sehr gut die topologische Struktur und die physikalisch-technischen Zusammenhänge in einem System.

Jeder Block besitzt eine definierte Anzahl von Eingängen u, internen Zuständen x und Ausgängen y (siehe Abbildung 2.5). Das zeitliche Verhalten kann entweder mit Hilfe von Matlab simuliert oder in Form eines echtzeitfähigen Programms implementiert werden.

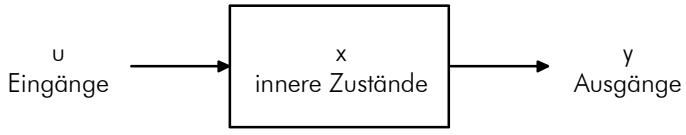

Abbildung 2.5: Elementarer Block eines Blockschaltplans

Die Reihenfolge, in der Blöcke eines Signalflussgraphen abgearbeitet werden, ergibt sich in erster Linie aus den Datenabhängigkeiten im Modell. Der Entwickler kann jedoch Einfluss darauf nehmen, indem er Blöcke in atomare Subsysteme gruppiert oder Blockprioritäten vergibt.

Grundsätzlich wird bei der Modellinitialisierung, d.h. vor der Ausführung einer Simulation oder vor der Codegenerierung, die Modellhierarchie abgeflacht und alle virtuellen Blöcke, die nur

zur besseren Strukturierung dienen, entfernt. Anschließend werden die Blöcke in einer Abarbeitungsliste sortiert. Blöcke, deren Ausgänge direkt von den eigenen Eingängen abhängen (sog. *Direkt Feedthrough Blocks*, z.B. Gain), werden am Ende der Liste eingereiht. Sie fordern die Bereitstellung ihrer Eingangssignale, bevor sie bearbeitet werden können. Alle Blöcke, deren Ausgänge nur von den internen Zuständen abhängen (z.B. Integratoren), werden hingegen am Anfang der Liste einsortiert.

Bei atomaren Subsystemen gilt das selbe Prinzip der Sortierung, jedoch mit der Zusatzbedingung, dass alle gruppierten Blöcke bei der Ausführung wie ein einziger elementarer Block zu bearbeiten sind.

Die Vergabe von Blockprioritäten hingegen beeinflusst nur die relative Lage der jeweiligen Blöcke in der Sortierliste.

Beim Simulieren ist die Abarbeitung immer sequentiell und es gibt nur eine Liste. Bei einer Realzeit-Implementierung ist hingegen eine Multitasking-Architektur möglich. In diesem Fall wird für jede Abtastrate eine eigene Abarbeitungsliste erstellt und im generierten Code umgesetzt. Um Dateninkonsistenzen beim Austausch von Signalen zwischen unterschiedlichen Raten zu vermeiden, werden spezielle Bufferblöcke *(Ratetransition Blocks)* eingesetzt.

Zusammenfassend gilt, dass die Hauptdomäne von Simulink in der Beschreibung von Systemen liegt, deren Verhalten durch mathematische Signalflussgraphen beschrieben werden kann. Es ist dabei irrelevant, ob es sich um simulierte zeitkontinuierliche Systeme handelt oder ob es um ein zeitdiskretes Abtastsystem geht, das für eine digitale Implementierung *(Direct Digital Control DDC)* auf einem Mikrokontroller entwickelt wurde.

2.3.2 Ereignisgesteuerte Systeme mit Stateflow

Neben der Modellierung kontinuierlicher oder quasikontinuierlicher Systeme, bei denen sich alle Variablen stetig über der Zeit verändern, gibt es auch eine Vielzahl von Systemen, die nur beim Auftreten bestimmter Ereignisse ihren Zustand sprunghaft verändern. Ereignisse können zum Beispiel in kontinuierlichen Prozessen von Grenzwertgebersignalen ausgelöst werden. So würde zum Beispiel der relevante diskrete Systemzustand des Prozesses erst beim Eintreten des Ereignisses „Übergang“ (von „Grenzwert noch nicht erreicht“ auf „Grenzwert überschritten“) wechseln. Solche Systeme bezeichnet man als ereignisdiskrete Systeme. Sie zeichnen sich durch eine zumeist endliche Menge klar unterscheidbarer Zustände aus, die eingenommen werden können.

Stateflow bietet die Möglichkeit, solche ereignisdiskrete Systeme in Form von erweiterten Zustandsautomaten oder Flussdiagrammen zu beschreiben. Es können somit komplexe Steuerungs- und Kontroll-Logiken entwickelt und simuliert werden. Bei Stateflow handelt es sich um eine Erweiterung von Simulink, da komplexe zustandsorientierte Problemstellungen nur sehr umständlich mit Signalflussgraphen modelliert werden können.

Die Syntax von Stateflow ist der *Statechart*-Syntax, die von Harel [58] im Jahre 1987 veröffentlicht wurde, sehr ähnlich. Durch die intuitive Gestaltung von Modellen fand die Notation von Harel eine weite Verbreitung und wurde in den unterschiedlichsten Werkzeugen (z.B.: STATEMATE [61], RoseRT, ASCET-SD) umgesetzt. Statecharts werden auch bei der Beschreibung von Verhalten in der *Unified Modelling Language* (*UML* [47]) eingesetzt.

Ein Stateflow Diagramm besteht aus den in Tabelle 2.1 aufgelisteten Grundelementen. Es ist mit den zur Verfügung stehenden syntaktischen Elementen möglich, Hierarchie, Parallelität und History in einem Diagramm darzustellen.

Befinden sich in einem Diagramm keine Zustände, sondern nur Transitionen und Verzweigungspunkte, so entspricht dies einem Flussdiagramm.

In Simulink Modellen wird ein Stateflow Diagramm durch einen speziellen Stateflow Block dargestellt. Dieser Block erlaubt es, Zustandsdiagramme im restlichen Modell zu integrieren. Er kann über Signale (Eingangs- und Ausgangsdaten) oder über Ereignisse (Trigger Ein- und Ausgänge) mit den restlichen Simulink Blöcken kommunizieren.

Tabelle 2.1: Grundelemente eines Stateflow Diagramms

graphische Elemente	nicht-graphische Elemente
• Zustände (states) • Transitionen (transitions) • Verzweigungspunkte (junctions)	• Daten (data) • Ereignisse (events)

Die Zusammenfassung aller Stateflow Blöcke eines Simulink Modells stellt die Stateflow Maschine dar. In Abbildung 2.6 ist die Integration von zwei Stateflow Blöcken (Chart1 und Chart2) eines Modells in die Stateflow Maschine dargestellt.

Die Ausführung eines Stateflow Charts wird durch Ereignisse (Events) getriggert. Diese kommen entweder aus Simulink (externe Events) oder entstehen in der Stateflow Maschine selbst. Für letztere, sogenannte interne Events, gibt es verschiedene Quellen. Sie entstehen durch aktives Versenden oder werden implizit (automatisch) beim Eintreten bzw. Verlassen eines Zustandes aber auch beim Verändern von bestimmten Variablen *(change event)* generiert. Durch das implizit definierte Abtastereignis kann die Bearbeitung eines Zustandsautomaten auch zyklisch bzw. zeitgesteuert erfolgen.

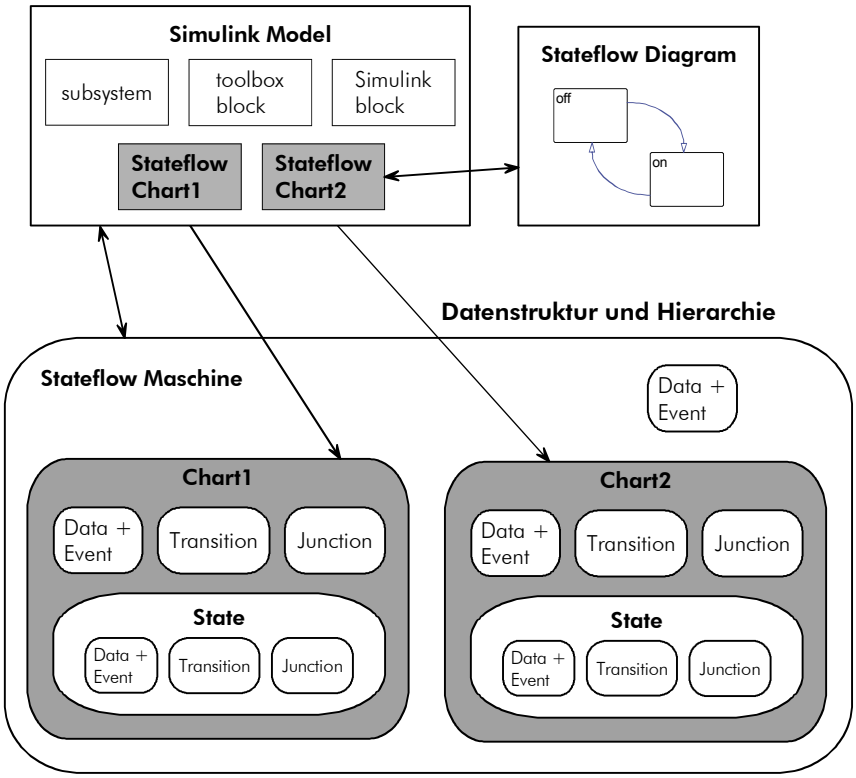

Abbildung 2.6: Integration von Stateflow Blöcken in ein Simulink Modell und die korrespondierende hierarchische Datenstruktur der Stateflow Maschine

Während einer Simulation oder einer Echtzeitausführung werden die verschiedenen Charts nur beim Eintreten relevanter Ereignisse ausgeführt.

Daten und Events sind ab der Hierarchie, in der sie definiert werden, für alle darunter liegenden Einheiten sichtbar. Zum Beispiel entsprechen Variablen, die auf der Maschinenebene

definiert sind, globalen Daten. Ähnlich verhält es sich für Ereignisse der höchsten Ebene. Sie triggern beim Versenden immer alle Diagramme der Stateflow Maschine. Ein globales Ereignis modelliert also ein *„global broadcast"* im ereignisdiskreten System. Auf diesem Weg und über Signale in Simulink können unterschiedliche Charts miteinander kommunizieren und sich gegenseitig synchronisieren.

Die semantische Interpretation der Statechart Syntax hat nach der Veröffentlichung sehr viel Interesse in der Forschungswelt geweckt und trotz der weiten Verbreitung gibt es bis heute dafür noch kein eindeutiges Verständnis. Etliche Forschungsgruppen beschäftigen sich mit der Definition von geeigneten Semantiken [62][63][66]. Die Unterschiede liegen dabei vor allem in der Interpretation von Parallelität und der gleichzeitigen globalen Kommunikation zwischen den Zuständen (z.B.: „Wann werden Änderungen für das System sichtbar?").

Die Semantik von Stateflow ermöglicht eine einfache softwaretechnische Implementierung der Modelle, erschwert aber dem Entwickler durch ihre Komplexität die einfache Vorhersagbarkeit des Systemverhaltens. Es ist daher sehr wichtig, eine genaue Simulation der Modelle durchführen zu können, um das exakte Verhalten des modellierten Systems zu verifizieren.

Grundsätzlich kann jedes Stateflow Chart nur ein Event nach dem anderen verarbeiten. Mehrere Ereigniseingänge für ein und dasselbe Diagramm werden bei ihrer Definition im Modell priorisiert. Treten zum Beispiel in einem Zeitschritt (d.h. gleichzeitig) zwei Ereignisse für das selbe Chart auf, werden sie eines nach dem anderen, entsprechend ihrer Definitionsreihenfolge, durch den Zustandsautomaten bearbeitet. Dasselbe gilt auch bei der Echtzeitimplementierung. Diese Art der ununterbrechbaren Ereignisbearbeitung wird auch mit *„run-to-completion"* Abarbeitungsmodell bezeichnet, da die Bearbeitung eines Ereignisses durch den Zustandsautomaten nicht durch ein anderes Ereignis für den selben Zustandsautomaten unterbrochen werden kann, sondern zu Ende bearbeitet werden muss. Wird dies nicht garantiert, treten Dateninkonsistenzen im Chart auf und ein unvorhersagbares Verhalten ist die Folge (siehe Kapitel 2.3.4).

Die Systemzeit bleibt während der Abarbeitung von solchen externen Ereignissen eines selben Zeitschrittes unverändert. Die Semantik fordert nämlich, dass die Bearbeitung eines Ereignisses durch den Automaten keine Zeit in Anspruch nimmt. Zeit vergeht nur in einem stabilen Zustand des Automaten.

Die Abarbeitung eines Zustandsautomaten aufgrund eines bestimmten Ereignisses erfolgt immer von der höchsten Hierarchieebene nach innen. Dadurch wird es möglich, den sehr effektiven Interrupt-Mechanismus der Statecharts auch bei Stateflow Automaten zu nutzen. Mittels einer Transition in der höchsten Ebene können alle darunter liegenden Zustände verlassen werden, was einem Interrupt gleich kommt.

Anders verhält es sich bei UML Charts. Dort wird im Inneren mit der Bearbeitung eines Ereignisses begonnen. Dadurch kann das Paradigma der „Überladung von Funktionen", das Teil der objektorientierten Sichtweise ist, leichter implementiert werden. Nehmen wir z.B. an, es gibt in verschiedenen Hierarchieebenen eines Modells mehrere Transitionen, die auf dasselbe Ereignis ERROR sensitiv sind, so besitzt jede dieser Transitionen zusätzlich eine Aktion, die beim Auftreten des Event ERROR ausgeführt werden soll. Nun ist verständlich, dass, je tiefer die Transition in der Hierarchie ist, um so genauer kann die Error Handling Aktion definiert werden. Dies entspricht auch der Überlagerung von Methoden einer generischen Klasse durch verfeinerte abgeleitete Unter-Kinderklassen. Daher werden hier von den tiefergelegenen Ebenen der Hierarchie ausgehend alle Transitionen bearbeitet, um die „übergeordneten" eventuell zu überlagern.

Bei der Definition von parallelen Zuständen in einem Stateflow Diagramm erhalten die jeweiligen Blöcke eine Nummer im oberen rechten Eck. Diese Nummer gibt Aufschluss über die Reihenfolge, in der die pseudoparallelen Automaten abgearbeitet werden. Da Stateflow für

eine rein softwaretechnische Implementierung auf einem Single-Prozessor System konzipiert wurde, gibt es keine wirklich parallele Umsetzung.

Aktionen in einem Zustandsautomaten erhalten sofort ihre Gültigkeit. Wird an einer beliebigen Aktionsstelle (z.B. Transition, Entry, Exit usw.) im Automaten eine Variable verändert, so ist dies sofort für alle anderen sichtbar. Genauso verhält es sich, wenn ein Event versendet wird. Die Bearbeitung des aktuellen Ereignisses wird unterbrochen, um das versendete Ereignis zu bearbeiten. Es handelt sich somit immer um ein synchrones, unterbrechendes Senden (siehe Problem der Prioritätsinversion in Kapitel 2.3.4). Dieser Sachverhalt kann mit der Verschachtelung synchroner Funktionsaufrufe verglichen werden (siehe Abbildung 2.7).

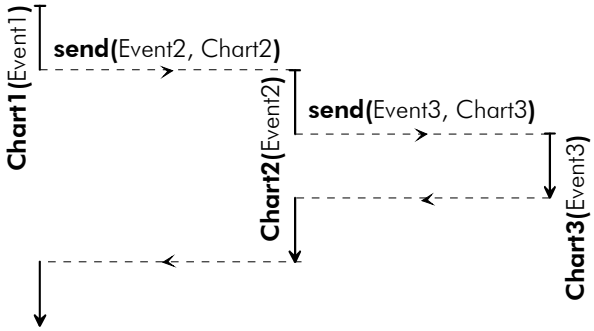

Abbildung 2.7: Abarbeitungsreihenfolge beim Versenden zweier lokaler Ereignisse

Daraus ergibt sich, dass beim Versenden eines globalen Events alle Charts der Stateflow Maschine getriggert werden. Dabei wird auch der Zustandsautomat, von dem aus der globale Event versendet wurde, durch sich selbst erneut getriggert. Es findet ein rekursiver Aufruf statt.

Der Einsatz von Ereignissen kann somit sehr unübersichtliche Auswirkungen haben und erfordert äußerste Sorgfalt des Entwicklers.

2.3.3 Umsetzung graphischer Spezifikationen

Im folgenden Abschnitt werden die verschiedenen, derzeitig möglichen Implementierungsformen einer graphischen Spezifikation aus Simulink und Stateflow erläutert. Ziel ist es, dann im darauffolgenden Kapitel die Defizite, die sich daraus ergeben, aufzuzeigen, um die in dieser Arbeit entwickelte SW-Architektur zu motivieren.

Durch die Forderung nach einer zentralen ausführbaren Spezifikation ergibt sich in einem modell-basierten Entwurfsprozess die Notwendigkeit, in jeder Phase der Entwicklung Code für das Modell zu generieren und zur Ausführung zu bringen. Dabei werden, je nach Designphase, unterschiedliche Anforderungen an das ausführbare Modell bzw. den generierten Code gestellt.

Simulation

Während der frühen Simulationsphase soll die Funktion des Designs verifiziert werden. Hier ist es vor allem wichtig, dass das Modell Algorithmen und die Semantik von Notationen getreu wiedergibt. Zeitliche Anforderungen sind nicht relevant, da während der Simulation die Simulationszeit von der realen Zeit entkoppelt ist.

Bei Simulink und Stateflow untersucht zu Beginn einer jeden Simulation der Simulator das gesamte Modell und ermittelt entsprechend den Regeln aus Kapitel 2.3.1 eine geeignete Abarbeitungsreihenfolge aller Blöcke. Je nach Zusammensetzung des Modells und dem gewählten numerischen Integrationsverfahren kann die Schrittweite der Simulationszeit unterschiedlich sein.

Da der Simulator „Herr über die Zeit" ist, können auch Algorithmen mit variabler Abtastrate Verwendung finden. Besonders bei Modellen mit kontinuierlichen Systemteilen kann durch eine stetige Anpassung der Schrittweite an die augenblickliche Dynamik des Systems in den meisten Fällen eine Beschleunigung der Simulationsdurchläufe, bei gleichbleibender relativer Genauigkeit der Ergebnisse, erreicht werden. Nach jedem Berechnungsschritt des kontinuierlichen Systemanteiles wird der neue Abtastzeitpunkt für die nachfolgende Berechnung bestimmt.

Bei Systemteilen mit festen Abtastraten stehen die Berechnungszeitpunkte bereits a priori fest.

Triggern „gleichzeitig" mehrere Ereignisse einen Zustandsautomaten, so werden alle im theoretisch selben Zeitpunkt behandelt (Reihenfolge siehe Kapitel 2.3.2). Die Bearbeitung scheint dadurch unendlich schnell zu sein.

Die Simulation von asynchronen Komponenten bzw. Subsystemen, die bei einer Echtzeitimplementierung durch externe Interrupts getriggert werden, kann durch Function-Call Generator Blöcke erfolgen. Diese werden synchron im Simulationsablauf der zyklischen Anteile integriert.

In Abbildung 2.8 ist exemplarisch die Berechnung eines Modells mit zwei unterschiedlichen, festen Sampleraten (T_{a1} und $T_{a2} = 2 \cdot T_{a1}$) über der Zeit aufgetragen.

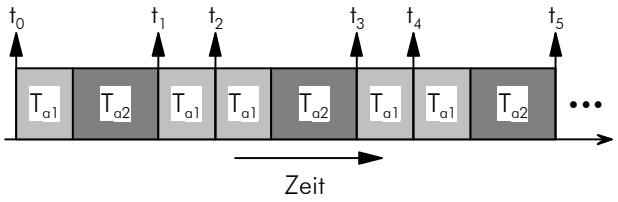

Abbildung 2.8: Berechnungsablauf von Multirate Modellen in Simulink während einer Simulation (feste Abtastintervalle)

Die Simulationszeitpunkte $t_0 = 0 \cdot T_{a1}$, $t_1 = 1 \cdot T_{a1}$, $t_2 = 2 \cdot T_{a1}$ usw. sind nicht mit der realen Zeit (x-Achse) synchronisiert. Für die Berechnungen zwischen t_0 und t_1 wird eine konstante Simulationszeit von t_0 angenommen, erst mit dem nächsten Berechnungsschritt wird die virtuelle Zeit weitergezählt. Somit können beliebig viele Berechnungen zum selben Zeitpunkt durchgeführt werden.

Realzeitimplementierung

Die Realzeitimplementierung einer graphischen Spezifikation zeichnet sich durch eine schritthaltende Verarbeitung der entwickelten Algorithmen aus. Der Hauptunterschied zu einer Simulation liegt in der Auffassung von Zeit. In einer Simulation ist die Zeit eine unabhängige Variable und wird durch den Simulator verwaltet. Bei einer Echtzeitimplementierung wird die Zeit durch die einbettende Umwelt vorgegeben. Es entstehen erstmals harte Zeitanforderungen für die Softwareimplementierung.

Da für die Berechungen nur ein beschränktes Zeitintervall zur Verfügung steht, kann zur Implementierung keine variable Abtastrate mehr verwendet werden. Der verwendete numerische Integrationsalgorithmus muss ebenfalls mit konstanten Abrastraten arbeiten (z.B. Runge-Kutta, Euler, etc.). Falls das zu implementierende System im kontinuierlichen Bereich modelliert wurde, gibt es umfangreiche Theorien [7], die eine Transformation auf ein zeitdiskretes System beschreiben.

Auch der Umfang an Blöcken im Modell und deren Komplexität ist für eine schritthaltende Implementierung durch die endliche Rechenkapazität beschränkt.

Die grundlegenden Design-Paradigmen für die Umsetzung realzeitfähiger Software wurden in Kapitel 2.2 beschrieben und werden auch zur Implementierung bei Simulink/Stateflow eingesetzt. Dabei ist, wie bereits beschrieben, die zyklische Verarbeitung von signalflussorientierten

Modellteilen aus Simulink und die Interrupt-getriebene Bearbeitung von zustandsorientierten Anteilen durch Stateflow eine intuitiv ersichtliche Art der Abbildung.

Je nach Anwendungsfall kann jedoch auch das zyklische Bearbeiten von Zustandsautomaten oder die Interrupt-getriebene Berechnung von Signalflussgraphen sinnvoll sein.

Abbildung zyklischer Modellanteile

Die Realisierung einer zyklischen Softwarestruktur ist bei der Implementierung eingebetteter Systeme eine sehr übliche Vorgehensweise. Der Werkzeughersteller stellt Software-Frames zur Verfügung, die eine automatisierte Abbildung zyklischer Modelle ermöglichen. Der Anwender soll dadurch motiviert werden, durch geringfügige Anpassungen am Frame, den automatisch generierten Code für ein eigenes, meist problemspezifisches Zielsystem anzuwenden. Die vorgeschlagenen Softwarestrukturen benötigen von der Hardware nur die Bereitstellung eines zyklischen Interrupts, um die Bearbeitung periodisch anzustoßen. Die unterschiedlichen Implementierungsfälle für Modelle mit zyklischen Blöcken werden im Folgenden kurz beschrieben.

Im *Singlerate-Fall* werden alle Blöcke des Modells mit derselben konstanten Abtastrate T_a bearbeitet (siehe Abbildung 2.9).

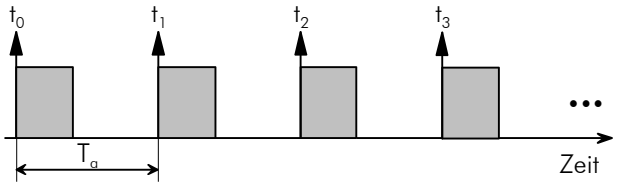

Abbildung 2.9: Zeitverlauf der Echtzeitimplementierung eines Singlerate Modells

Die Softwarestruktur entspricht einer Single-Thread Architektur und wird durch einen periodischen Interrupt getriggert. Damit keine Realzeitverletzung auftritt, müssen alle Berechnungen innerhalb des Abtastintervalls abgeschlossen sein. Die Deadline entspricht somit der Zeit T_a. Die Systemzeit t schreitet diskret voran und ist während des gesamten Zeitintervalls der Periode konstant. Es gilt folgende Bedingung: $t = k \cdot T_a$ mit $k \in \aleph_{+,0}$ (positive natürliche Zahl mit Null).

Im *Multirate-Fall* setzt sich das Modell aus Blöcken mit mehreren unterschiedlichen Abtastraten zusammen. Es existieren zwei verschiedene Umsetzungen, die *Single-* und die *Multi-Thread* Implementierung.

Bei einer *Single-Thread* Implementierung wird wieder nur eine Abtastrate implementiert. Das System wird mit dem größten gemeinsamen Teiler aller enthaltenen Abtastraten getriggert. Zu jedem Zeitschritt werden aber immer nur die im Augenblick relevanten Modellteile berechnet (siehe Abbildung 2.10).

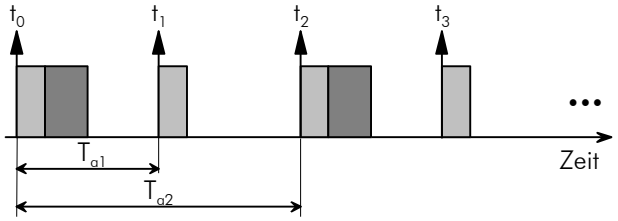

Abbildung 2.10: Zeitverlauf der Echtzeitimplementierung eines Multirate Modells bei einer Single-Thead Implementierung

Die Berechnung der jeweiligen Blöcke muss innerhalb der gewählten Abtastperiode beendet sein, andernfalls gibt es eine Realzeitverletzung. Die Deadline entspricht hier wieder der einzigen implementierten Abtastrate, dem größten gemeinsamen Teiler aller Sampleraten.

Bei der beschriebenen Singel-Thread Implementierung wird keine Parallelisierung oder Priorisierung der Berechnungen unterschiedlicher Modell-Abtastraten vorgenommen und somit eine zu eng bemessene Deadline für die Berechnungen gefordert.

Beim Einsatz einer *preemptiven Multi-Thread* Software-Architektur werden dagegen die Berechnungen unterschiedlicher Abtastraten parallelisiert und für Systemteile mit einer großen Abtastperiode wird somit auch die entsprechend längere Deadline herangezogen. Die Priorisierung erfolgt nach dem *Rate Monotonic Scheduling Schema* [96][105] und garantiert eine optimale Auslastungsgrenze (bei Scheduling mit festen Prioritäten). Die Abarbeitungsreihenfolge für ein Beispiel mit drei unterschiedlichen Sampleraten ist in Abbildung 2.11 dargestellt.

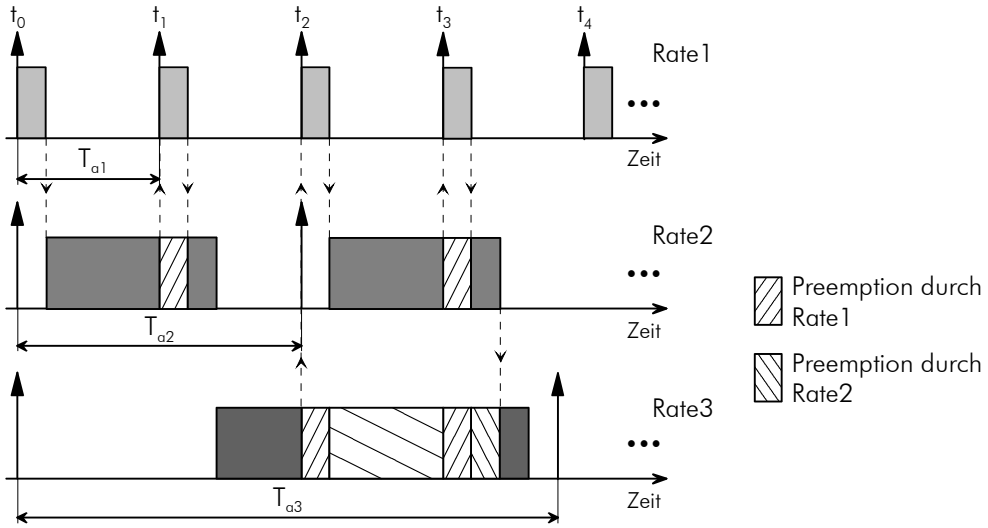

Abbildung 2.11: Zeitverlauf einer Multiratethread Implementierung eines Multirate Modells mit drei unterschiedlichen Abtastraten

Abbildung ereignisgesteuerter Modellanteile

Die Umsetzung von Interrupt- bzw. ereignisgetriebenen Software-Architekturen ist im Bereich der eingebetteten Systeme zwar ebenfalls sehr wichtig, aber bei weitem nicht so verbreitet und beliebt wie die zyklische Bearbeitungsform. Der heutige Haupteinsatzbereich solcher Komponenten liegt in der Anbindung von Hardware durch Treibersoftware. Interrupt- bzw. ereignisgetriebene Modellteile werden nur zu bestimmten, extern definierten Zeitpunkten abgearbeitet.

Ein solcher asynchroner Bearbeitungswunsch wird dem System durch Hardware Interrupts (IRQs) mitgeteilt. Auch hier wird üblicherweise eine Deadline definiert, die entweder durch den kleinsten Zeitabstand zwischen zwei aufeinanderfolgende Interrupts oder durch die späteste Reaktion des Systems auf eine Berechnungsanforderung festgelegt ist (*„end-to-end response“* aus der nichtfunktionalen Spezifikation).

Für die softwaretechnische Umsetzung gibt es hier bedeutend weniger Möglichkeiten. Bei einem Target ohne RTOS ist nur die Anbindung des Interrupt-getriebenen Modell-Codes als Interrupt Service Routine (ISR) an die IRQ Bearbeitungsarchitektur möglich.

Bei Targets mit RTOS gibt es zusätzlich die Möglichkeit, den ereignisgetriebenen Modell-Code als separate Task mit fester vordefinierter Priorität zu implementieren, die durch einen Semaphor und eine entsprechende ISR getriggert wird. Dadurch ist es möglich, die Abarbeitung nicht auf Hardware Interrupt-Ebene, sondern auf Software-Ebene durchzuführen. Die Interrupt-Maskierzeiten sind im Vergleich zur ersten Variante geringer, was ein besseres Interruptverhalten ermöglicht.

In Tabelle 2.2 sind die beschriebenen Implementierungsvarianten mit den dazugehörigen Software-Architekturen noch einmal zusammengefasst.

Die Multi-Thread Software-Architektur für ein Target mit RTOS zur Abbildung hybrider graphischer Spezifikationen, ist in Abbildung 2.12 dargestellt. Multirate Modelle mit einem Interrupt-getriebenen Modellteil (ISR + Task) können automatisiert darauf abgebildet werden. Im nachfolgenden Abschnitt werden die damit verbundenen Probleme erläutert.

Tabelle 2.2: Aufstellung der Implementierungsvarianten in Simulink/Sateflow und Software-Architekturen

Target	**Software-Architektur**			
	zyklisch		**ereignisgesteuert**	
ohne RTOS	Single-Thread	pseudo Multi-Thread	ISR	
mit RTOS	Single-Thread	Multi-Thread*	ISR	Task mit Semaphor*

(*: siehe Abbildung 2.12)

Grundsätzlich gelten die hier beschriebenen softwaretechnischen Aspekte sowohl bei der Implementierung auf einem Rapid Prototyping System als auch auf einem eingebetteten Zielsystem. Die Unterschiede liegen vor allem in der mehr oder weniger aufwendigen Umsetzung und der sich letztendlich ergebenden Auslastung des Systems.

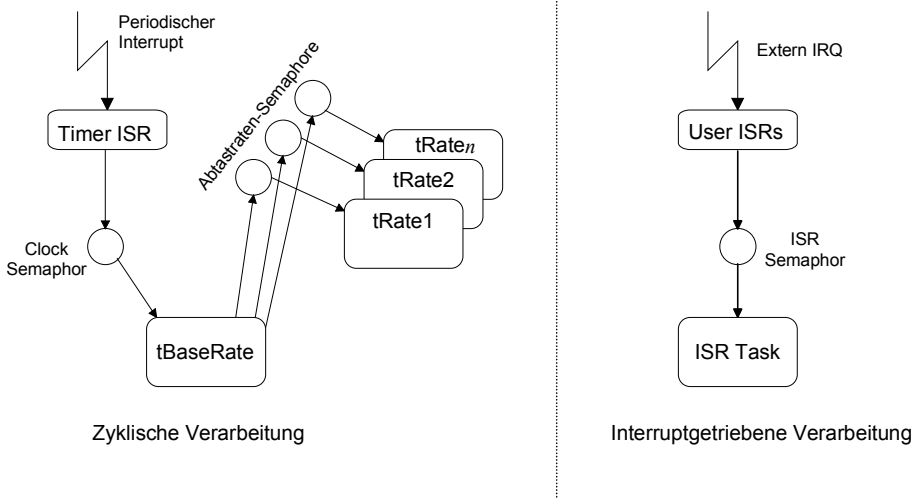

Abbildung 2.12: Software-Architektur eines Targets mit RTOS im Falle eines Multirate Modells mit einem Interrupt-getriebenen Modellteil und einer ISR Task zur Verkürzung der Interruptlatenzzeiten

2.3.4 Probleme bei der derzeitigen Implementierung

Die derzeit eingesetzte Software-Architektur (siehe Abbildung 2.12) entstand in erster Linie zur Implementierung zyklischer Mutirate-Modelle aus Simulink. Die Erweiterung der Modellierungsmöglichkeiten in Richtung ereignisdiskreter Systeme durch Stateflow stellt jedoch neue Anforderungen an die Software-Architektur, die im Augenblick nicht angemessen erfüllt werden können. Für eine ressourcenschonende, automatisierte Abbildung hybrider Modelle auf eingebettete Systeme ist daher eine erweiterte Software-Architektur erforderlich.

Die neuen Anforderungen entstehen vor allem aus den Konzepten der Verarbeitung von Zustandsautomaten, die, verglichen mit der Verarbeitung von Signalflussgraphen, grundlegend unterschiedlich sind. Ein Stateflow Chart bearbeitet immer nur ein Ereignis nach dem anderen und darf dabei nicht durch neue Ereignisse unterbrochen werden. Durch dieses *run-to-completion* Schema und dem asynchronen Charakter von Ereignis-Quellen kann es durchaus

vorkommen, dass gleichzeitig mehrere Ereignisse zur Bearbeitung durch einen Chart bereit stehen, ohne dass dadurch ein Laufzeitfehler signalisiert wird. Die dafür notwendige Art der Ereignispufferung ist jedoch in der aktuellen Implementierung nicht vorgesehen. Nachfolgend sind einige der gravierendsten Defizite der aktuellen Implementierung dargestellt.

Dateninkonsistenzen durch Kommunikation während einer Preemption

Zur Zeit muss ein Ereignis für ein Chart sofort durch den Automaten bearbeitet werden. Tritt während der Bearbeitung ein neues externes Ereignis für denselben Zustandsautomaten auf und wird dieser unterbrochen, gibt es eine Realzeitverletzung oder es kommt zu Dateninkonsitenzen, da die Software nicht reentrant ist.

Diese Art des Fehlers kann auch bei zyklischen Zustandsautomaten auftreten, wenn sie mit unterschiedlichen Sampleraten getriggert werden. In Abbildung 2.13 ist ein solcher Sachverhalt wiedergegeben. *Chart1* und *Chart2* sind zwei zyklisch bearbeitete Zustandsautomaten. Die Bearbeitung beider Automaten wird somit durch ein jeweiliges *sampleEvent* angestoßen, zusätzlich kann *Chart1* über das Ereignis *triggerEvent* mit *Chart2* kommunizieren.

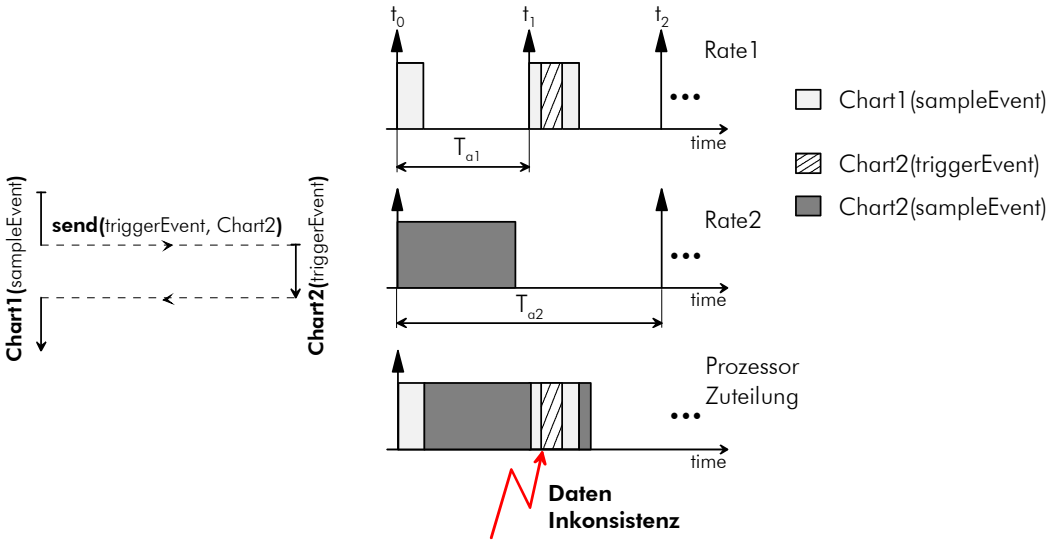

Abbildung 2.13: Dateninkonsistenz-Problem bei der Kommunikation zweier zyklischer Zustandsautomaten mit unterschiedlichen Abtastraten

Durch die unterschiedlichen Abtastraten werden die Automaten bei der derzeitigen Implementierung automatisch auf zwei Tasks mit unterschiedlichen Prioritäten abgebildet. Findet nun während der Bearbeitung des *sampleEvents* (bei t_0) durch *Chart2* eine Preemption durch *Chart1* statt (bei t_1) und sendet dieser bei der Bearbeitung seines *sampleEvents* das Ereignis *triggerEvent*, wird *Chart1* erneut getriggert, obwohl es noch bei der Bearbeitung seines ersten *sampleEvents* ist. Dies führt zu einer Inkonsistenz in der Datenstruktur von *Chart2*.

Zur Zeit kann diese Art von Fehler bei zyklischen Automaten nur durch folgende Einschränkungen vermieden werden:

- zwangsläufige Implementierung auf einer Single-Tasking SW-Architektur
- Verbieten unterschiedlicher Abtastraten von Charts die über Events kommunizieren
- Verbieten einer Kommunikation zwischen Charts mittels Events (Empfehlung durch das MAAB[5] [69])

[5] *MathWorks Automotive Advisory Board* (MAAB) wurde 1998 auf Anfrage führender Automobilhersteller ins Leben gerufen. Es hat das Ziel den Informationsaustausch zwischen Tool-Anwender im automobilen Bereich und dem Werkzeugentwickler (The MathWorks) zu fördern [69].

Besonders verheerend sind die Auswirkungen solcher Laufzeitfehler, da sie während einer sequenziellen Simulation nicht erkannt werden. Diese Art von transiente Fehler würde wahrscheinlich auch nicht bei einem Rapid Prototyping System auftreten, da erst bei der Implementierung auf einem eingebetteten Target die Auslastung des Systems zur gegenseitigen Preemption führen würde. Somit sind solche Fehler erst in dieser späten Implementierungs-Phase der Entwicklung, in der für gewöhnlich die Kosten zur Beseitigung sehr hoch sind, bemerkbar.

Schlechte Nutzung der Systemressourcen zum Garantieren der geforderten Reaktionszeiten

Eine gravierende Einschränkung der existierenden Software-Architektur betrifft den Realzeitnachweis des Systems. Dadurch, dass es keine Ereignispufferung gibt, ist die Deadline für die Bearbeitung eines Ereignisses durch einen Zustandsautomaten gleich dem minimalen Zeitabstand zwischen zwei Ereignissen aus der Menge aller Ereignisse, die diesen Zustandsautomaten triggern können.

Diese Deadlines sind ausschließlich auf die derzeitige Implementierung zurückzuführen und entsprechen auf keinem Fall den Anforderungen aus der physikalischen Umwelt oder der nichtfunktionalen Spezifikation des Systems. Zum Teil sind diese Zeiten sehr stark von der augenblicklichen Auslastung des Systems abhängig, wie z.B. im zuvor beschriebenen Szenario. Diese Implementierungs-Architektur erlaubt somit keine effektive Nutzung der Ressourcen und ist deshalb für eingebettete Realzeitanwendungen nicht geeignet.

Ungünstige Verkettung von Berechnungen unterschiedlicher Wichtigkeit

Ein inhärentes Problem der aktuellen Echtzeitimplementierung ist die Prioritätsinversion, die beim synchronen Versenden von Ereignissen auftritt. Die Implementierung eines Sende-Befehls erfolgt im Modell-Code nämlich als synchroner Funktionsaufruf.

Beim Versenden von Ereignissen wird der eigentliche Sende-Chart von der Bearbeitung des aktuellen Events abgehalten. Es ist möglich, dass durch das Senden einer weniger wichtigen Nachricht an einen anderen Automaten die aktuelle Bearbeitung des dringenden Ereignisses um eine undefinierte Zeit verzögert wird (siehe Abbildung 2.7).

Dieser Sachverhalt bereitet zwar bei der Simulation keine Probleme, da dort, wie bereits mehrmals erwähnt, die Zeit in der Hand des Simulators liegt, aber bei einer realen Implementierung kann eine solche Inversion bereits ernsthafte Auswirkungen auf die Einhaltung der Realzeitbedingungen haben.

Es entsteht eine enge Verkopplung eines Großteils der Zustandsautomaten durch direkte Funktionsaufrufe. Damit wird eine Bestimmung von worst-case Reaktions-Zeiten (*end-to-end-response*) sehr schwierig.

Für eine Realzeitimplementierung wäre die Bereitstellung einer Möglichkeit zur Priorisierung von Ereignissen viel günstiger. Dadurch könnten Zustandsautomaten Events an andere Charts versenden, ohne selbst bei der Bearbeitung unterbrochen zu werden. Erst nach dem Ende der Bearbeitung ihres höher-prioren Ereignisses würden dann die Empfangscharts mit der Bearbeitung ihrer Ereignisse beginnen. Durch eine solche Art des asynchronen Versendens von Ereignissen könnte das Problem der Prioritätsinversion entschärft werden.

Das Ziel dieser Arbeit ist es, eine neue SW-Architektur zu entwickeln, um eine effektivere automatisierte Abbildung hybrider Modelle zu unterstützen. Dabei sollen Kriterien wie eine mögliche Realzeitanalyse der worst-case Reaktions-Zeiten, als auch eine optimierte Auslastung der eingebetteten Ressourcen berücksichtigt werden.

Da die vorgestellte Software-Architektur nur durch den Einsatz eines Realzeit-Betriebsystems effektiv umsetzbar ist, wird die Implementierung auf ein Target mit RTOS betrachtet.

2.4 Andere CASE-Werkzeuge für eingebettete Systeme

Nachfolgend ist eine beschränkte Auswahl von weiteren CASE-Tools, die zur Modellierung und Entwicklung eingebetteter Systeme eingesetzt werden können, beschrieben.

ARTiSAN RealTime Studio (ARTiSAN Software Tools, Inc.) ist eine Entwicklungsplattform für eingebettete Systeme, die vorwiegend auf UML 1.4 basiert. Durch die Erweiterung um einige spezielle Notationen *(Concurrency Diagram, Constraints Diagram, System Architecture Diagram, Table Relationship Diagram, General Graphics Diagram, Text Diagram)* soll vor allem die Entwicklung eingebetteter Realzeit-Systeme unterstützt werden. Aus dem entwickelten Modell kann durch Codegeneratoren für Java, C++, C und Ada ein Skelett generiert werden, das als Basis für die reale Implementierung dient. Eine komplette automatisierte Umsetzung des graphischen Modells ist somit nicht möglich.

Rational Rose RealTime (Rational Software, IBM) ist eine Software Entwicklungsumgebung, die für die Modellierung von Realzeit-Systemen zugeschnitten ist. Die verwendeten Notationen sind UML basiert. Das Werkzeug unterstützt eine vollständige Codegenerierung des gesamten Modells für die Sprachen Java, C++ oder C. Damit kann die modellierte Anwendung vollständig und automatisiert in eine echtzeitfähige Implementierung transformiert werden. Algorithmen, die durch Statecharts nicht beschrieben werden können, müssen als Source-Code mit in das Modell integriert werden.

Telelogic Tau (Telelogic AB) ist ein CASE-Tool, das die Modellierung eingebetteter Systeme mittels des neuen UML 2.0 Standards ermöglicht. Die Verabschiedung des UML 2.0 Standards durch die OMG ist im Laufe des Jahres 2003 geplant. Das Werkzeug unterstützt die Codegenerierung für C++ und C. Teile die nicht explizit durch die bereitgestellten Notationen beschrieben werden können, müssen ebenfalls, wie bei RoseRT, als Source-Code (C++/C) im Modell integriert werden.

AutoFocus [55] ist ein CASE-Tool, das bei einem Software-Technik Praktikum an der TU München entwickelt wurde und jetzt sowohl durch die TU München als auch der Firma Validas weiterentwickelt wird. Es unterstützt die Modellierung, Simulation, Validierung und Codegenerierung von eingebetteten Systemen. Modelle können sowohl in Ada als auch in ANSI-C umgewandelt werden. Durch die formale Basis der Notation können diverse automatisierte Untersuchungen des gewonnenen Designs durchgeführt werden.

Die bis hier beschriebenen Werkzeuge bieten alle keine Unterstützung beim Entwurf regelungstechnischer Systeme. Diese Teile eines hybriden Systems müssten in einem anderen Werkzeug entwickelt und als externer Source-Code in das graphische Modell importiert werden. Eine Analyse der Reglereigenschaften im Gesamtsystem ist somit nicht mehr möglich. Weiteres Defizit dieser Werkzeuge ist das Fehlen einer Realzeit-Analyse der letztendlich generierten Implementierung. Lediglich RoseRT bietet durch einen Drittanbieter (Tri-Pacific Software, Inc.) ansatzweise eine integrierte Realzeit-Analyse *(Rate Monotonic Anaysis, RMA)* an. Die Hauptanwendung dieser Werkzeuge liegt in der Entwicklung von ereignisgesteuerten Systemen unter Verwendung objekt-orientierter Methoden.

ASCET/SD (Etas GmbH) hingegen unterstützt neben der Beschreibung von Zustands-Maschinen auch die Modellierung von Regelungssystemen durch Blockdiagramme. Es wird vor allem in der Automobilindustrie zur letztendlichen Implementierung von Regelungen und Steuerungen eingesetzt und unterstützt die C-Codegenerierung der beschriebenen Modelle. Durch die Berücksichtigung spezieller Eigenschaften eines Zielsystems bei der Codegenerierung kann die Größe und die Ausführungszeit einer automatisch gewonnenen Implementierung sehr stark optimiert werden. Obwohl eine Beschreibung regelungstechnischer Zusammenhänge im Werkzeug möglich ist, werden Entwurfsmethodiken aus der Regelungstechnik [8] nur zum Teil unterstützt. Die Folge ist, dass in der Industrie während der Entwicklungsphase bevorzugt etablierte *Computer Aided Control System Design* Tools (CACSD, z.B. Simulink + Toolboxen) eingesetzt werden und ASCET/SD nur für die letztendliche Umsetzung auf die

Zielplattform verwendet wird. ASCET/SD bietet auch keine Unterstützung für eine Realzeit-Analyse der resultierenden Software-Implementierung.

Ptolomy II [56] ist ein Werkzeug, das im Rahmen verschiedener Forschungs-Projekte entstanden ist und zur Modellierung eingebetteter Systeme konzipiert wurde. Besonderes Augenmerk wurde dabei auf die Kombination unterschiedlicher Technologien und deren integrierte Simulation gelegt. Beispiele für solche Technologien sind: eingebettete Software, spezielle digitale Hardware, konfigurierbare Hardware, analoge Signalverarbeitung, Hochfrequenz-Schaltungen und mikroelektromechanische Systeme (MEMS). Die integrierte Modellierung aller Teile soll durch die Unterstützung vieler domänenspezifischer Sprachen und Nomenklaturen ermöglicht werden. Die automatisierte Implementierung von eingebetteter Software ist bei der Entwicklung des Werkzeuges nicht das Hauptziel und somit ist es für den Einsatz in dieser Arbeit nicht geeignet.

Giotto [94][95] ist kein richtiges Werkzeug, sondern eine werkzeugunterstützte Methodik zur Entwicklung eingebetteter Regelungs-Systeme für sicherheitsrelevante Anwendungen mit harten Echtzeit-Bedingungen. Es bietet ein Programmier-Modell, das auf das Zeit-getriggerte Paradigma aufbaut. Zur Funktions-Modellierung wird Simulink verwendet. Giotto bietet ein Abstraktions-Layer, das zwischen dem mathematischen Modell (high-level) eines Regelungs-Systems und dem darunter liegenden plattformspezifischen ausführbaren Code liegt. Diese Zwischenschicht soll eine einfachere Validierung und Verifikation der Software erlauben, als es auf dem plattformspezifischen Level möglich wäre. In diesem Forschungs-Projekt wird zwar unter anderem die Validierung der entgültigen Implementierung betrachtet, aber der Hauptnachteil bei der Anwendung der vorgeschlagenen Architektur liegt für hybride Systeme in der ausschließlichen Unterstützung des zeitgesteuerten Implementierungs-Modells und in der damit verbundenen ineffizienten Umsetzung ereignisgesteuerter Anteile.

3 SW-Architektur hybrider graphischer Spezifikationen

Die Verwendung von abstrakten, graphischen Beschreibungsformen zur Funktionsspezifikation ermöglicht nicht nur eine frühzeitige Simulation von Verhalten, sondern verbirgt auch implementierungstechnische Details und Fragestellungen vor dem Entwickler. Ziel eines modell-basierten Entwurfsprozesses ist letztendlich immer eine realzeitfähige Implementierung der graphischen Modelle, bei der dann diese verborgenen Details sehr wohl gelöst werden müssen.

Im nachfolgenden Kapitel werden die Anforderungen an eine generische Software-Architektur für hybride graphische Spezifikationen zusammengefasst. Anschließend werden die Konzepte der vorgestellten SW-Architektur beschrieben. Am Ende dieses Kapitels wird das notwendige Laufzeitsystem zur Umsetzung der vorgestellten SW-Architektur erläutert.

3.1 Anforderungen

Die Anforderungen an eine realzeitfähige Software-Architektur für die automatisierte Implementierung hybrider graphischer Spezifikationen ergeben sich aus unterschiedlichen Bereichen.

Zum Einen soll eine Verkopplung der Semantiken der verwendeten Modellierungssprachen, unter Berücksichtigung ihrer unterschiedlichen zeitlichen Anforderungen, ermöglicht werden.

Zum Anderen ist es erwünscht, dass die resultierende SW-Architektur einen mathematischen Realzeitnachweis in Form einer worst-case Reaktionszeit Berechnung zulässt. Dabei dürfen die berechneten worst-case Zeiten auf keinen Fall kleiner sein als die in der realen Umsetzung vorzufindenden Reaktionszeiten. Eine Aussage über die Güte des Nachweises ist die Übereinstimmung zwischen realen und berechneten worst-case Werten, je kleiner die Überschätzung ist, desto besser ist der Nachweis.

Eine weitere nicht weniger wichtige Anforderung ergibt sich aus dem modell-basierten Entwurfsprozess, und zwar die gewünschte Übereinstimmung zwischen der funktionalen Simulation einer Spezifikation und dem letztendlichen Verhalten ihrer realzeitfähigen Implementierung durch die neue SW-Architektur.

3.1.1 Kopplung unterschiedlicher Laufzeitmodelle

Die in Kapitel 2.3.3 beschriebenen zwei Arten von Laufzeitmodellen (*models-of-computation*) sind die

- *zyklische* und die
- *ereignisgesteuerte*

Verarbeitung. Beide Anteile können zwar in der vorhandenen SW-Architektur (Abbildung 2.12) umgesetzt werden, führen aber in ungünstigen Situationen zu einem Fehlverhalten, bzw. zu unnötigen Laufzeitverletzungen (Kapitel 2.3.4).

Zyklisches Laufzeitmodell

Der *zyklische Anteil* fordert von der unterlagerten SW-Architektur lediglich die Einhaltung der Abtastrate und die Konsistenz der Eingangsdaten innerhalb dieses Zeitfensters. Die Periodendauer entspricht dabei der *Deadline* für die jeweilige Bearbeitung eines Modellschrittes. Da alle Abtastschritte nacheinander abgearbeitet werden, können verschiedene Trigger-Ereignisse nie gleichzeitig um die Bearbeitung durch eine Task konkurrieren. Sollte die Bearbeitung eines Abtastzeitpunktes einmal nicht vor dem Eintreffen eines nachfolgenden Samples bearbeitet worden sein, bedeutet dies eine eindeutige Deadline-Verletzung und zeigt, dass eine schritthaltende Verarbeitung in diesem Fall nicht möglich war.

Ereignisgesteuertes Laufzeitmodell

Der *ereignisgesteuerte Anteil* hingegen ist für die Bearbeitung mehrerer ihm zugeordneter, periodischer oder auch sporadischer Ereignisse zuständig, die beliebige Deadlines (auch kürzer als ihre Periode) haben können. Durch seine *run-to-completion* Semantik müssen alle Berechnungsanforderungen von außen zuerst serialisiert werden, was eine Zwischenpufferung der Ereignisse durch die unterlagerte Software-Architektur unumgänglich macht.

Die Bearbeitung eines Ereignisses durch den ereignisgesteuerten Anteil im Kontext seiner Task darf somit nicht durch ein anderes Ereignis aus demselben Puffer unterbrechbar sein, da sonst Dateninkonsistenten auftreten können. Für Ereignisse aus demselben Puffer ist die zugeordnete Server-Task also *nicht-preemptiv*. Diese Eigenschaft ist besonders wichtig, wenn man Blockierzeiten für die Realzeitanalyse betrachten möchte, da auch diese sogenannten reaktiven Anteile vorgegebene Zeitanforderungen einhalten müssen.

Beide Systemanteile, *zyklisch* und *ereignisgesteuert*, sollen in der neuen SW-Architektur gekoppelt werden und müssen gleichzeitig um die Zuweisung des Prozessors konkurrieren. Besonders wichtig ist, dabei eine geeignete Prozessor-Zuweisungsstrategie zu finden, um jeweils der „richtigen" zeitkritischen Aufgabe den Prozessor zuzuweisen. Nur so ist eine hohe, effektive Auslastung des Target-Systems bei Einhaltung aller Realzeitanforderungen, zu erreichen.

3.1.2 Möglichkeit eines Realzeitnachweises

Im vorhergehenden Abschnitt wurde die Semantik von ereignisgesteuerten Modellierungssprachen am Beispiel von Statecharts/Stateflow mit dem *run-to-completion model of computation* erläutert und auf die Notwendigkeit einer Ereignispufferung in der neuen SW-Architektur eingegangen.

Bei der methodischen Bestimmung von worst-case Reaktionszeiten oder dem Nachweis der Einhaltung von geforderten *end-to-end* Zeitbedingungen von beliebigen hybriden graphischen Spezifikationen ist die Berücksichtigung des Verhaltens von Ereignispuffern und anderen Strukturen der SW-Architektur von essenzieller Bedeutung.

Besonders wichtig ist in diesem Umfeld die Realisierung und Bestimmung von vorhersagbaren, deterministischen maximalen Obergrenzen für die Blockierung von Ereignissen in den verwendeten Puffern. Zu diesem Zweck sind besondere Vorkehrungen bzw. Algorithmen für die Verwaltung von Ereignissen in den Warteschlangen zu definieren und umzusetzen. Das Ziel dieser Algorithmen ist die weitgehende Entkopplung unterschiedlicher Prioritätsebenen und die Definition von maximalen Schranken der Blockierzeiten.

3.1.3 Übereinstimmung Simulation und Implementierung

Eine der Grundideen eines modell-basierten Entwurfsprozesses ist die Abstraktion der Funktion eines Systems auf eine höhere Ebene, der sogenannten Modellebene. Die dabei verwendeten

Notationen erlauben eine einfache Beschreibung komplexer Sachverhalte. Durch die Verwendung von operationalen Beschreibungsformen [29][6] bei der Modellierung kann die spezifizierte Funktion anhand einer Ausführung des Modells sogar verifiziert werden. Da es bei dieser Art der Simulation ausschließlich um eine funktionale Verifikation geht, spiegelt das simulierte Verhalten nur die Semantik der verwendeten Spezifikationssprache wieder.

Der Einsatz von ausführbaren Modellen ermöglicht eine neue Art der Entwicklung mit frühem Feedback über das Verhalten des Systems und die Möglichkeit einer sukzessiven Verfeinerung des Designs (*Refinement*).

Für die letztendliche, realzeitfähige Implementierung einer graphischen Spezifikation ergibt sich daraus die Notwendigkeit der weitgehenden Übereinstimmung zwischen dem im Vorfeld simulierten funktionalen Verhalten und dem Verhalten, das sich aus der realzeitfähigen Umsetzung durch die SW-Architetktur ergibt. Trotz der Parallelisierung und Priorisierung von Modellteilen in der Echtzeitimplementierung darf die simulierte, gewünschte Funktion nicht unnötig verfälscht werden.

Diese Anforderung kann leider nicht generisch für jede graphische Spezifikation, einfach durch die Wahl einer bestimmten SW-Architektur, garantiert werden. Die Möglichkeit einer rückwirkungsfreien Implementierung, d.h. einer vollkommenen Übereinstimmung zwischen Simulation und RT-Implementierung, hängt sehr stark von dem jeweiligen Design des abzubildenden Modells ab. Grundvoraussetzung ist die Möglichkeit der Trennung zwischen

- *Funktionalen-Aspekten* und
- *Implementierungs-Aspekten*

des Modells bzw. des Designs. Die Funktionalen-Aspekte eines Modells betreffen ausschließlich die umgesetzte Funktion des beschriebenen Systems. Die Implementierungs-Aspekte, wie z.B. die Abbildung auf unterschiedliche Prioritäten im RTOS oder die konkreten Ausführungszeiten auf unterschiedlichen Rechnerarchitekturen, sollten dagegen nur die unterlagerte SW-Architektur berühren.

Ist eine Trennung bei einem konkreten Design möglich, wird durch die Veränderung von Implementierungs-Aspekten nicht das funktionale Verhalten der Implementierung beeinträchtigt, sondern lediglich die nichtfunktionalen Eigenschaften beeinflusst, wie z.B. die maximale Antwortzeit oder die Blockierzeit von Ressourcen. Ein solches Design wird auch als ein „*gutes Design*“ bezeichnet [113].

Bei der Entwicklung einer neuen realzeitfähigen SW-Archtitektur kann somit auf keinem Fall garantiert werden, dass unabhängig vom konkret abzubildenden Design (graphischen Modell) keine Rückwirkungen durch konkrete Implementierungsentscheidungen (z.B. Zuordnung von Prioritäten im System, etc.) auf die spezifizierte Funktion entstehen. Eine vollständige Übereinstimmung zwischen simuliertem und echtzeitfähigem Verhalten kann deshalb nicht generell gewährleistet werden, besonders da in beiden Fällen unterschiedliche Laufzeitsysteme verwendet werden.

Um dem Entwickler dennoch einen Leitfaden bzw. eine Anleitung zur Erstellung eines *guten Designs* zu geben, werden nachfolgend einige Grundanforderungen an das Systemdesign genannt. Allgemein gilt, dass sich die Funktion eines *guten Designs* so wenig wie möglich auf die Eigenschaften des unterlagerten Laufzeitsystems stützen sollte.

Bearbeitungsreihenfolge von Ereignissen

Durch die Vergabe von Prioritäten in der realzeitfähigen SW-Architektur kann möglicherweise eine von der Simulation divergierende Reihenfolge bei der Bearbeitung von Ereignissen auftreten. Es könnte sogar eine Überholung von zeitlich nacheinander auftretenden Ereignissen vorkommen, um z.B. kürzere Deadlines einzuhalten. Um solche Unterschiede zwischen der Simulation und der RT-Implementierung tolerieren zu können, sollte das funktionale Verhalten des Designs unabhängig von der Bearbeitungsreihenfolge der eintreffenden Ereignisse sein.

Wenn jedoch die Funktion des Modells auf bestimmte Ereignis-Sequenzen reagieren soll, kann diese Bedingung nicht immer erfüllt werden. In solchen Fällen muss der Designer besonders auf eine geeignete Wahl der Prioritäten achten, damit bei solchen kritischen Ereignissen keine Veränderung zwischen der Eintritts- und Bearbeitungs-Reihenfolge auftritt.

Zeitliche Diskrepanzen

Die „absolute" Zeit des Systems könnte unter Umständen nicht monoton fortlaufend sein. Im Falle von sich überholenden Ereignissen, wird ein höher-priores Ereignis, das zum Zeitpunkt t_2 eingetroffen ist, vor einem nieder-priorien Ereignis, das zu einem früheren Zeitpunkt t_1 eintraf, bearbeitet. Dadurch wird die absolute Zeit, die das System erfährt, mit t_2 in der Zukunft gelegt und dann mit t_1 wieder in die Vergangenheit gerückt. Solche Diskrepanzen dürfen das funktionale Verhalten des Designs ebenfalls nicht beeinträchtigen.

Reale Systemzustände

Einer der häufigsten Gründe für die Abhängigkeit der funktionalen Spezifikation vom unterlagerten Laufzeitsystem ist die Verwendung von nicht realen Systemzuständen im Design. Nicht reale Zustände sind Zustände, die in einem Zeitschritt über eine Transition erreicht werden und im selben Zeitpunkt wieder über eine andere Transition verlassen werden. Die Umsetzung solcher Zustandsübergänge ist sehr stark vom unterlagerten Laufzeitsystem abhängig und verhindert eine eindeutige Trennung zwischen Funktionalen- und Implementierungs-Aspekten. Aus diesem Grund dürfen in einem guten Design grundsätzlich nur reale Zustände vorkommen.

Ein Zustand ist ein *realer Zustand*, wenn er durch eine Transition erreicht und nicht zum selben Zeitpunkt wieder verlassen wird. Mit anderen Worten, ein Zustand, der durch die Bearbeitung eines externen Ereignisses eingenommen wurde, darf nicht im selben „Augenblick" wieder verlassen werden. Es muss gewährleistet werden, dass nur reale Zustände im System enthalten sind.

3.2 Konzept der SW-Architektur hybrider Systeme

Als Ausgangspunkt für die neue SW-Architektur wurde die vom Werkzeughersteller bereits verwendete Architektur eingehend untersucht, um die vorhandenen Defizite zu erkennen. Die nachfolgend präsentierten Konzepte dienen dazu, die erkannten Probleme zu lösen und den in Kapitel 3.1 angeführten Anforderungen Rechnung zu tragen.

Die neuen Grundkonzepte der Architektur können wie folgt zusammengefasst werden:

- Eine Zwischenpufferung von Ereignissen für die Bearbeitung des ereignisgetriebenen Modelteiles ist essenziell.
- Jeder Berechnungswunsch im System wird auf ein Ereignis abgebildet. Dadurch wird eine einheitliche Kopplung und Betrachtung von kontinuierlichen bzw. *zeitgesteuerten* und *ereignisgesteuerten* Modellteilen auf Implementierungsebene möglich.
- Die Prioritäten der Berechnungswünsche im System werden auf die Prioritäten der Ereignisse abgebildet und sind auch für die Aktionen des Systems gültig. Die Priorität des bearbeitenden Threads wird daher ständig geändert und dem bearbeiteten Ereignis angepasst.
- Durch die Pufferung von Ereignissen mit unterschiedlichen Prioritäten und das *run-to-completion* Paradigma der Aktivitäten müssen besondere Maßnahmen für die Vorhersagbarkeit von Reaktionszeiten vorgesehen werden.

Die Software-Architektur wird nachfolgend zuerst aus einer statischen Sicht beschrieben, bevor das dynamische Verhalten erklärt wird.

3.2.1 Statische Sicht der Software-Architektur

Messagequeue und Thread

Die Kombination eines Threads mit einer Messagequeue (siehe Abbildung 3.1) bildet das Grundelement der hier vorgestellten Software-Architektur. Alle Berechnungswünsche des Systems werden in Form von Ereignissen an die Warteschlange des Threads übergeben und dort zwischengepuffert.

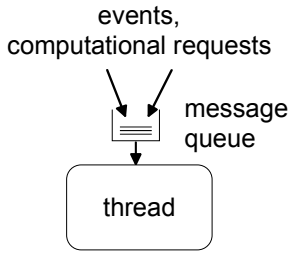

Abbildung 3.1: Grundelement der vorgestellten SW-Architektur

Der Thread bearbeitet die Berechnungswünsche (Ereignisse) nacheinander ab. Die Bearbeitung entspricht der jeweiligen Reaktion des Systems auf das konsumierte Ereignis.

Eine begonnene Bearbeitung wird dabei immer zu Ende durchgeführt, bevor der Thread mit einer neuen Aktivität beginnt. Damit ist es auf einfache Weise möglich, das *run-to-completion* Paradigma von ereignisgesteuerten Modellteilen zu realisieren und Dateninkonsistenten innerhalb des Threads zu vermeiden.

Ein weiterer Vorteil dieser Architektur ist die einheitliche Behandlung von zyklischen und ereignisgetriebenen Modellanteilen auf Implementierungsebene, vor allem beim später beschriebenen Realzeitnachweis (siehe Kapitel 4).

Ereignisse und Aktionen

Ereignisse bzw. Nachrichten (*Messages*) beschreiben alle möglichen Bearbeitungsanforderungen im System. Jedem Ereignis ist genau eine Aktion zugeordnet, die der jeweiligen Reaktion des empfangenden Threads entspricht. Dem Bearbeitungswunsch wird somit durch eine dem Ereignis zugeordnete Aktion im Empfänger-Thread Rechnung getragen.

Ereignisse können entweder durch externe Interrupts im System entstehen oder intern als Folge von anderen Aktionen propagiert werden. Wird zum Beispiel durch ein Hardware-Interruptrequest (IRQ) eine Interrupt Service Routine (ISR) getriggert, entspricht dies einem *externen Ereignis*. Versendet jedoch eine Aktivität bei der Bearbeitung eines Ereignisses eine neue Nachricht, ist diese ein *internes Ereignis*. Die Unterscheidung zwischen internen und externen Ereignissen ist vor allem bei der Realzeitanalyse von großer Bedeutung (siehe Kapitel 4.1.2).

Umsetzung zyklischer und interruptgetriebener Modellteile

Nachdem die Grundelemente der Software-Architektur kurz erklärt wurden, soll nun die übergeordnete Struktur zur Umsetzung von beliebigen hybriden Modellen dargestellt werden. Hierzu wird in der Abbildung 3.2 die SW-Architektur der Realzeitimplementierung für ein allgemeines graphisches Modell mit einem zyklischen Anteil und einem ereignisgesteuerten Anteil dargestellt.

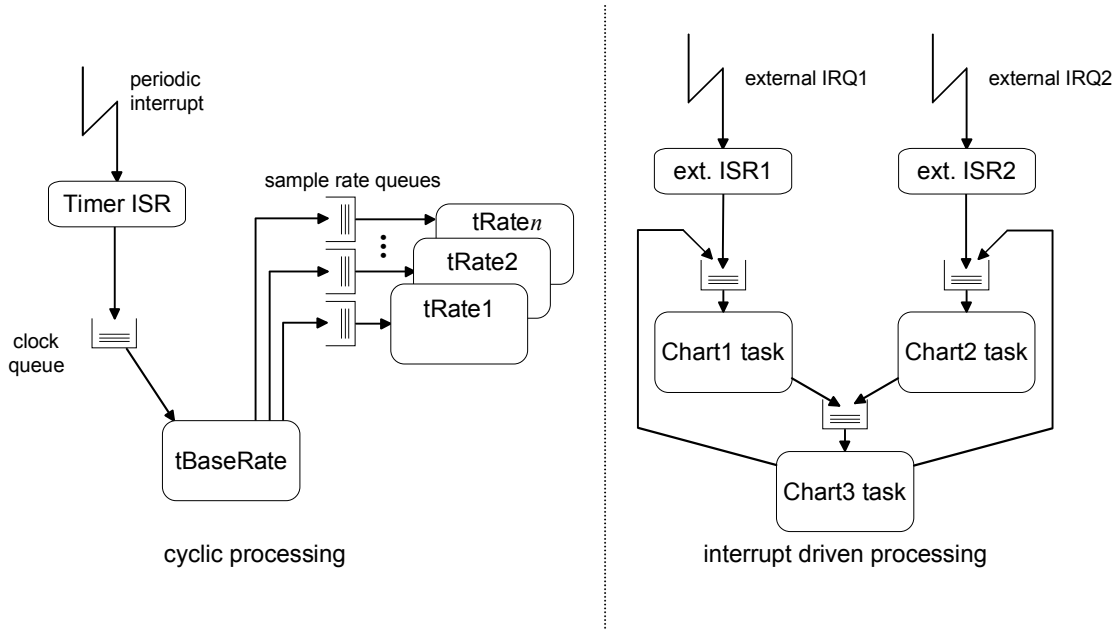

Abbildung 3.2: SW-Architektur der Realzeitimplementierung für ein allgemeines hybrides Modell

Zyklischer SW-Anteil

Der zyklische Anteil der Software-Architektur (linke Seite in Abbildung 3.2) ist grundsätzlich aus den verschiedenen festen Abtastraten des Modells zusammengesetzt. Für jede dieser Abtastraten wird bei der Codegenerierung automatisch im SW-Frame ein eigener Thread mit der dazugehörigen Messagequeue generiert (Grundelement der Architektur).

Ist die kürzeste Periodendauer im Modell der kleinste gemeinsame Teiler aller vorkommenden Sampleraten, so beinhaltet der entsprechende Thread neben dem eigentlichen Modellcode der Abtastrate auch den modellspezifischen Scheduling-Code und ist deshalb eine Art Manager-Thread des zyklischen Systemanteiles.

Für den Fall, dass die kleinste Abtastrate nicht der kleinste gemeinsame Teiler aller Sampleraten ist, wird eine zusätzliche neue Samplerate eingeführt, die lediglich die Aufgabe der Manager-Task übernimmt, also keinen spezifischen Modellcode bearbeitet. Ihre Abtastrate definiert sich dabei als der kleinste gemeinsame Teiler aller Sampleraten des Modells.

Ein periodischer Hardware Interrupt wird als Motor für die gesamte zyklische Software-Struktur verwendet. Seine Periode wird durch die resultierende kürzeste Samplerate des zyklischen Anteils definiert. In der Timer-ISR wird durch das Verschicken eines Ereignisses der Manager-Thread periodisch getriggert.

Der Manger-Thread *tBaseRate* triggert dann aus dem modellspezifischen Scheduling-Code, durch das Versenden entsprechender Nachrichten alle anderen periodischen Threads, aber nur wenn eine Bearbeitung erforderlich ist.

Um die Bearbeitungsreihenfolge der unterschiedlichen Abtastraten gemäß dem bereits beschriebenen *Rate-Monotonic-Priority* Schema zu realisieren, werden den jeweiligen Trigger-Ereignissen automatisch geeignete Prioritäten gegeben. So besitzt z.B. das Ereignis der Manager-Task die höchste Priorität des zyklischen Anteils und das Ereignis für die längste Abtastperiode die niedrigste Priorität.

Bei dem gerade beschriebenen zyklischen Software-Anteil wird grundsätzlich in der Empfangsqueue jeder Abtastrate (Thread) nur ein Typ von Ereignis empfangen. Zudem ist im Normalfall die maximale Anzahl von wartenden Ereignissen in der Empfangsqueue eines Threads nicht größer als eins. Dies kommt dadurch zustande, dass jeder Bearbeitungswunsch

in Form eines Ereignisses einem Thread zugeführt wird und binnen des nachfolgenden neuen Abtastzeitpunktes bearbeitet sein muss (Semantik zyklischer Modellteile, Kapitel 3.1.1), andernfalls würde eine Laufzeitverletzung angezeigt werden. Zur Vereinfachung können somit die Messagequeues im zyklischen SW-Anteil bei der realen Umsetzung auf ein Realzeit-Betriebssystem (RTOS) durch Semaphoren ersetzt werden (analog zum zyklischen Teil der SW-Architektur in Abbildung 2.12). Um jedoch eine einheitliche Realzeitbetrachtung in Kapitel 4 zu erlauben, soll dafür die in Abbildung 3.2 dargestellte allgemeine SW-Architektur angenommen werden.

Ereignisgesteuerter SW-Anteil

Der ereignisgesteuerte Anteil der Software-Architektur (rechte Seite, Abbildung 3.2) wird ebenfalls aus Grundelementen (Messagequeue mit Thread) zusammengesetzt, wobei sich die Struktur hierbei direkt aus dem graphischen Modell ergibt. Bei der Modellierung des ereignisgesteuerten Anteiles werden nämlich Konstrukte verwendet, die eins zu eins auf die darunter liegende Software-Architektur abgebildet werden können. Es wurde zu diesem Zweck eine Blockbibliothek entwickelt, die Komponenten wie einen *Message-Queue-Receive* Block oder einen *Send-Message* Block beinhalten (siehe Kapitel 5.1.1).

Sinngemäß wird im ereignisgesteuerten Teil eines Modells die zu realisierende Funktionalität, je nach Ereignismenge und Wichtigkeit, in logische Einheiten unterteilt. Diese logischen Einheiten werden dann auf jeweils unterschiedliche Threads abgebildet. Diese Threads sind durch das *run-to-completion* Paradigma für Ereignisse desselben Teilsystems, welche in derselben Messagequeue gepuffert werden, nicht preemptive. Aus diesem Grund ist eine vernünftige Partitionierung des Systems für die Einhaltung von *end-to-end* Zeitanforderungen von essenzieller Bedeutung. Ein Rückschluss über die Qualität der konkreten Partitionierung kann durch den Realzeitnachweis (Kaptiel 4) gewonnen werden.

Die hier vorgestellte Arbeit bietet somit keine automatisierte Lösung für das Problem der Partitionierung von ereignisgesteuerten Modellteilen auf Threads. Sie stellt vielmehr eine Software-Architektur mit dazugehöriger Analysemethodik vor, die eine quantitative Evaluierung einer speziell gewählten Partitionierung bereitstellt. Erst wenn man die Qualität einer gewählten Lösung ermitteln kann, wird eine Optimierung sinnvoll und zielführend.

3.2.2 Dynamisches Verhalten

Nachdem die statische Struktur der Software-Architektur erläutert wurde, soll nun das dynamische Verhalten beschrieben werden. Besonders wichtig ist es dabei zu erkennen, dass das dynamische Verhalten große Auswirkungen auf die Berechnung der worst-case Reaktionszeiten hat und deshalb vor allem sehr wichtig für die Realzeitanalyse der Echtzeitimplementierung von hybriden Modellen ist.

Message Sortierung in Queues und Prioritätsvererbung

Die Ereignisse in der hier vorgestellten Software-Architektur beschreiben alle Berechnungsanforderungen im System, die dann durch Aktivitäten in den einzelnen Threads realisiert werden. Da es bei der Verarbeitung in Realzeitsystemen, neben der korrekten logischen Funktion, auch um die Einhaltung von Zeitbedingungen geht, werden allen Ereignistypen im System (mögliche unterschiedliche Berechnungsanforderungen) feste Prioritäten zugeordnet. Dadurch wird eine Priorisierung der unterschiedlichen Aktivitäten im System und eine entkoppelte Betrachtung der worst-case Reaktionszeiten möglich.

Bei der Verwendung von festen Ereignis-Prioritäten in der SW-Architektur müssen verschiedene Fälle betrachtet werden.

Zum einen werden die Prioritäten beim Sortieren der wartenden Ereignisse in den Messagequeues berücksichtigt. Falls also mehrere Ereignisse in derselben Warteschlange auf die Bear-

beitung durch den Thread warten, werden diese nach den zugeordneten Prioritäten sortiert. Die dabei verfolgte Strategie ist: alle wartenden Ereignisse werden nach abfallender Priorität sortiert, wobei das Ereignis mit der höchsten Priorität an der ersten Stelle in der Queue eingereiht wird. Damit kann gewährleistet werden, dass immer das wichtigste Ereignis in der Queue als erstes bearbeitet wird.

Zum anderen werden die Nachrichten-Prioritäten auch als Thread-Prioritäten während der Verarbeitung des korrespondierenden Ereignisses verwendet. Die Ereignisse eines Systems stellen bekanntlich alle Bearbeitungswünsche mit dazugehörigen Aktionen der Threads dar. Eine Vererbung der Ereignis-Priorität an den bearbeitenden Thread ist daher nur eine logische Schlussfolgerung aus der Definition der Ereignisse (Bearbeitungswünsche). Das für gewöhnlich Thread-zentrierte Laufzeitsystem in dem Threads feste Priorität haben, wird durch diese neue Sichtweise in ein Ereignis-zentriertes System transformiert.

Das *run-to-completion* Paradigma der Aktivitäten hat zur Folge, dass möglicherweise höherpriore Ereignisse in einer Messagequeue warten müssen, während sich nieder-priore Ereignisse in der Bearbeitung durch den zuständigen Thread befinden. Um solche Prioritätsinversionen zeitlich zu begrenzen und einen Realzeitnachweis erst möglich zu machen, müssen geeignete Verfahren angewendet werden (siehe nächster Abschnitt).

Prioritätsbasiertes Scheduling von Threads

Da die festen Prioritäten der Ereignisse für die Bestimmung der aktuell aktiven Aufgabe verwendet werden, ist die neue Software-Architektur ein Ereignis-zentriertes System. Die Verwaltung der verschiedenen parallel ausführbaren Threads wird jedoch nach wie vor durch einen konventionellen, prioritätsbasierten Scheduler im Laufzeitsystem realisiert. Dies ermöglicht eine kostengünstige Umsetzung der Architektur, da kein spezielles Realzeit-Betriebsystem entwickelt werden muss. Das *run-to-completion* Paradigma der Aktivitäten und die Pufferung von höherprioren Messages in den Queues kann jedoch zu unvorhersagbaren Blockierzeiten führen und damit jede Art des Realzeitnachweises unmöglich machen.

Zur Veranschaulichung einer solchen Prioritäts-Inversions-Situation soll kurz ein mögliches Szenario, für die in Abbildung 3.3 dargestellte konkrete SW-Architektur, beschrieben werden.

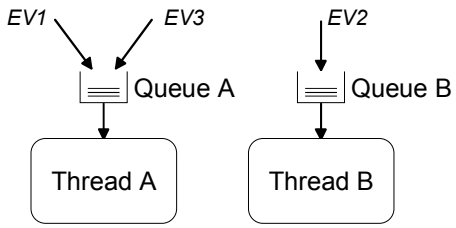

Abbildung 3.3: SW-Architektur Beispiel zum Erläutern der Prioritäts-Inversion

Für die Prioritäten der Ereignisse und ihrer zugeordneten Aktionen gilt:

$$(\text{Priorität von } EV1) < (\text{Priorität von } EV2) < (\text{Priorität von } EV3).$$

In Abbildung 3.4 ist eine mögliche Ereignis-Sequenz für das betrachtete System wiedergegeben. Zur Erklärung des Laufzeitverhaltens bei der Entstehung einer Prioritäts-Inversions-Situation wurden für jeden Thread der SW-Architektur drei Zeitachsen aufgetragen:

- eine für die eintreffenden Ereignisse,
- eine für die wartenden Ereignisse in den jeweiligen Empfangs-Queues, mit der richtigen Sortierung (nächstes zu bearbeitendes Ereignis unten im Puffer) und
- eine für die Bearbeitung einer bestimmten Aktion durch den Thread.

Vor dem Zeitpunkt t_0 ist das System im Wartezustand (*Idle-State*). Beim Eintreffen des Ereignisses $EV1^{(1)}$ zum Zeitpunkt t_0 wird direkt Aktion $A1^{(1)}$ im *Thread A* gestartet und dabei die Priorität

von *Thread A* an die Priorität von $EV1^{(1)}$ angepasst. Es findet zu diesem Zeitpunkt keine Zwischenpufferung des Ereignisses in *Queue A* statt.

Beim Eintreffen des zweiten Ereignisses $EV1^{(2)}$ zum Zeitpunkt t_1 ist *Thread A* immer noch mit Aktion $A1^{(1)}$ beschäftigt und durch das *run-to-complition* Paradigma (vermeiden von Dateninkonsistenten in *Thread A*) wird eine Zwischenpufferung von $EV1^{(2)}$ notwendig. Im Diagramm ist dieser Sachverhalt auf der Zeitachse *Queue A* erkennbar.

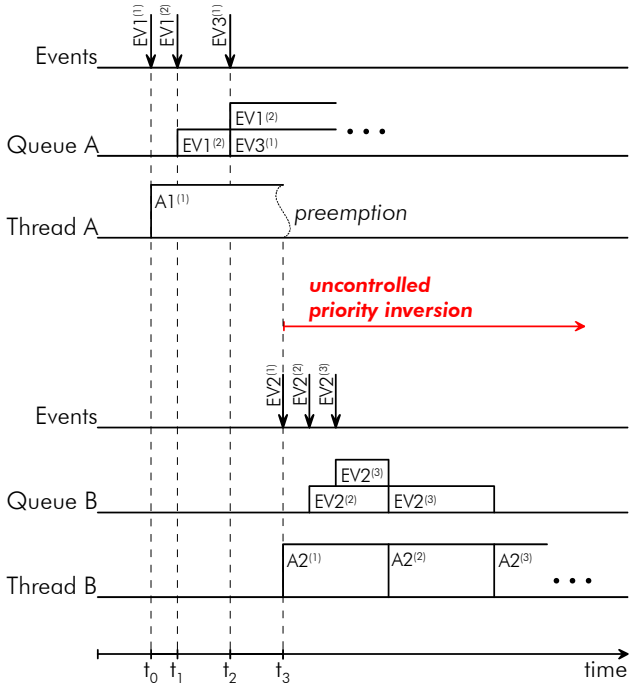

Abbildung 3.4: Zeitlicher Verlauf beim Auftreten einer Prioritäts-Inversion

Zum Zeitpunkt t_2 trifft $EV3^{(1)}$ ein, *Thread A* ist immer noch mit $A1^{(1)}$ beschäftigt und das neue Ereignis muss ebenfalls in der Warteschlange eingereiht werden. Da die Priorität von $EV3^{(1)}$ höher ist als die des wartenden Ereignisses $EV1^{(2)}$, wird $EV3^{(1)}$ in der Warteschlage vor $EV1^{(2)}$ einsortiert (siehe Zeitpunkt t_2, auf Zeitachse *Queue A*). Die Priorität von *Thread A* bleibt dabei unverändert.

Beim Auftreten des ersten Ereignisses vom Typ *EV2* zum Zeitpunkt t_3 wird *Thread B* aktiviert (*blocked* → *ready*) und wie bei $EV1^{(1)}$ zum Zeitpunkt t_0 wird seine Priorität an die Priorität von $EV2^{(1)}$ angepasst. Daraufhin wird ein Taskwechsel im Laufzeitsystem zum Starten (*running*) von *Thread B* durchgeführt, da die Priorität von *EV2* größer ist als die der aktuell durch den *Thread A* bearbeiteten Aktion *A1*. Von diesem Zeitpunkt an können beliebig viele Aktionen mit Prioritäten höher als *EV1*, aber kleiner als die der wartenden $EV3^{(1)}$, *Thread A* von der Bearbeitung abhalten und eine unkontrollierte Prioritätsinversion liegt vor (z.B. unbestimmbar viele *EV2* mit einer Priorität < als Priorität von *EV3* könnten bearbeitet werden, obwohl *EV3* wartet).

Die Konkurrenz priorisierter Ereignisse um einen Thread mit variabler Priorität entspricht dem Problem, das sonst in Thread-zentrierten Systemen bei der Verwaltung von gemeinsamen Ressourcen zwischen Threads mit unterschiedlichen Prioritäten auftritt. Hierzu wurden in der Vergangenheit geeignete Protokolle der Prioritätsvererbung entwickelt [104], die analog auch für das hier vorgestellte Prioritätsinversionsproblem angewandt werden können.

Die aktuelle Priorität der unterschiedlichen laufenden Threads wird dabei nicht nur durch die augenblicklich bearbeitete Nachricht definiert, sondern zur Vermeidung von unbegrenzten Blockierzeiten, auch durch die wartenden Ereignisse in den Queues beeinflusst. Die Verwendung unterschiedlicher Strategien bzw. Prioritätsvererbungs-Protokolle zur Bestimmung der effektiven Priorität eines Threads führt zu unterschiedlichen Blockierzeiten bzw. worst-case

Reaktionszeiten für Ereignisse. Nachfolgend werden zwei wichtige Prioritätsvererbungs-Strategien zwischen Ereignissen und den zugeordneten Threads vorgestellt.

Basic Priority Inheritance (BPI)

Beim Verwenden des *Basic Priority Inheritence Protokolls* zur Bestimmung der effektiven augenblicklichen Priorität von Threads wird die Priorität des bearbeitenden Threads immer durch die maximale Priorität aller Ereignisse, die im Augenblick durch den Thread bearbeitet werden könnten oder sich gerade in Bearbeitung befinden, definiert.

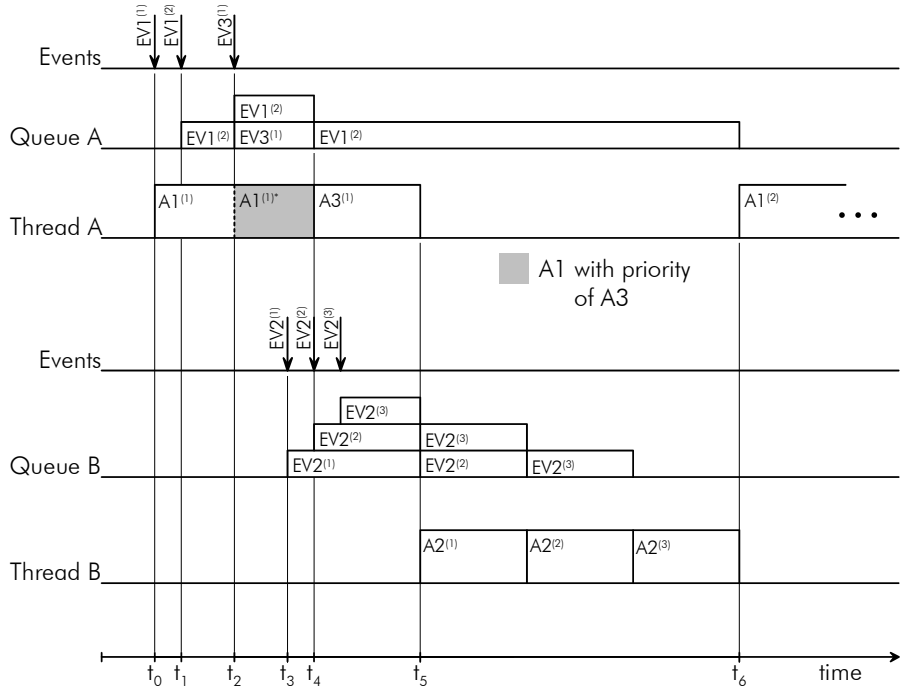

Abbildung 3.5: Zeitlicher Verlauf beim Beispiel aus **Abbildung 3.4** bei Verwendung des BPI-Protokolls

So wird beim Einreihen einer höher-prioren Message in die Queue, diese nicht nur an die erste Stelle der Warteschlange geschoben, sondern falls ihre Priorität auch höher ist als die Priorität des bearbeitenden Threads, wird dessen Priorität an die neue höchste Priorität angepasst. Sie wird sozusagen vererbt. Dadurch kann die Blockierzeit eines höher-prioren Ereignisses in der Queue durch ein nieder-priores Ereignis, das sich in Bearbeitung befindet, beschränkt werden.

In Abbildung 3.5 ist für das vorhin beschriebene Beispiel das geänderte Laufzeitverhalten aufgezeichnet. Zum Zeitpunkt t_2, wird durch das *BPI Protokoll* nicht nur $EV3^{(1)}$ in die Warteschlange einsortiert, sondern auch die Priorität des *Threads A* auf die neue Priorität von $EV3^{(1)}$ angehoben (Priorität des ersten wartenden Ereignisses ist höher als aktuelle Priorität von *Thread A*). Dadurch wird zum Zeitpunkt t_3 eine Preemption vermieden und eine unkontrollierte Blockierung bzw. Prioritäts-Inversions-Situation unterbunden.

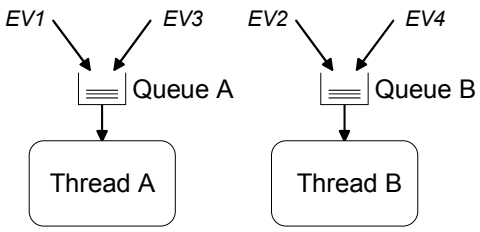

Abbildung 3.6: SW-Architektur Beispiel zum Erläutern der Verkettung von Blockierzeiten beim BPI-Protokoll

Der Nachteil dieses Protokolls ist die mögliche Verkettung von Blockierzeiten über mehrere Threads. Um diesen Sachverhalt genauer darzustellen wird nachfolgend ein weiteres Beispiel beschrieben. Die dafür gewählte konkrete SW-Architektur ist in Abbildung 3.6 abgebildet.

Für die Prioritäten der Ereignisse und ihrer zugeordneten Aktionen gilt:

$$(\text{Priorität von } EV1) < (\text{Priorität von } EV2) < (\text{Priorität von } EV3) < (\text{Priorität von } EV4).$$

In Abbildung 3.7 ist eine mögliche Ereignis-Sequenz, die zu einer Verkettung von Blockierzeiten führt, mit dem dazugehörigen Laufzeitverhalten im Falle des BPI Protokolls aufgezeichnet.

Zum Zeitpunkt t_0 tritt $EV1^{(1)}$ auf und *Thread A* beginnt mit der Bearbeitung von Aktion $A1^{(1)}$ mit der vererbten Priorität von $EV1^{(1)}$.

Beim Auftreten von Ereignisses $EV2^{(1)}$ zum Zeitpunkt t_1 findet ein Taskwechsel von *Thread A* nach *Thread B* statt, da das neue Ereignis *Thread B* nicht nur aktiviert, sondern dessen Priorität über die aktuelle Priorität von *Thread A* setzt.

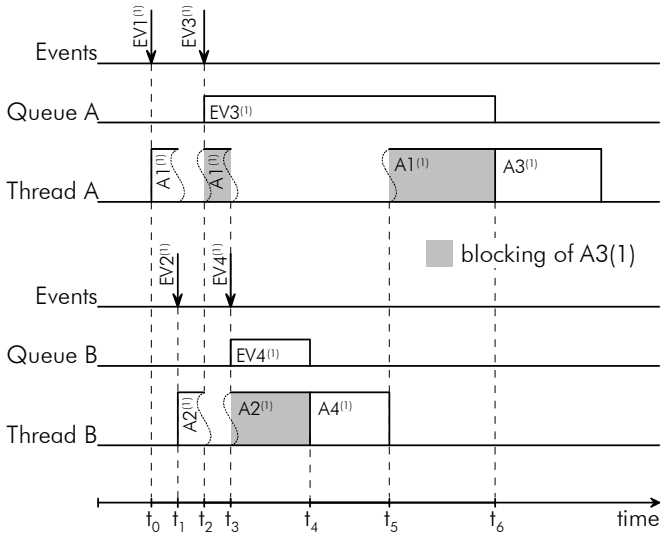

Abbildung 3.7: Zeitlicher Verlauf beim BPI-Protokoll und Verkettung von Blockierzeiten

Durch das Ereignis $EV3^{(1)}$ zum Zeitpunkt t_2 wird jedoch bei der Anwendung des BPI Protokolls nun dem *Thread A* die Priorität des wartenden Ereignisses vererbt, was wiederum einen Taskwechsel zur Folge hat. *Thread A* wird wieder aktiviert und es beginnt die Blockierung von Aktion *A3* durch die Bearbeitung von Aktionen mit Prioritäten, die unter der Priorität von *A3* liegen. Nach t_2 ist es $A1^{(1)}$.

Tritt nun zum Zeitpunkt t_3 das Ereignis $EV4^{(1)}$ ein, wird wiederum durch das BPI-Protokoll *Thread B* aktiviert, da sofort die aktuelle Priorität von *Thread B* angehoben wird. Hierdurch entsteht eine verkettete Blockierung von Aktion $A3^{(1)}$ durch die Verschachtelung über Thread-Grenzen von Prioritätsvererbungen. Aktion $A2^{(1)}$ erhält durch das BPI-Protokoll „kurzzeitig" eine Priorität, die höher liegt als die von Aktion $A3^{(1)}$. Sie kann somit zur Blockierung von $A3^{(1)}$ beitragen.

Das Zeitinterval zwischen t_4 und t_5 zählt nicht mit zur Blockierungs-Zeit von $A3^{(1)}$ (grauer Bereich). Dort wird Aktion $A4^{(1)}$ ausgeführt, deren Priorität grundsätzlich über der Priorität von *EV3* liegt und somit per Definition keine Blockierung darstellt, sondern als Interferenz mit in die RT-Analyse eingeht (siehe Abschnitt4.2.1).

Da für jeden physikalischen Thread im System immer nur genau eine solche Verschachtelung möglich ist, kann wie in Kapitel 4.2.1 beschrieben, eine eindeutige mathematische Obergrenze für die Blockierzeit bei einer gegebenen konkreten SW-Architektur berechnet werden.

Preemption Threshold (PT)

Um die Defizite des Basic Priority Inheritence Protokolls zu beseitigen, wird für die Ereigniszentrierte Architektur, analog zum Priority Ceiling Protokoll, das Preemtption Threshold Protokoll [114][111] verwendet.

Hierbei wird für jede Nachrichtenwarteschlange die höchste Priorität aller möglichen eintreffenden Ereignistypen, die sogenannte Ceiling-Priorität, bestimmt. Diese Priorität wird dann dem zugeordneten Server-Thread vererbt, sobald das Laufzeitsystem den Thread aktiviert und ihn startet.

Die Wahl des zu aktivierenden Threads durch das Laufzeitsystem wird jedoch immer noch anhand der einzelnen statischen Prioritäten der wartenden Ereignisse in den unterschiedlichen Queues bestimmt. Erst wenn diese Entscheidung getroffen wurde, läuft der aktivierte Thread mit der zuvor bestimmten Ceiling-Priorität los. Dadurch kann ihm nur durch eine höher-priore Message, höher als die Ceiling-Priority, der Prozessor entzogen werden.

Das dadurch erreichte Verhalten entspricht dem Basic Priority Inherintence Protokoll, für den Fall, dass sofort nach dem Starten der Bearbeitung eines Ereignisses, ein neues Ereignis mit der höchsten zu erwartenden Priorität in die zugeordnete Queue eingereiht wird. Die Priorität des Threads würde dann sofort auf die höchste erreichbare Priorität gesetzt.

Durch das *Preemption Threshold Protokoll* kann eine Verkettung von Verdrängungen über Thread-Grenzen hinweg vermieden und die konkreten Blockierzeiten im Vergleich zum BPI-Protokoll minimiert werden.

In Abbildung 3.8 ist das geänderte Laufzeitverhalten im Falle des *PT*-Protokolls für das zuvor beschriebene konkrete Beispiel (siehe Abbildung 3.7) dargestellt.

Die Ceililing-Priorität von *Queue A* entspricht der Priorität des Eventtyps *EV3* und die Ceiling-Priorität von *Queue B* der Priorität des Eventtyps *EV4*.

Durch das sofortige Anheben der Priorität von *Thread A* zum Zeitpunkt t_0 auf die Ceiling-Priorität von *Queue A* wird eine Verschachtelung zum Zeitpunkt t_1 vermieden. Aus der beschriebenen Ereignis-Sequenz ist auch gut ersichtlich, dass zum Zeitpunkt *t3* beim Eintreffen von $EV4^{(1)}$ eine Preemption stattfindet, da die Priorität von Aktion $A4^{(1)}$ über der aktuellen Priorität des *Threads A* von Aktion $A1^{(1)}$ (Priorität von $EV3^{(1)}$) liegt.

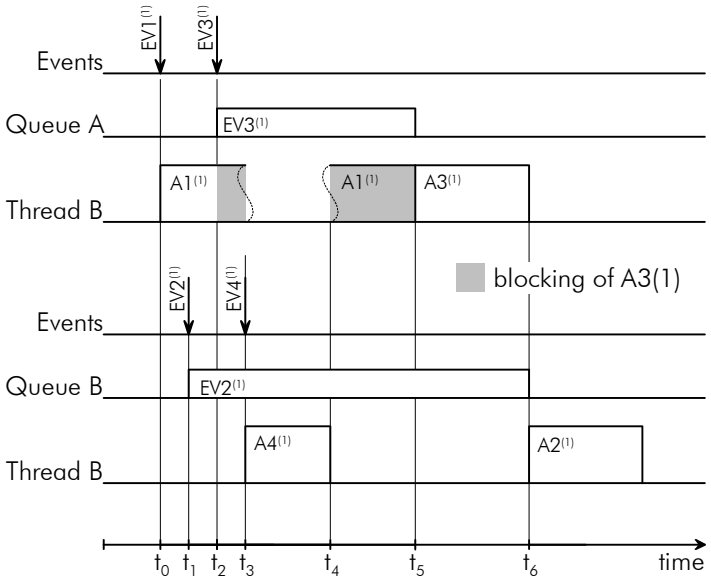

Abbildung 3.8: Zeitlicher Verlauf beim PT-Protokoll und Vermeidung der Verkettung von Blockierzeiten

Der Nachteil beim *PT*-Protokoll liegt in der Vergrößerung der Jitter-Effekte[6] bei der Bearbeitung von Ereignissen mit höherer Priorität gegenüber dem *BPI*-Protokoll.

3.3 Laufzeitsystem

In den vorhergehenden Abschnitten wurde zwar die neue Software-Architektur beschrieben, nicht aber auf die notwendige Unterstützung durch ein unterlagertes Laufzeitsystem eingegangen. Dies soll im folgenden Abschnitt nachgeholt werden.

Grundsätzlich ist für die Realisierung der vorgestellten Software-Architektur die Bereitstellung eines Realzeitbetriebsystems (RTOS) mit den folgenden Grundfunktionalitäten notwendig:

- Preemptives Multi-Tasking
- Prioritätsbasiertes Scheduling mit FIFO Verhalten bei Threads mit gleichen Prioritäten
- Möglichkeit der Veränderung von Prioritäten bei Threads während der Laufzeit
- Möglichkeit der Interrupt-Anbindung in Form von Interrupt Service Routinen
- Versenden von Messages mit Prioritäten
- Messagequeues mit Berücksichtigung der Message-Prioritäten beim Sortieren
- ❖ Prioritätsvererbungs-Protokolle zwischen den Messages in den Queues und den bearbeitenden Threads

Bis auf den letzten Punkt sind die geforderten Funktionen in vielen am Markt erhältlichen Realzeitbetriebsystemen enthalten. Besonders durch die Standardisierung von Programmierungs-Schnittstellen (Application Programming Interface, API) in den letzten Jahren hat eine zunehmende Vereinheitlichung des Funktionsumfangs stattgefunden. Erwähnenswert ist in diesem Kontext die Verbreitung des POSIX 4 Standards [123][124], der bereits von vielen Realzeitbetriebsystemen unterstützt wird.

Der POSIX 4 Standard (oder POSIX 1003.1) beinhaltet bereits die für diese Software-Architektur essenzielle Funktionalität von priorisierten Messages und Messagequeues. Dies ist der Hauptgrund, warum bei der letztendlichen Umsetzung des Frames (siehe Kapitel 6) ein Realzeitbetriebsystem mit einer POSIX 4 API Unterstützung gewählt wurde. Dadurch mussten lediglich die Prioritätsvererbungs-Protokolle als Erweiterung des Laufzeitsystems implementiert werden. Die Notwendigkeit der Erweiterung des Laufzeitsystems erforderte auch eine genaue Analyse der Interna des Betriebsystems selbst und damit eine Eingrenzung bei der Wahl auf reine Open-Source Systeme. Mehr über die Umsetzung ist in Kapitel 6.1 zu finden.

[6] Beim BPI Protokoll ist die Auftrittswahrscheinlichkeit von Blockierungen höher-priorer Aktionen durch nieder-priorer Aktionen die eine aktuelle höhere Priorität durch Vererbung besitzen geringer als beim PT Protokoll, wo die höhere Priorität IMMER automatisch beim Starten der Aktion vererbt wird. Die Jitter-Frequenz von höher-prioren Aktionen wird dadurch häufiger.

4 Realzeitnachweis

In diesem Kapitel wird der Realzeitnachweis für die entwickelte Software-Architektur präsentiert. Ausgehend von der Erklärung des verwendeten Taskmodells mit der statischen und dynamischen Sicht werden anschließend die verwendeten Methoden und Formeln vorgestellt und erklärt. Zum Schluss wird die Vorgehensweise bei einem gegebenen graphischen Modell beschrieben.

4.1 Taskmodell

Das hier vorgestellte Taskmodell beschreibt die präsentierte Software-Architektur in abstrakter Weise, um eine weitgehend modellunabhängige Methode der Realzeitanalyse zu ermöglichen. Außerdem erlauben die nachfolgenden mathematischen Formulierungen eine eindeutige Beschreibung, der für die Analyse notwendigen mathematischen Zusammenhänge.

Wichtig ist darauf hinzuweisen, dass die hier vorgestellte Realzeitanalyse unabhängig von der speziellen Ausprägung eines graphischen Modells ist. Sie ist somit ohne Einschränkung der Allgemeinheit für jegliche im Zusammenhang mit dieser Arbeit beschriebenen hybriden graphischen Spezifikation, die auf die vorgestellte SW-Architektur abgebildet wurde, anwendbar. Für die Analyse variieren lediglich die jeweiligen modellabhängigen Parameter.

4.1.1 Messagequeue und Thread

Das Grundkonstrukt der beschriebenen Software-Architektur ist ein Thread, der die Berechnungswünsche aus der ihm zugeordneten Messagequeue bearbeitet. Die Menge aller physikalischen Threads im Laufzeitsystem ist definiert als $\Gamma = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_t\}$. Die Gesamtanzahl t ergibt sich dabei aus der Summe der unterschiedlichen Abtastraten im zeitgesteuerten Teil und der Anzahl explizit definierter Threads im ereignisgesteuerten Teil eines graphischen Modells.

Zum dynamischen Verhalten der SW-Architektur bzw. des Taskmodells gehört das Vorgehen beim Einordnen einer neuen Message in eine Queue. Hierbei werden die Prioritäten der Ereignisse zum Sortieren verwendet. Das Ereignis in der Warteschlange mit der höchsten Priorität wartet an erster Stelle auf die Bearbeitung durch den Server-Thread. Messages mit derselben Priorität werden nach dem FIFO Prinzip eingeordnet. Durch die Sortierung der Ereignisse bzw. der Berechnungswünsche in den Warteschlangen wird eine Priorisierung der Aktivitäten im System realisiert.

4.1.2 Messages und Aktionen

Messages bzw. Ereignisse beschreiben jegliche Art von Berechnungswunsch im System. Jedes Ereignis hat dabei genau eine Aktion zur Folge, die der jeweiligen Reaktion des zugeordneten Threads entspricht. Durch die Durchführung der gewünschten Berechnungen wird das Ereignis bei der Bearbeitung „verbraucht". Die Bearbeitung selbst folgt dem *run-to-complition* Paradigma und kann nicht durch ein anderes Ereignis derselben Messagequeue verdrängt bzw. unterbrochen werden.

Die Menge aller Ereignisse im System ist definiert als $\varepsilon = \{EV_1, \ldots, EV_m\}$, wobei zwischen externen Ereignissen $\{EV_1, \ldots, EV_n\}$ und internen Ereignissen $\{EV_{n+1}, \ldots, EV_m\}$ unterschieden wird.

Externe Ereignisse entstehen zum Beispiel durch Interrupts von Peripheriehardware, wogegen interne Ereignisse als Folge von Aktionen im System „propagiert" werden.

Die Menge aller Aktionen im System ist definiert als $\alpha = \{A_1, ..., A_m\}$, wobei definitionsgemäß Aktion A_i das Ereignis EV_i bearbeitet.

Da in der vorgestellten Software-Architektur alle Berechnungswünsche durch Ereignisse ausgedrückt werden, besitzen im Laufzeitsystem die Ereignisse und nicht die Threads feste Prioritäten. Die Priorität eines Ereignisses ist definiert als $\pi(E_i)$ und wird zur Laufzeit dem bearbeitenden Thread vererbt, somit erhält auch die zugeordnete Aktion dieselbe Priorität wie das Ereignis. Die Priorität einer Aktion ist definiert als $\pi(A_i)$. Es gilt $\pi(E_i) = \pi(A_i)$.

Für die Berechnung der worst-case Reaktionszeit der Software-Implementierung eines graphischen Modells werden unter anderem die einzelnen worst-case Ausführungszeiten aller im System auftretenden Aktionen verwendet. Die worst-case Berechnungszeit einer Aktion ist definiert als $C(A_i)$.

Die sich so ergebenden Parameter sind bei einem Ereignis E_i die Priorität $\pi(E_i)$ und bei der zugeordneten Aktion A_i die Priorität $\pi(A_i)$, die worst-case Ausführungszeit $C(A_i)$ und der entsprechende bearbeitende Thread der Aktion $\Gamma(A_i)$.

Die Verwaltung von parallel ausführbaren Threads wird durch einen prioritätsbasierten Scheduler im Laufzeitsystem realisiert. Die Priorität des Threads wird dabei in erster Linie durch die augenblicklich bearbeitete Message und ihre Priorität bestimmt. Das zu bearbeitende Ereignis vererbt seine Priorität an den Serverthread und der prioritätsbasierte Scheduler entscheidet dann, welcher der lauffähigen Serverthreads als erstes aktiv wird. Somit wird immer gewährleistet, dass das Ereignis bzw. die Aktion mit der höchsten Priorität zum Zuge kommt.

Durch das *run-to-completion* Paradigma der Aktionen im System können jedoch aufgrund von Verdrängungen Prioritätsinversions-Situationen entstehen (siehe Abbildung 3.4), die eine unvorhersagbare Blockierung höher-priorer Ereignisse in Warteschlangen durch nieder-priore Aktionen in den Server-Threads zur Folge haben.

Um dies zu vermeiden, muss, neben der Vererbung der Priorität beim Start einer Aktion, auch eine Möglichkeit geschaffen werden, die momentane Server-Thread- bzw. Aktionspriorität an die jeweilig sich ändernde maximale Priorität der in der Warteschlange wartenden anderen Ereignisse anzupassen.

Hierzu wurden zwei unterschiedliche Ansätze betrachtet, das *Basic Priority Inheritance* Protokoll und das *Preemption Threshold* Protokoll. Auf die Unterschiede bei der Anwendung der vorgestellten Verfahren wird in den jeweiligen Abschnitten im späteren Verlauf dieses Kapitel näher eingegangen.

4.1.3 Aktions-Präzedenz-Graph (APG)

Wie bereits erwähnt, können Aktionen entweder durch externe oder durch interne Ereignisse getriggert werden. Die Kommunikationsbeziehung zwischen zwei Aktionen kann wie folgt beschrieben werden: sendet eine Aktion A_i ein Ereignis an Aktion A_j so gilt $A_i \rightarrow A_j$, wobei A_j der Nachfolger von A_i ist.

Das Versenden eines internen Ereignisses während der Bearbeitung einer Aktion entspricht dem asynchronen Triggern eines anderen Berechnungswunsches. Hierbei gilt als Einschränkung bei der hier vorgestellten SW-Architektur, dass die Prioritäten aller gesendeten Ereignisse nicht größer sein darf als die statische Priorität der Sende-Aktion selbst. Die statische Priorität einer Nachfolge-Aktion kann somit maximal gleich hoch sein, wie die Priorität der Vorgänger-Aktion bzw. des Vorgänger-Events.

In mathematischer Form bedeutet dies: $\pi(A_j) \leq \pi(A_i) \quad \forall \quad A_i \rightarrow A_j$.

Durch diese Einschränkung wird die letztendliche Umsetzung der beschriebenen asynchronen Triggerung zwischen Aktionen vereinfacht. Im Laufzeitsystem reicht ein einfaches prioritätsbasiertes Scheduling-Verfahren aus, das im Falle von gleichen Prioritäten ein FIFO Verhalten besitzt, um beim Versenden eines Folge-Ereignisses eine direkte Unterbrechung der eigenen Aktivitäten zu vermeiden.

Auf der anderen Seite führt diese Bedingung aber zu keiner gravierenden Einschränkung des möglichen Lösungsraumes (Designfreiheiten), da in einem Realzeitsystem mit gewünschten und garantierten Antwortzeiten kaum eine nieder-priore Aktion ein Ereignis mit höherer Priorität versenden sollte. Dabei könnte nämlich nur in bestimmten Fällen die Reaktionszeit der höher-prioren Aktion ermittelt werden.

Durch die beschriebene Kommunikationsbeziehung zwischen Aktionen entstehen somit Aktionsketten (siehe Abbildung 4.1), die immer von einer externen Ereignis-Quelle starten und sich je nach umgesetzten graphischen Modell über mehrere Aktionsebenen erstrecken. Diese Aktionsketten beschreiben die Beziehungen zwischen allen Aktionen im System und werden auch als *Aktions-Präzedenz-Graphen (APG)* oder *Transaktions-Graphen (TG)* τ bezeichnet. Es existieren genau so viele unterschiedliche APG im System wie unterschiedliche externe Ereignis-Quellen.

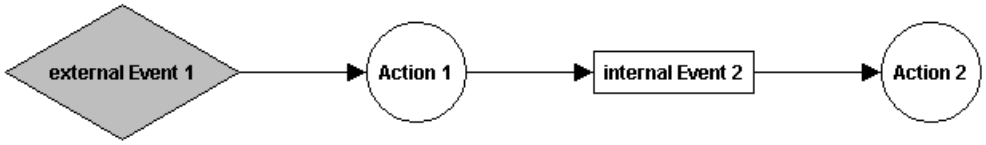

Abbildung 4.1: Beispiel eines einfachen Aktions-Präzedenz-Graphen

Zur Durchführung des hier beschriebenen Realzeitnachweises werden zwei unterschiedliche Abstraktionsebenen von *Aktions-Präzedenz-Graphen* eingeführt.

Allgemeiner Aktions-Präzedenz-Graph (Level 2)

Im *allgemeinen Aktions-Präzedenz-Graphen (Level 2)* werden *alle* möglichen Kommunikationsbeziehungen im System dargestellt. Dabei werden beim Versenden interner Ereignisse durch Aktionen zwei grundsätzliche Möglichkeiten der Handhabung unterschieden.

Fordert die Implementierung einer bestimmten Aktion *immer* das Propagieren *aller* ausgehenden Ereignisse, verhalten sich alle Nachfolgepfade im *allgemeinen APG* als UND Pfade. In diesem Fall müssen somit immer *alle* Nachfolgepfade mit in die Realzeitanalyse aufgenommen werden.

Triggert eine Aktion jedoch nur bestimmte, sich ausschließende Pfade, handelt es sich um sogenannte ODER Pfade. In diesem Fall muss immer nur einer der exklusiven ODER Pfade bei der Realzeitanalyse berücksichtigt werden. Solche ODER Pfade ermöglichen eine gezieltere Untersuchung des Systems, da konkretes Wissen über das Verhalten der Applikation mit in die Analyse aufgenommen werden kann.

Graphisch lässt sich diese zusätzliche Information in Form von Bedingungsknoten, wie in Abbildung 4.2 dargestellt, beschreiben.

Gehen einmal mehrere Nachfolgeereignisse aus einer Aktion hervor, ohne durch einen Bedingungsknoten zu laufen (z.B. Abbildung 4.2, *EV3*), so entspricht dies einer impliziten UND Bedingung für alle Nachfolgepfade.

Die zusätzlichen Parameter in Abbildung 4.2 sind die den Ereignissen zugeordneten statischen Prioritäten *prio* (größere Zahl entspricht höhere Priorität) und die jeweilige worst-case Ausfüh-

rungszeit C der Aktionen. Die Abbildung von Hardware-Interrupt-Prioritäten auf Software-Ereignis-Prioritäten wird in Kapitel 4.1.3 beschrieben.

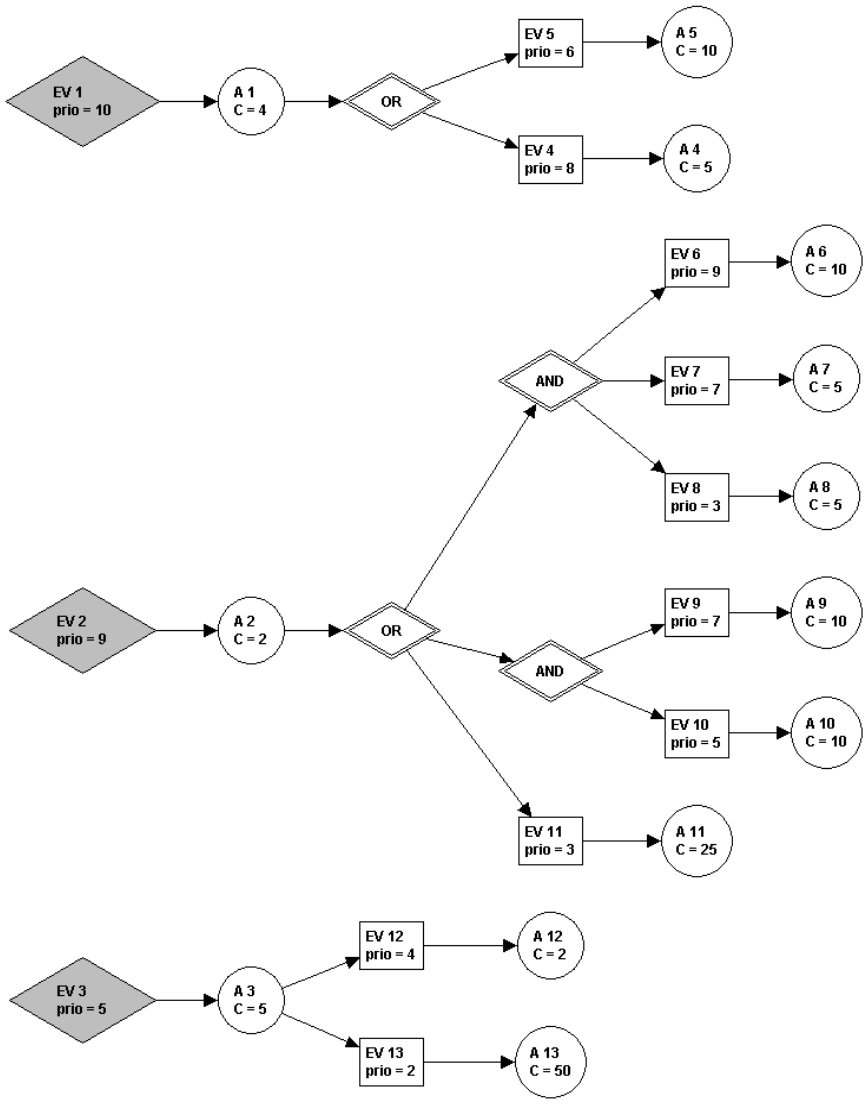

Abbildung 4.2: Beispiel *allgemeiner APG (Level 2)* mit OR und UND Bedingungsknoten

Konkreter Aktions-Präzedenz-Graph (Level 1)

Vor jedem Schritt der nachfolgend vorgestellten Analysemethode muss aus dem *allgemeinen APG* eine konkrete Instanz abgeleitet werden, da zur Vereinfachung der mathematischen Analyse ausschließlich UND Pfade in den entgültig betrachteten Aktions-Präzedenz-Graphen erlaubt sind.

Die dazu notwendigen Strukturentscheidungen bei den entsprechenden ODER Bedingungsknoten werden durch

- die Zusammensetzung des *allgemeinen APG*,
- die aktuelle Priorität der zu untersuchenden Aktion und
- der jeweilig betrachteten Phase des Nachweises (siehe Abschnitt 4.2)

beeinflusst. Die resultierende konkrete Instanz des *allgemeine APG* wird nachfolgend auch als der *konkreter APG (Level 1)* bezeichnet.

Die Beschränkung auf UND Pfade bei der mathematischen Analyse hat den Vorteil, dass beim Eintreten eines bestimmten externen Ereignisses, ausgehend von der Wurzel des entsprechenden *konkreten APG,* alle darin enthaltenen Aktionen getriggert werden. Der *konkrete APG* spiegelt somit die gesamte Reaktion des Systems auf das externe Ereignis für einen bestimmten Fall wieder, wo hingegen der *allgemeine APG* verschiedene Konstellationen mit exklusiven Pfadknoten wiedergibt.

Aus dieser Erkenntnis leitet sich auch die Transformationsbedingung ab, die einen *Level 2 APG* auf einen *Level 1 APG* abbildet. Es wird bei jedem ODER Bedingungsknoten jeweils der exklusive Pfad gewählt, der unter den betrachteten Umständen die größte kumulierte Ausführungszeitanforderung wiederspiegelt. Die betrachteten Umstände sind, wie erwähnt, von der Analysephase des Realzeitnachweises (Blockierzeiten, Startzeitpunkte oder Endzeitpunkte) und dem verwendeten Prioritätsvererbungs-Protokoll abhängig.

Zur Veranschaulichung der Vorgehensweise bei der Transformation werden nachfolgend aus den in Abbildung 4.2 dargestellten *allgemeinen APG*, die jeweiligen *konkreten APG* zur Untersuchung von drei unterschiedlichen Aktionen ($A13$, $A4$ und $A5$) abgeleitet.

Es wird hier zur Vereinfachung angenommen, dass jede Aktion auf einen eigenen Thread abgebildet wird und die gewonnenen *konkreten APG* zur Bestimmung des worst-case Startzeitpunktes der untersuchten Aktion verwendet werden.

Dadurch ergibt sich die aktuelle Priorität der Aktion lediglich aus der statisch definierten Priorität des Trigger-Ereignisses und ist unabhängig vom gewählten Prioritätsvererbungs-Protokoll des Laufzeitsystems.

Im Falle einer komplexeren SW-Architektur muss jeweils die niedrigste Priorität einer untersuchten Aktion zur Bestimmung der jeweiligen kumulierten Rechenzeitanforderung bei den unterschiedlichen Pfaden einer ODER Verbindung angenommen werden (siehe Berechnung der kumulativen Rechenzeitanforderung in Abschnitt 4.2.3, Gleichung 4.15 und 4.17). Je nach Analysephase wirkt sich auch das Prioritätsvererbungs-Protokoll auf die genannte niedrigste Priorität aus.

Untersuchung von Aktion $A13$ mit der statischen Priorität 2

Bei der Untersuchung des worst-case Startzeitpunktes von Aktion $A13$ ergibt sich aus den *allgemeinen APG*, die in Abbildung 4.3 dargestellten *konkreten APG*. Hierzu wird für jeden ODER Pfad jeweils die kumulierte Rechenzeitanforderung, die von der augenblicklich betrachteten Priorität (2) abhängt, ermittelt und anschließend der Pfad mit dem Maximum in den konkreten APG aufgenommen.

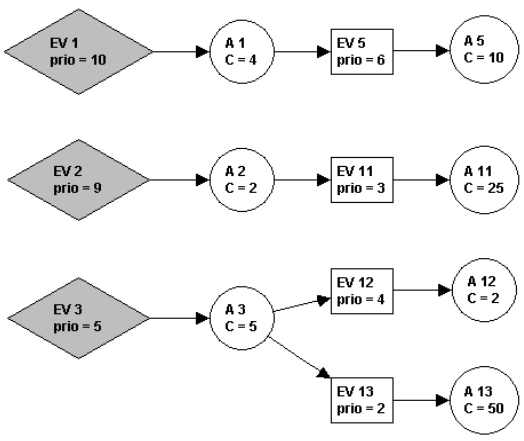

Abbildung 4.3: *Konkrete APG* bei einer untersuchten Priorität von 2

Beim *allgemeinen APG* mit dem Start-Event $EV2$ ergeben sich drei mögliche ODER Pfade. Diese Pfade besitzen bei der untersuchten Priorität (2) jeweils die kumulierten Rechenzeitanforderungen von: 20 ($A6 + A7 + A8$), 20 ($A9 + A10$) und 25 ($A11$) Zeiteinheiten. Der Maximum-Pfad ist somit der mit Aktion $A11$.

Untersuchung von Aktion $A12$ mit der statischen Priorität 4

Die in Abbildung 4.4 dargestellten *konkreten APG* ergeben sich aus den *allgemeinen APG* bei der Untersuchung des worst-case Startzeitpunktes von Aktion $A12$.

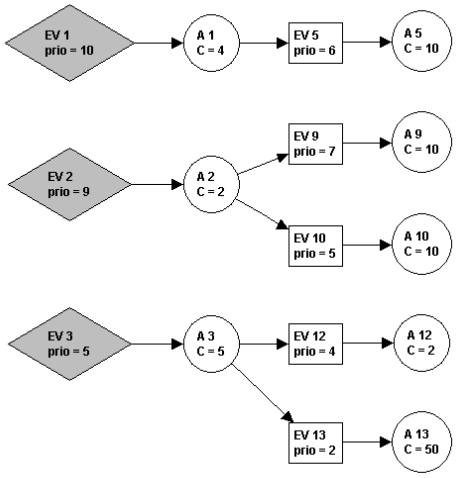

Abbildung 4.4: *Konkrete APG* bei einer untersuchten Priorität von 4

Beim *allgemeinen APG* mit dem Start-Event $EV2$ ergeben sich bei der untersuchten Priorität (4) jeweils die kumulierten Rechenzeitanforderungen von: 15 ($A6 + A7$), 20 ($A9 + A10$) und 0 Zeiteinheiten. Der Maximum-Pfad ist somit der mit den Aktionen $A9$ und $A10$.

Untersuchung von Aktion $A5$ mit der statischen Priorität 6

Die in Abbildung 4.5 dargestellten *konkreten APG* ergeben sich aus den *allgemeinen APG* bei der Untersuchung des worst-case Startzeitpunktes von Aktion $A5$.

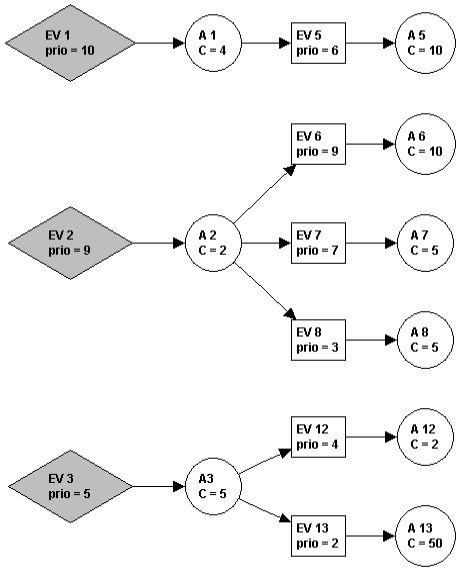

Abbildung 4.5: *Konkrete APG* bei einer Priorität von 6

Beim *allgemeinen APG* mit dem Start-Event $EV2$ ergeben sich bei der untersuchten Priorität (6) jeweils die kumulierten Rechenzeitanforderungen von: 15 ($A6 + A7$), 10 ($A9$) und 0 Zeiteinheiten. Der Maximum-Pfad ist somit der mit den Aktionen $A6$ und $A7$.

Um die Zugehörigkeit eines Ereignisses E_i oder einer Aktion A_i zu einem APG τ wieder zu geben, wird in den nachfolgenden mathematischen Beschreibungen folgende Nomenklatur verwendet: E_i^{τ} bzw. A_i^{τ}.

Nachdem die Grundformen eines Aktions-Präzedenz-Graphen erläutert wurden, soll auf die möglichen Ausprägungen bei der hier vorgestellten Software-Architektur eingegangen werden.

Ereignisgesteuerter Software-Anteile als APG

Der interruptgetriebene Modellanteil (siehe Abbildung 4.6) der hier beschriebenen SW-Architektur setzt sich im wesentlichen aus

- Interrupt Service Routinen (*ISR*) zur Einstreuung externer Ereignisse und aus
- Messagequeues mit den dazugehörigen Server-Threads, zur Bearbeitung der Aktionen

zusammen.

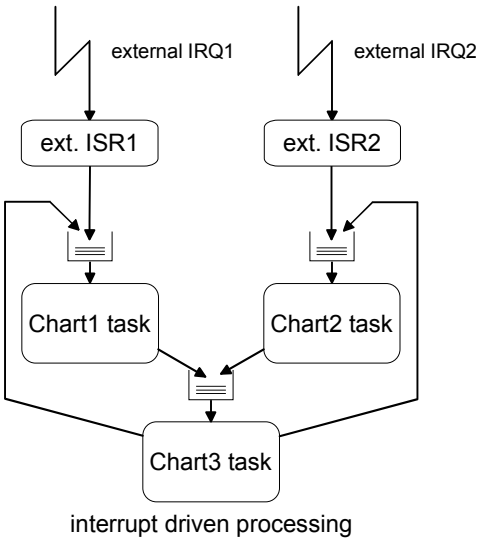

Abbildung 4.6: Beispiel eines ereignisgesteuerten Anteils der vorgestellten Software-Architektur

Die konkrete Struktur der Software-Architektur für ein gegebenes Modell wird durch das Design des Entwicklers definiert. Der Entwickler beschreibt durch die graphische, ausführbare Spezifikation die Funktionen der verschiedenen Modellteile, die dann auf unterschiedliche Threads eines RTOS abgebildet werden sollen. Durch die Modellstruktur wird auch die Anzahl der Messagequeues bzw. der physikalischen Threads des ereignisgesteuerten Anteils definiert. Die Kommunikationsbeziehungen zwischen den Threads bzw. Aktionen werden ebenfalls auf graphischer Ebene durch das Versenden und Empfangen von Ereignissen definiert und festgelegt.

Aus dieser, in einem konkreten Modell enthaltene Kommunikationsbeziehung und dem Applikationswissen, über welche Aktion welches Folge-Ereignis unter welchen Umständen versendet, kann der für die Realzeitanalyse notwendige *allgemeine Aktions-Präzedenz-Graph* abgeleitet werden.

Ermittlung des allgemeinen Aktions-Präzedenz-Graphen

Jeder ereignisgesteuerte *Aktions-Präzedenz-Graph* startet mit einer externen Ereignis-Quelle.

Solche Ereignis-Quellen können entweder:

- Interrupt Requests der Hardware (*IRQ*),
- oder bedingt ausgeführte Systemteile des zyklischen Modells (z.B. Ausnahmebehandlung bei einer Grenzwertüberschreitung),

sein.

Die erste Aktion eines solchen ereignisgesteuerten APG beschreibt die Reaktion des Systems auf ein Ereignis aus der zugeordneten Quelle.

Im Falle eines Hardware-Interrupt-Requests ist die zugeordnete Interrupt-Service-Routine (ISR) die erste Aktion des *APG*. Für eine einheitliche Realzeitanalyse und der Berücksichtigung von Rechenzeitanforderungen durch die IRQs bzw. deren ISRs ist eine eindeutige Abbildung der HW-Prioritäten externer Interrupt-Quellen auf die internen Prioritäten von Aktionen im SW-System unerlässlich. Da die HW-Prioritäten grundsätzlich über den im Laufzeitsystem definierten SW-Prioritäten liegen, erfolgt die Abbildung durch eine geeignete Verschiebung alle HW-Prioritäten über allen im System vorkommenden SW-Aktions-Prioritäten. Dadurch wird das reale Systemverhalten der Aktionen (Mischung zwischen ISRs und RTOS-Threads) bei der Analyse wiedergegeben und auch die Beeinflussung unterschiedlicher HW-Prioritäten[7] untereinander einfach im Nachweisverfahren berücksichtigt.

Durch die Analyse der Kommunikationsverbindungen der ersten Aktion des APG kann dann sukzessive der *gesamte APG* der betrachteten Ereignis-Quelle gebildet werden.

Dabei wird für jede betrachtete Aktion (ausgehend von der ersten) bestimmt, ob Nachfolge-Ereignisse bei der Ausführung versendet werden können. Ist dies der Fall, werden je nach Funktionsspezifikation der betrachteten Sende-Aktion, im *allgemeinen APG,* wenn nötig ODER bzw. UND Bedingungsknoten eingefügt, bevor das bzw. die gesendeten Ereignisse und ihre resultierenden Aktionen (Empfangs-Aktionen) mit in den Graphen aufgenommen werden.

In diesem Schritt wird das applikationsspezifische Wissen, das in der Sende-Aktion enthalten ist, mit in die Analyse integriert. Der Vorteil ist eine genauere Beschreibung des Systemverhaltens und eine weniger pessimistische Untersuchung der umgesetzten graphischen Spezifikation.

Die ermittelten Empfangs-Aktionen bearbeiten wiederum die empfangenen Berechnungswünsche entsprechend der für sie definierten graphischen Spezifikation ab und können wiederum Nachfolgeereignisse versenden. So werden sukzessiv alle Aktionen und Kommunikationsverbindungen des Systems abgearbeitet, bis das Ende aller Pfade erreicht wird und somit keine Nachfolgerereignisse mehr vorkommen.

Einige elementare Grundkonstrukte, die sich beim ereignisgesteuerten Modellteil ergeben können, sollen nachfolgend zur näheren Erläuterung vorgestellte werden. Hierzu wird jeweils eine konkrete Ausprägung der SW-Architektur und die, bei entsprechendem Applikationsverhalten resultierenden *allgemeinen APG* und *konkreten APG,* vorgestellt.

---

[7] Für den Fall, dass die HW-Prioritäten eine von den SW-Prioritäten divergierende Sortier-Reihenfolge besitzen, kommt zur Verschiebung auch ein Vorzeichenwechsel hinzu.
Beispiel: HW-Level 1 ≡ höchste Priorität, HW-Level 40 ≡ niederste Priorität; SW-Level 1 ≡ niederste Priorität, SW-Level 254 ≡ höchste Priorität. Lösung: neuesHW-Level = (300 – altesHW-Level)

Einfache Verkettung

Bei einer einfachen Verkettung wird durch das fortlaufende Versenden von internen Ereignissen eine einfache Kette von Aktionen gebildet. Aus der dargestellten SW-Architektur in Abbildung 4.7 kann die Anzahl der unterschiedlichen internen (*EV1* und *EV2*) und externen (*IRQ*) Ereignistypen entnommen werden. Dadurch wird auch die Anzahl unterschiedlicher möglicher Aktionen im APG definiert.

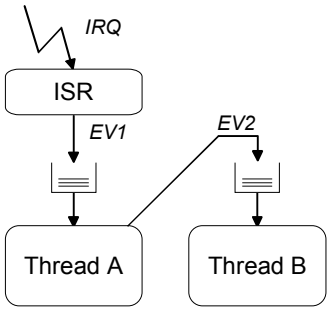

Abbildung 4.7: Konkrete SW-Architektur für eine einfache Verkettung

Für den einfachsten Fall, dass die Aktion *A1* im *Thread A* als Reaktion auf *EV1* immer *EV2* verschickt, ist der *allgemeine APG* und der *konkreten APG* identisch (siehe Abbildung 4.8). Es ist nämlich im *allgemeinen APG* kein ODER Bedingungsknoten enthalten, der transformiert werden müsste.

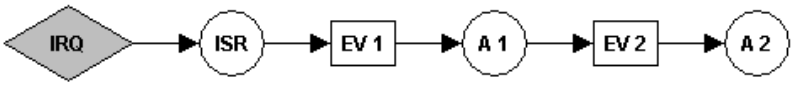

Abbildung 4.8: Allgemeiner und konkreter APG einer einfachen Verkettung

Verzweigung

Bei einer Verzweigung ist die grundlegende SW-Architektur (siehe Abbildung 4.9) unabhängig vom Applikationsverhalten. Es können sich jedoch durch die spezifizierte Funktion unterschiedliche APG ergeben.

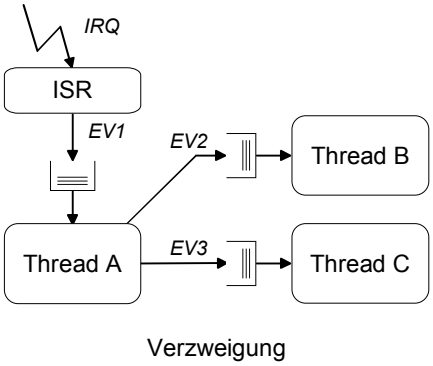

Abbildung 4.9: Konkrete SW-Architektur für eine Verzweigung

Nehmen wir an, dass Aktion *A1* im *Thread A*, je nach internem Zustand (z.B. Vorgeschichte), entweder *EV2* ODER *EV3* verschickt. Dieses Applikationswissen würde dazu führen, dass der *allgemeine APG* nach Aktion *A1* einen ODER Bedingungsknoten erhält und wie in Abbildung 4.10 dargestellt, aufgebaut wäre.

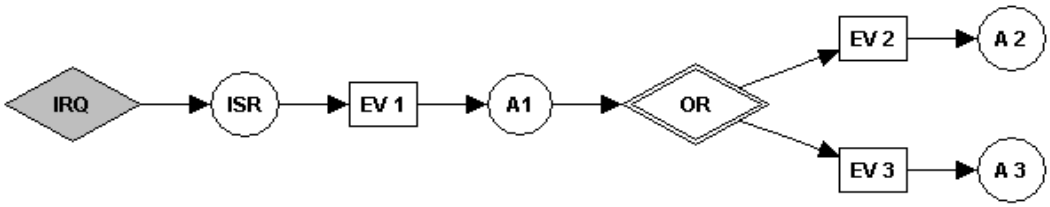

Abbildung 4.10: *Allgemeiner APG* bei einer ODER-Bedingung zwischen $EV2$ und $EV3$

Die möglichen zwei *konkreten APG* sind dann entweder Abbildung 4.11 oder Abbildung 4.12.

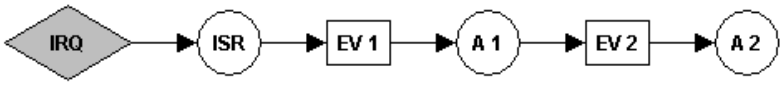

Abbildung 4.11: Transformation vom *allgemeinen APG* in *konkreten APG* (Fall1)

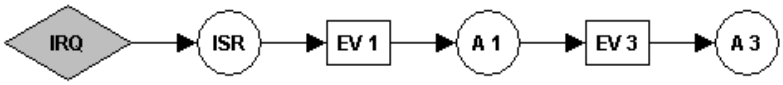

Abbildung 4.12: Transformation vom *allgemeinen APG* in *konkreten APG* (Fall2)

Für den Fall, dass Aktion $A1$ immer beide Nachfolge-Events verschickt, würde der neue *allgemeine APG* der Überlagerung der zuvor erhaltenen einzigen *konkreten APG* entsprechen und anstelle der vorherigen ODER Bedingung eine implizite UND-Bedingung beinhalten (siehe Abbildung 4.13). Hier entsprechen sich *allgemeiner* und *konkreter APG* wieder.

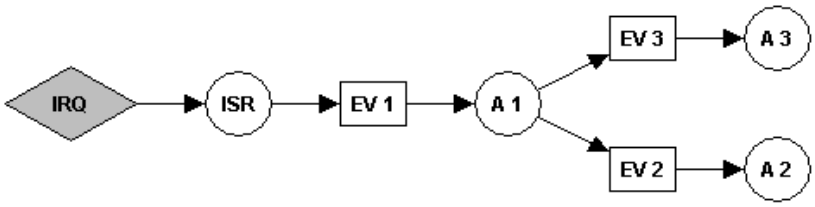

Abbildung 4.13: *APG* bei einer UND-Bedingung zwischen $EV2$ und $EV3$

Vereinigung

Bei einer simplen Vereinigung nutzen zwei *APG* ein und denselben Thread für zwei unterschiedliche Aktionen. In der hier dargestellten SW-Architektur ist dies durch den Empfang zweier unterschiedlicher Ereignistypen ($EV3$ und $EV4$) durch ein und denselben Thread dargestellt.

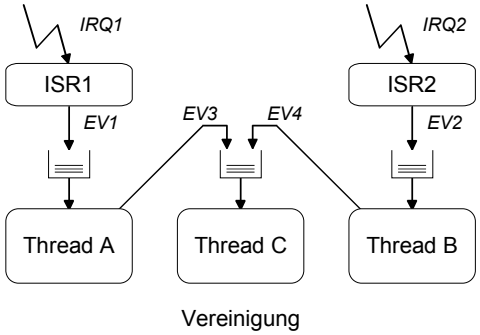

Abbildung 4.14: Konkrete SW-Architektur für eine Vereinigung

Die zwei *APG* sind im einfachsten Fall zwei einfache Verkettungen deren letzten Aktionen *A3* und *A4* durch *Thread C* bearbeitet werden. Die eigentliche Beeinflussung untereinander (z.B. mögliche gegenseitige Blockierung durch „run-to-complition" Paradigma) ist nicht direkt in den *APG* ersichtlich, sondern wird erst bei der Analyse durch die geeignete mathematische Berechnung berücksichtigt. Ein Hinweis für eine mögliche Beeinflussung kann in Abbildung 4.15 die Angabe der physikalischen Threads der unterschiedlichen Aktionen sein.

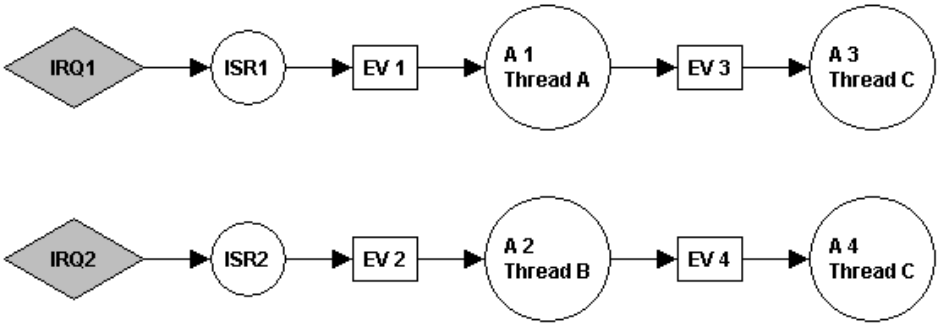

Abbildung 4.15: *APG* für das einfachste Beispiel einer Vereinigung

Schleife (Loop)

Als letzte Variante der Grundstrukturen ereignisgesteuerter Modellanteile soll nachfolgend die Schleife beschrieben werden. Grundlegend für die Behandlung von rekursiven Pfaden in jeder Art von realzeitfähiger Software, ist die Angabe einer oberen, maximalen Anzahl von möglichen Durchläufen der Schleife. Durch die Angabe einer solchen Obergrenze kann jede Schleife in eine einfache Verkettung transformiert werden, die für die Analyse bedeutend einfacher zu handhaben ist.

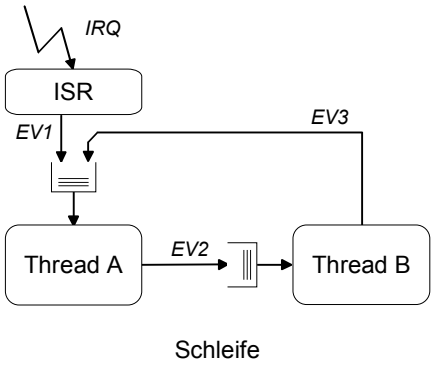

Abbildung 4.16: Konkrete SW-Architektur für eine Schleife

Bei der in Abbildung 4.16 dargestellten SW-Architektur liegt eine Schleife vor (siehe Abbildung 4.17). Sowohl Aktion *A1* als auch Aktion *A3* können das Nachfolge-Ereignis *EV2* verschicken und zudem wird Aktion *A3* durch Aktion *A2* getriggert.

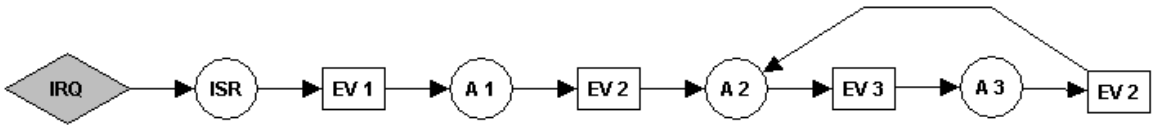

Abbildung 4.17: *Allgemeiner APG* für eine Schleife

Um diese Art von Schleifen in *allgemeinen APG* bei der nachfolgenden Analyse berücksichtigen zu können, werden die einzelnen Schleifendurchläufe in eine einfache Verkettung transformiert und dadurch der *konkrete APG* des Systems gewonnen.

Damit auch die Reaktionszeit gleicher Aktionen unterschiedlicher Instanzen einer Schleife untersuchen werden können, wird bei der Transformation den beteiligten Aktionen ein eindeutiger Name gegeben. Das Ergebnis ist in Abbildung 4.18 für einen maximalen Schleifendurchlauf von zwei dargestellt.

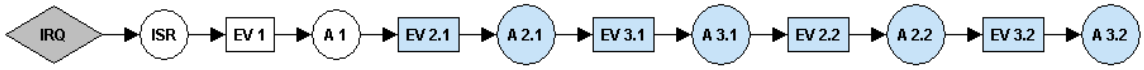

Abbildung 4.18: *Konkreter APG* für eine Schleife mit maximal zwei Durchläufen

Zyklische Software-Anteile als APG

Der zyklische Modellanteil (siehe Abbildung 4.19) setzt sich wie bereits in Kapitel 3.2.1 erläutert, im wesentlichen aus:

- einer Periodischen Interrupt Service Routine (ISR),
- einer Manager-Task (tBaseRate) mit dem Scheduling-Anteil und möglicherweise dem Modellanteil der höchsten Samplerate und
- beliebig vielen Tasks für die langsameren Sampleraten (tRate*n*)

zusammen.

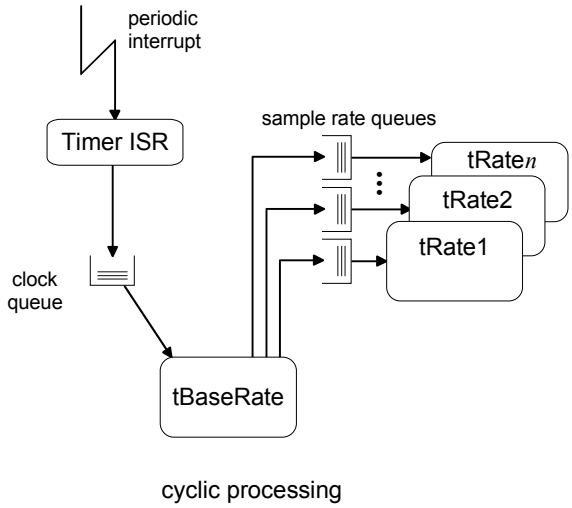

Abbildung 4.19: Zeitgesteuerter Anteil der vorgestellten Software-Architektur

Die Anzahl der effektiven Threads für den zeitgesteuerten Anteil einer graphischen Spezifikation ergibt sich letztendlich durch die Anzahl der Abtastraten im System und einer zusätzlichen Task für den Manager-Anteil, falls ein bestimmtes Abtastratenverhältnis vorliegt.

Da jeder in Abbildung 4.19 dargestellte Thread immer nur denselben Typ von Berechnungsanforderung empfängt, ist der sich daraus ergebende *allgemeine APG*, wie in Abbildung 4.20 dargestellt, relativ einfach.

Der Periodische Hardware Interrupt *PIT_IRQ* stößt zyklisch die Bearbeitung des zeitgesteuerten SW-Anteils an. Die zugeordnete *PIT_ISR* bearbeitet dieses einzige reale externe Ereignis des zyklischen Modellteiles und versendet ihrerseits ein Sample-Event *SampleEV* an die Manager-Task *tBaseRate*. Der dort enthaltene modellspezifische Scheduling-Anteil verteilt dann wieder-

um, dem Abtastraten-Teiler-Verhältnis entsprechend, alle anderen Sample-Trigger-Ereignisse an die anderen Tasks $tRate_n$.

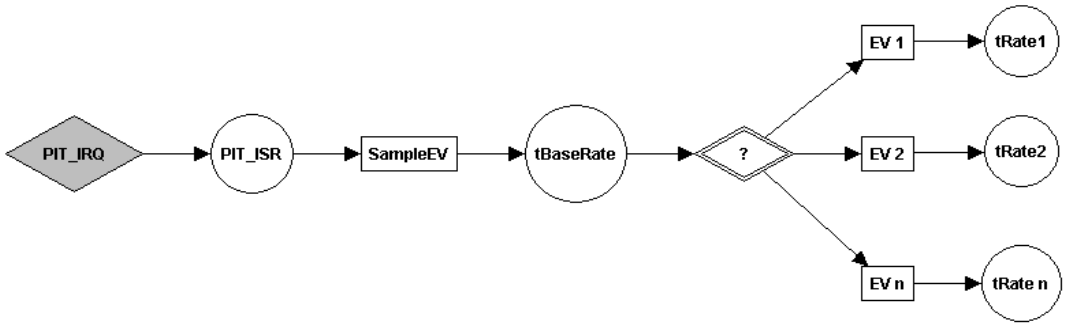

Abbildung 4.20: Vorläufiger *allgemeiner APG* für den zyklischen Modellanteil

Die Prioritäten der versendeten Ereignisse spiegeln dabei die Prioritätsvergabe bei einer Rate-Monotonic-Priorityassignement Strategie wieder. Es gilt:

$$\pi(HW\_IRQ_x) > \pi(SampleEV) > \pi(EV_1) > \pi(EV_2) > ... > \pi(EV_n).$$

Der in Abbildung 4.20 mit einem „?" gekennzeichnete Bedingungsknoten spiegelt das deterministische Verhalten des Sample-Trigger-Verteilens wieder. Bei einer worst-case Betrachtung während der Realzeitanalyse müsste der „?" Knoten als eine UND Bedingung gewertet werden, da in einem solchen Fall (z.B. Zeitpunkt $t = 0$) immer alle vorkommenden Berechnungswünsche aktiviert werden müssten. Diese Annahme gilt aber nicht immer und würde zu einer eindeutigen und unnötigen Überdimensionierung bzw. zu einer sehr pessimistischen Realzeitanalyse führen.

Dieser vorläufige *allgemeine APG* in Abbildung 4.20 kommt somit der gegebenen Software-Architektur zwar sehr nahe, würde bei einer Analyse des Systems aber nicht den unterschiedlichen Aktivierungszeitpunkten der verschiedenen Sampleraten Rechnung tragen, da die „?" Bedingung nicht einer wirklichen UND Verknüpfung entspricht.

Um dieses Modellierungs-Problem zu lösen, muss das deterministische Verhalten beim Triggern der Nachfolge-Aktionen entsprechend dem Sampleraten-Verhältnis berücksichtigt werden.

Dieses Applikationswissen über das deterministische Verhalten des „?" Bedingungsknoten kann einfach und ohne grundlegende Anpassungen des Realzeitnachweises berücksichtigen werden. Hierzu wird nachfolgend die Transformation des in der Abbildung 4.20 gegebenen vorläufigen *allgemeinen APG,* in einen für die Analyse günstigeren und weniger pessimistischen *konkreten APG* erläutert.

Transformation des zyklischen allgemeinen APG in konkrete APG

Die Grundidee der Transformation ist die Aufspaltung aller Sampleraten des Systems in einzelne separate Transaktionsgraphen (siehe Abbildung 4.21) mit jeweils geeigneten zeitlichen Anforderungsfunktionen (Abtastraten).

Dadurch wird eine gezieltere Berücksichtigung der geforderten Berechnungszeitanforderungen in der späteren Analyse möglich. Bei der Transformation müssen die durch die Implementierung der Software-Architektur gegebenen Abhängigkeiten beibehalten werden, um sie dann in der Analyse mit berücksichtigen zu können.

Ein zentraler Aspekt der vorgestellten Software-Architektur ist die besondere Stellung der Manager-Task *tBaseRate*. *tBaseRate* beinhaltet als einzige Task des zyklischen SW-Anteils, neben

dem Modell-Teil auch einen modellspezifischen Scheduling-Anteil, der die Verteilung aller anderen Sample-Trigger-Ereignisse übernimmt.

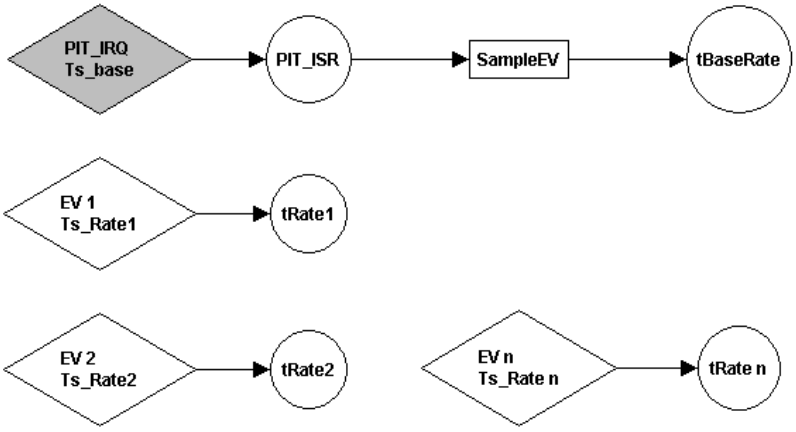

Abbildung 4.21: Aufteilung eines *APG* auf mehrere *abtastratenbezogene APG*

Dieser Software-Teil ist ebenfalls stark von dem bereits erwähnten Abtastraten-Verhältnis abhängig, da zu bestimmten Zeitpunkten nur bestimmte Ereignisse versendet werden. Die Ausführungszeit dieses Teiles soll deshalb durch die Transformation ebenfalls dem jeweiligen *konkreten Sampleraten-Aktions-Präzedenz-Graphen* zugeordnet werden.

Berücksichtigt man dieses Implementierungsdetail bei der Transformation, so ist es naheliegend die Aktion *tBaseRate* aus Abbildung 4.21 in geeignete, unabhängige Aktionen zu teilen. Hierzu wird zuerst der Scheduling-Anteil vom Modell-Anteil getrennt und anschließend der Scheduling-Anteil weiter aufgelöst, um die so erhaltenen neuen Aktionen auf die verschiedenen verursachenden Sampleraten-APG aufzuteilen.

Die sich daraus ergebenden endgültigen *konkreten APG* für den zyklischen Teil einer graphischen Spezifikation sind in Abbildung 4.22 dargestellten.

*Beschreibung des konkreten APG für tBaseRate (*Abbildung 4.22 *oben)*

Der *konkrete APG* der Grund-Abtastrate *tBaseRate* wird nach der Transformation durch die einzige, reale, externe Ereignis-Quelle des zyklischen Systems, dem periodischen Interrupt (PIT) *PIT_IRQ*, getriggert. Die Priorität des Hardware IRQs wird dabei, wie bereits beschrieben, auf die Priorität des externen Ereignisses *PIT_IRQ* abgebildet und zu *HW_ISRx* definiert.

Die erste Aktion *PIT_ISR* des Graphen entspricht der Interrupt Service Routine und spiegelt die damit angeforderte Rechenzeit *C_PIT_Handler* wieder. Sie versendet während der Verarbeitung immer das interne Ereignis *SampleEV*, das die Priorität *tBase_prio* besitzt. Daraufhin wird sowohl der Scheduling-Teil für *tBase* (Aktion *Scheduling0* mit der Ausführungszeit *C_tBaseSchedule*) als auch der Modellteil (Aktion *tBaseRate* mit *C_tBaseModel*) angestoßen.

Das interne Ereignis *EV_Ts_0* ist dabei kein reales im Software-System vorkommendes Ereignis, sondern dient bei der beschriebenen Transformation nur zur Aufteilung des Scheduling-Overheads der Manager-Task auf die unterschiedlichen APG der verschiedenen Sampleraten des Modells.

Beschreibung des konkrete APG für $tRate_x$ *(*Abbildung 4.22 *Rest)*

Die *konkreten APG* der anderen Abtastraten des Modells sind grundsätzlich immer gleich aufgebaut.

Die zuvor internen Ereignisse *EVx* (siehe Abbildung 4.20) werden durch die vollständige Transformation zu pseudo-externe Ereignisse, mit einer jeweils eigenen zeitlichen Anforderungsfunktion (siehe Gleichung 4.3 auf Seite 54), umgewandelt.

Die Prioritäten dieser Ereignisse *EVx* sind für alle Abtastraten identisch und entsprechen der Priorität der *tBaseRate* Aktion, *tBase_prio*.

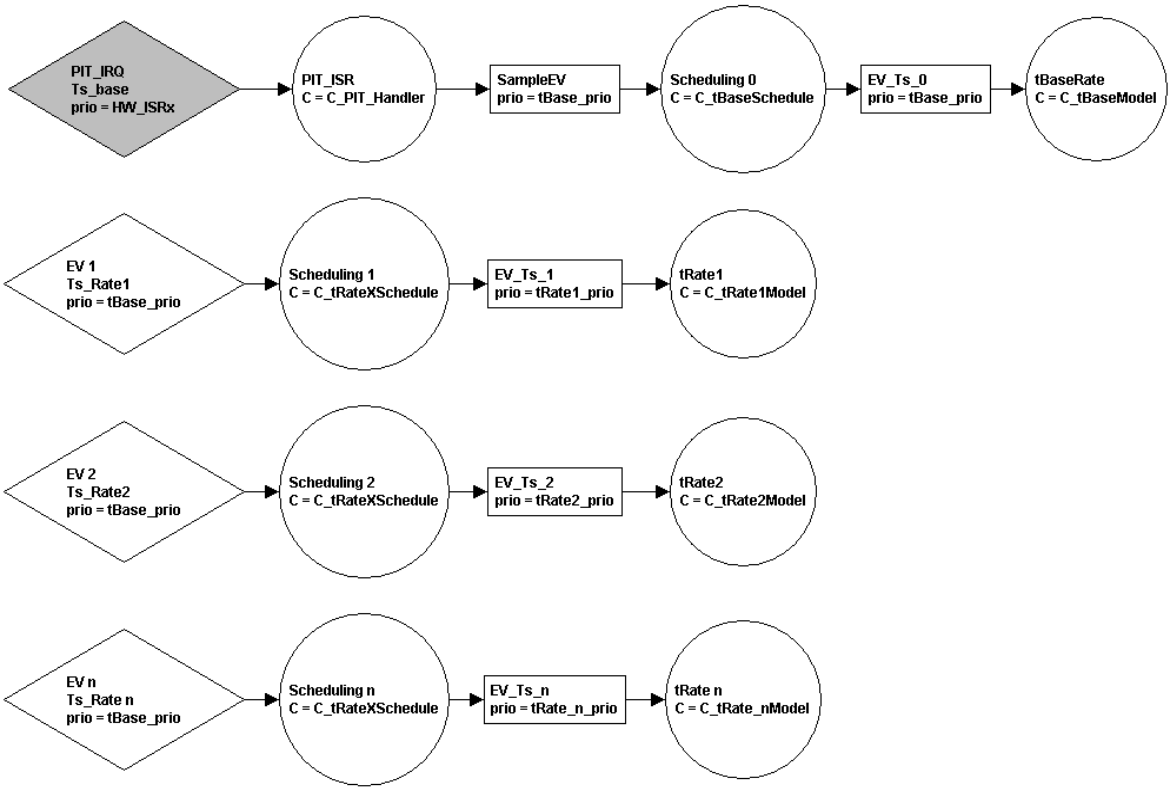

Abbildung 4.22: Letztendliche *APG* des zyklischen Modellanteiles

Diese Wahl der Prioritäten ergibt sich dadurch, dass die jeweils erste Aktion *SchedulingX* der neuen, transformierten *APG*, den Scheduling-Overhead der Software-Architektur durch die besagten Sampleraten mit der jeweiligen Rechenzeitanforderung von *C_tRateXSchedule* wiedergeben. Die dafür verantwortliche Software ist aber in der Manager-Task (*tBaseRate*) enthalten. Um diese Querabhängigkeit[8] im Nachweis zu berücksichtigen, werden also die Prioritäten dieser Aktionen mit der Priorität der *tBaseRate* gleichgesetzt.

So wird erreicht, dass bei der Ausführung der Aktionen zu *tBaseRate* die Beeinflussung (z.B. Verdrängung) durch den Scheduling-Overhead aller vorhandenen Sampleraten mit berücksichtigt wird, jedoch nur mit der Häufigkeit der jeweiligen Abtastrate. Dies ist der eigentliche Nutzen aus der beschriebenen Transformation und bildet einen der Realität sehr naheliegenden Sachverhalt nach.

Bei dem zyklischen Modellteil gibt es nach der Transformation keinen *allgemeinen APG,* da keine direkten Bedingungsknoten mehr in den *Aktions-Präzedenz-Graphen* vorzufinden sind.

[8] Zum Zeitpunkt t = 0, an dem zum erstenmal alle Abtastraten des Modells bearbeitet werden, müssen zur Einhaltung der Realzeitbedingungen nicht nur die Aktionen *Scheduling0* und *tBaseRate* innerhalb der Grund-Abtastrate (Ts_base) bearbeitet werden, sondern auch *Scheduling1 ... Scheduling_n* . Auch sie sind in der Task *tBaseRate* enthalten!

4.1.4 Systembelastung durch das einbettende System

Zur Berechnung der worst-case Antwortzeit von bestimmten Aktionen auf externe Ereignisse sind, neben der worst-case Berechnungszeit aller Aktionen und der Darstellung der Zusammengehörigkeit in Aktions-Präzedenz-Graphen, auch die Beschreibung der Berechnungsanforderungen durch das einbettende System erforderlich. Zu diesem Zweck wird die Notation von Ereignisströmen nach Gresser [117] eingesetzt.

Ereignisströme ES (Beispiel in Abbildung 4.23) beschreiben das zeitliche Verhalten des einbettenden Systems. Sie erlauben eine formale Beschreibung der maximal möglichen Anzahl i von Ereignissen eines speziellen Typs in einem beliebigen Intervall a_{i-1}, das sich mit der Periodizität z_{i-1} wiederholt. Im Falle einmaliger Ereignisse ist $z_{i-1} = \infty$. Ein Ereignisstrom ES ist eine Menge von Ereignistupeln $(z_i , a_i)^T$.

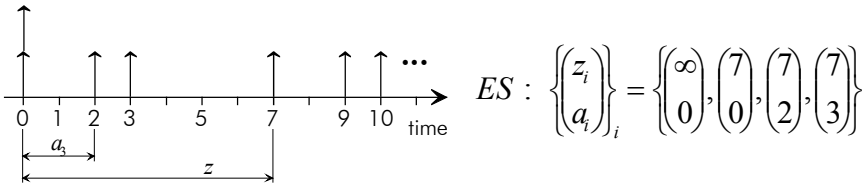

Abbildung 4.23: Ereignisstrom Beispiel

Die maximale Anzahl von externen Ereignissen EV im Zeitintervall $I = [x, x+t)$, deren zeitliches Verhalten durch den Ereignisstrom ES beschrieben wird, kann durch die Ereignisfunktion $\psi(I)$ berechnet werden.

$$\Psi\ (I) = \sum_{\forall \binom{z_i}{a_i} \in ES} \begin{cases} 0 & I \le a_i \\ \left\lceil \dfrac{I - a_i}{z_i} \right\rceil & (I > a_i) \wedge (z_i < \infty) \\ 1 & (I > a_i) \wedge (z_i = \infty) \end{cases} \qquad \textbf{(4.1)}$$

Für die maximale Anzahl im geschlossenen Zeitintervall $I = [x, x+t]$, gibt es die Funktion $\psi^+(I)$.

$$\Psi^+(I) = \sum_{\forall \binom{z_i}{a_i} \in ES} \begin{cases} 0 & I < a_i \\ \left\lfloor \dfrac{I - a_i}{z_i} \right\rfloor + 1 & (I \ge a_i) \wedge (z_i < \infty) \\ 1 & (I \ge a_i) \wedge (z_i = \infty) \end{cases} \qquad \textbf{(4.2)}$$

Durch die im Kapitel 4.1.3 beschriebene Transformation des zyklischen Systemanteils ergibt sich, ausgehend von den unterschiedlichen Abtastraten $\{T_{s1}, T_{s2}, ..., T_{sn}\}$, folgende allgemeine Beschreibung für die Ereignisströme ES_{Tsi} der unterschiedlichen *APG*.

$$ES_{T_{si}} : \left\{ \binom{T_{si}}{0} \right\} \qquad \textbf{(4.3)}$$

Für die Ereignisströme der ereignisgesteuerten *APG* kann keine allgemeine Bedingung aufgestellt werden. Sie sind aber Inhalt der nichtfunktionalen Anforderungssprezifikation bzw. müssen aus dieser extrahiert werden.

4.2 Formeln für den Realzeitnachweis

Grundsätzlich gibt es heute bei Softwaresystemen zwei Arten des Realzeitnachweises und zwar

- die mathematische Bestimmung der maximalen Auslastungsobergrenze einer Software-Architektur (*utilization bound theorem* bei zyklischen Systemen) [96]
- die iterative Berechnung der worst-case Reaktionszeit eines konkreten Software-Systems bei einer gegebenen worst-case Belastung durch das einbettende System (*completion time theorem*) [97][105][101]

Der hier vorgestellte Realzeitnachweis stützt sich auf die zweite Methode, der iterativen Berechnung der Antwortzeiten des Systems.

Ausgehend vom kritischen Augenblick (*critical instant*) wird die Reaktion des Systems auf die größte, aus der Anforderungsspezifikation ermittelten Belastung berechnet. Der kritische Augenblick des Systems definiert sich dabei als die ungünstigste Konstellation aller möglichen Berechnungsanforderungen. Er wurde bereits von Liu und Layland 1973 [96] mathematisch als der Fall bestimmt, in dem alle periodischen Systemanforderungen in Phase liegen. Durch die Verwendung von Ereignisströmen und der ihnen zugrundeliegenden Definition ergibt sich in dem hier beschriebenen System der kritische Augenblick, als die in Phase liegende Summe aller Ereignisströme, die das System belasten. Dies gilt, da in Gresser [117] die Berücksichtigung von Abhängigkeiten zwischen den Ereignisströmen als „Relaxation" der Anforderungen beschrieben wurde und daher eine synchrone Phasenlage ein Maximum der Belastung wiedergibt.

Da grundsätzlich in einem System mit ereignisgesteuerten Anteilen das Vorhandensein von mehreren Instanzen ein und desselben Ereignistyps nicht unbedingt auf eine Laufzeitverletzung schließen lässt, muss neben dem kritischen Augenblick auch das *Busy-Window* berücksichtig werden, das von Lehoczky [99] eingeführt und später von Tindell [101] weiterentwickelt wurde. Das *Level-i Busy-Window* oder auch *Level-i Busy-Period* ist als die maximale Zeit definiert, die ein Prozessor mit Tasks der Priorität i oder höher ohne Zwischenpause beschäftigt ist.

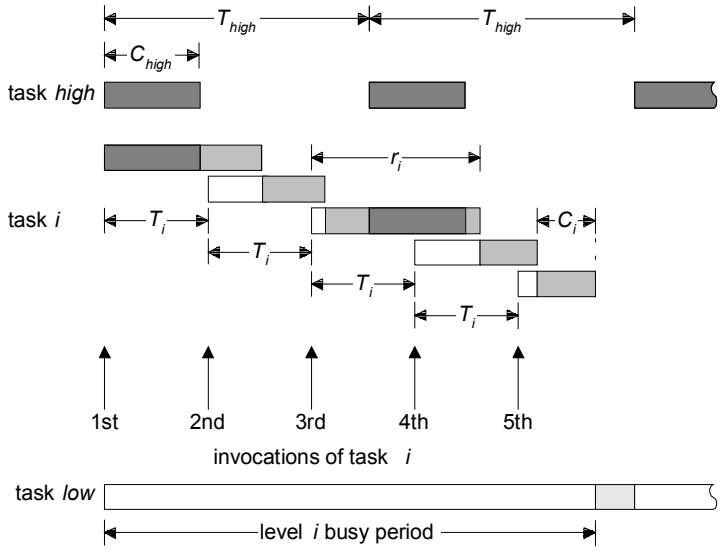

Abbildung 4.24: Beispiel eines *Level-i Busy Window* [Quelle: Tindell [101]]

Tindell beweist, dass nicht unbedingt die erste Reaktionszeit einer Instanz im *Busy-Window* immer der worst-case Reaktionszeit entspricht, sondern weitet die Untersuchung auf die Reaktionszeiten aller Instanzen einer Task während eines *Busy-Windows* aus.

Somit sind für die zuverlässige Ermittlung der worst-case Reaktionszeiten von Aktionen in einem konkreten Software-System die Berechnung aller Reaktionszeiten eines *Busy-Windows*, bei Berücksichtigung des kritischen Augenblicks beim Starten der rekursiven Berechnung, notwendig.

Weitere für das Verständnis der nachfolgenden mathematischen Formeln wichtige Funktionen sind die *Ceiling* Funktion $\left\lceil \frac{3}{2} \right\rceil = 2$ und die *Floor* Funktion $\left\lfloor \frac{3}{2} \right\rfloor = 1$.

Das grundsätzliche Vorgehen bei der hier vorgestellten Berechnung der worst-case Reaktionszeit für eine bestimmte Aktion entspricht der durch Saksena und Karvelas in [113] vorgestellten Methodik zur Bestimmung von Reaktionszeiten in Objekt-Orientierten Designs.

Im ersten Schritt wird der worst-case Startzeitpunkt der untersuchten Aktion bzw. ihrer jeweiligen Instanz im *Busy-Window* ermittelt.

Hierzu wird iterativ die Summe aus

- der maximalen Blockierzeit der betrachteten Aktion durch andere nieder-priorer Aktionen (ist konstant und ergibt sich aus der SW-Architektur)
- der Interferenz durch alle anderen Aktionen von anderen Transaktionen mit einer Priorität größer oder gleich der betrachteten Aktion
- der Interferenz durch alle Aktionen derselben Transaktion (wie die der betrachteten Aktion) bis zu der Instanz, die im Augenblick betrachtet wird
- die Interferenz durch alle Aktionen derselben Transaktion, ausgehend von der betrachteten Instanz mit allen nachfolgenden Instanzen, jedoch ohne deren Nachfolgern

berechnet, wobei jeweils durch die Ereignisfunktion und ihrer impliziten Abhängigkeit vom Berechnungsergebnis selbst (implizite Formeln), Änderungen in den verschiedenen Termen entstehen. Es kann mathematisch bewiesen werden, dass die Iterative Berechnung gegen einen stationären Wert konvergiert, falls die Auslastung des Systems kleiner als 1 ist [97][102].

Im nächsten Schritt wird, ausgehend von dem worst-case Startzeitpunkt, der worst-case Endzeitpunkt iterativ berechnet. Auch hier wird wiederum eine Summe aus

- dem zuvor berechneten Startzeitpunkt
- der Interferenz von anderen Transaktionen
- der Interferenz von der eigenen Transaktion

gebildet. Hierbei unterscheidet sich die Berechnung der Zeiten zwischen den verschiedenen Prioritätsvererbungs-Protokollen. Die Wahl eines bestimmten Protokolls wirkt sich auch auf die Berechnung der Blockierzeit der verschiedenen Aktionen aus.

Nach der Berechnung aller Endzeitpunkte der verschiedenen Instanzen einer Aktion eines *Busy-Windows*, können die verschiedenen Reaktionszeiten berechnet werden. Hierzu wird dem jeweiligen Endzeitpunkt der Ankunftszeitpunkt der Instanz abgezogen. Die worst-case Reaktionszeit der untersuchten Aktion ergibt sich dann aus dem Maximum aller ermittelter Reaktionszeiten.

Durch diese Berechnung erhält der Designer ein quantitatives Feedback über die schlimmste zu erwartende Verzögerung zwischen Eintreffen eines Ereignisses und Reaktion der Software-Architektur.

4.2.1 Blockierzeiten

Die maximalen Blockierzeiten von Aktionen ergeben sich aus:

- der konkreten Software-Architektur eines zu untersuchenden Modells und
- der gewählten Prioritätsvererbungs-Strategie zwischen Nachrichten in den Warteschlangen und den bearbeitenden Server-Threads.

Ohne eine geeignete Prioritätsvererbungs-Strategie würde es aufgrund von Prioritätsinversions-Situationen nicht möglich sein, eine eindeutige obere Grenze für die Blockierung zu berechnen.

Die hier betrachteten Blockierzeiten definieren sich aus der maximalen Zeit, die eine Aktion höherer Priorität A_i warten muss, da eine Aktion niederer Priorität A_j vorher zu Ende bearbeitet wird. Dies kommt vor, wenn z.B. ein Server-Thread gerade während der Bearbeitung von Aktion A_j eine neue, höher-priore Berechnungsanforderung A_i in Form einer Nachricht erhält, diese aber in der Warteschlange solange warten muss, bis A_j zu Ende ist (*run-to-completion* Paradigma). Um eine unbegrenzte Prioritätsinversion in dieser Situation zu vermeiden, wurden bereits zwei Strategien vorgestellt das *Basic Priority Inheritence (BPI)* und das *Preemption Threshold (PT)* Protokoll zur Vererbung der Prioritäten.

Basic Priority Inheritence Protokoll

Im Falle des *Basic Priority Inheritence Protokolls* werden sukzessive, jeweils beim Eintreffen einer höher-prioren Nachricht, die Priorität des bearbeitenden Threads an die jeweils höchste Priorität angepasst. Dies hat zwar den Vorteil, dass zur Laufzeit die effektive Beeinflussung zwischen unterschiedlichen Prioritätsebenen minimiert wird, was sich positiv auf den Jitter höher-priorer Threads auswirkt. Es hat aber den Nachteil, dass im schlimmsten Fall eine unwahrscheinliche, wenn auch sehr ungünstige Verkettung von Blockierzeiten auftreten kann. Die Folge ist, dass die errechnete maximale Blockierzeit genau diesen worst-case Fall berücksichtigen muss und somit als Summe über alle Threads des Systems berechnet wird.

Zur Bestimmung der maximalen Blockierzeit muss im ersten Schritt die maximale transiente Priorität $\gamma(A_j)$, die eine Aktion durch Vererbung während der Laufzeit einnehmen kann, berechnet werden. Diese ergibt sich aus der maximalen Ereignis-Priorität, die in einer Messagequeue empfangen werden kann, bzw. sie ist die maximale Priorität aller Aktionen, die durch ein und denselben Thread bearbeitet werden können. Dies kann unabhängig von jeglichem *allgemeinen* oder *konkreten APG* bestimmt werden. Gleichung 4.4 gibt die mathematische Berechnung der maximalen transienten Priorität $\gamma(A_j)$ von Aktion A_j wieder.

$$\gamma(A_j) = \max_{k \in \{activity\ set\}} \left(\pi(A_k) \mid \Gamma(A_k) = \Gamma(A_j) \right) \qquad \textbf{(4.4)}$$

Da bei der Verwendung des BPI-Protokoll theoretisch jeder Thread im System die Bearbeitung einer Aktion blockieren kann, muss bei der Berechnung der maximalen Blockierzeit $B(A_i)$ diese ungünstige Verkettung berücksichtigt werden. Eine Blockierung liegt vor, wenn die statische feste Priorität $\pi(A_k)$ der in Bearbeitung befindlichen Aktion A_k kleiner ist als die blockierte Aktion A_i, jedoch durch die dynamische Vererbung (BPI) eine Blockierung ermöglicht wird ($\pi(A_i)$ liegt zwischen $\pi(A_k)$ und $\gamma(A_k)$). Weiter wird für jeden Thread die Aktion mit der jeweils längsten Bearbeitungszeit angenommen. Dies entspricht somit dem „worst-case" der Blockierung. In Gleichung 4.5 ist der beschriebene Zusammenhang mathematisch wiedergegeben.

$$B(A_i) = \sum_{\forall\ threads} \max_{k \in \{activity\ \ set\}} \left\{ C(A_k) \mid \gamma(A_k) \geq \pi(A_i) > \pi(A_k) \right\} \qquad \textbf{(4.5)}$$

Preemption Threshold Protokoll

Zur Verringerung der Blockierzeit kann anstelle des BPI Protokolls eine erweiterte Strategie angewandt werden, das *Preemption Threshold Protokoll*. Es entspricht dem Priority Ceiling Protokoll zur deadlock-freien Verwaltung gemeinsam genutzter Ressourcen. Die Strategie der Prioritätsvererbung ist hier etwas anders als im vorher besprochenen Fall. Es werden a-priori jedem Ereignis bzw. jeder Aktion zwei Prioritäten zugeordnet. Zum einen die statische feste Priorität $\pi(A_i)$, wie auch im BPI Fall und zum anderen die *Preemption Threshold* Priorität, die sich wiederum aus der maximalen Priorität aller Aktionen in einem Thread ergibt, also wieder $\gamma(A_i)$ (siehe Formel 4.1) ist.

Zur Laufzeit wird vor dem Starten einer Aktion zur Scheduling-Entscheidung immer die feste Priorität der Ereignisse bzw. Aktionen herangezogen, dies entspricht der Vorgehensweise beim BPI-Protokoll. Sobald aber ein Thread durch den Scheduler zur Bearbeitung einer bestimmten Aktion gestartet wird, wird dessen Priorität von Anfang an auf die Preemption Threshold Priorität $\gamma(A_i)$ gesetzt. Dadurch wird eine verkettete Blockierung durch die anderen Threads unterbunden und es wird nur eine einmalige Blockierung durch eine andere Aktion möglich. Gleichung 4.6 gibt die mathematische Berechnung der neuen Blockierzeit mit dem PT-Protokoll wieder.

$$B(A_i) = \max_{k \in \{activity\ set\}} \{C(A_k) \mid \gamma(A_k) \geq \pi(A_i) > \pi(A_k)\} \qquad \textbf{(4.6)}$$

Der Nachteil dieser Strategie liegt nicht im Realzeitnachweis, sondern in der Performance des Gesamtsystems, da durch das sofortige Hochsetzen der Prioritäten auch im Normalfall niederpriore Aktionen höher-priore Aktionen verzögern können. Dies hat zwar keine Auswirkungen auf das Einhalten der Realzeitbedingungen (wird bei der Berechnung der worst-case Reaktionszeit bereits berücksichtigt), erhöht aber den Jitter im System.

4.2.2 Startzeitpunkte

Die Berechnung des worst-case Startzeitpunktes $S_i^\tau(q)$ ist unabhängig von den beschriebenen Prioritätsvererbungs-Protokollen, da sich diese erst beim Beginn der Bearbeitung der betrachteten Aktion unterschiedlich verhalten.

Die Berechnung des frühesten Startzeitpunktes $S_i^\tau(q)$ einer bestimmten Instanz q der Aktion A_i, die Teil der Transaktion τ ist, setzt sich, wie bereits erwähnt, aus vier Termen zusammen.

$$S_i^\tau(q) = B(A_i) + \qquad \textbf{(4.7)}$$

$$+ \sum_{k \neq \tau} \left[\Psi_k^+(S_i^\tau(q)) \cdot \sum_l \left(C(A_l^k) \mid \pi(A_l^k) \geq \pi(A_i^\tau) \right) \right] + \qquad \textbf{(4.8)}$$

$$+ (q-1) \cdot \sum_l \left(C(A_l^\tau) \mid \pi(A_l^\tau) \geq \pi(A_i^\tau) \right) + \qquad \textbf{(4.9)}$$

$$+ \left(\Psi_\tau^+(S_i^\tau(q)) - q + 1 \right) \cdot \sum_l \left(C(A_l^\tau) \mid \neg \left(A_i^\tau \rightarrow A_l^\tau \right) \wedge \pi(A_l^\tau) \geq \pi(A_i^\tau) \right) \qquad \textbf{(4.10)}$$

Der erste Term 4.7 berücksichtigt die längste Blockierzeit $B(A_i)$ der Aktion A_i, da dies der früheste mögliche Startzeitpunkt der Aktion im worst-case Fall ist.

Durch den zweiten Term 4.8 wird die Interferenz aller anderen Transaktionen im System berücksichtig. Es muss nämlich sichergestellt werden, dass alle Aktionen die Teil der anderen Transaktionen ($k \neq \tau$) sind, dessen Priorität gleich oder größer als die der betrachteten Aktion

ist, ebenfalls vor dem Start von Aktion A_i^τ bearbeitet werden müssen. Hierzu wird iterativ die maximale Anzahl von eingetroffenen Ereignissen $\psi^+(S_i^\tau(q))$ im betrachteten Zeitintervall $S_i^\tau(q)$ mit der damit verbundenen summierten Bearbeitungsdauer, der in Frage kommenden Aktionen (dessen Priorität größer oder gleich der betrachteten Priorität ist), multipliziert. Dies entspricht der kumulierten Rechenzeitanforderung durch die anderen externen Ereignisse im System.

Der dritte Term 4.9 ist nur für nachfolgende Instanzen derselben Aktion (nicht die erste Instanz $q = 1$) wichtig, die als Folge der Anhäufung von Bearbeitungswünschen derselben Transaktion τ entstehen können. Ein typisches Beispiel dafür ist das Auftreten von Burst-Events bei einem ereignisgesteuerten Modellanteil. Da alle Vorgänger-Aktionen derselben Transaktion ebenfalls die betrachtete Aktion vom Starten abhalten, wird auch hier die Summe der kumulierten Bearbeitungszeit berechnet und als Verzögerung zum Startzeitpunkt addiert.

Im vierten Term 4.10 wird nun nicht nur die aktuell betrachtete Instanz der Aktion mit ihren Vorgänger-Aktionen der Transaktion berücksichtigt, sondern auch alle nachkommenden Instanzen ($q+1, q+2, ...$), jedoch ohne deren Nachfolger ($\neg(A_i^\tau \rightarrow A_l^\tau)$). Hierbei ist wichtig zu berücksichtigen, dass zwar die Nachfolger der späteren Instanzen von A_i^τ nicht berücksichtigt werden, jedoch die Ausführungszeit der späteren Instanzen sehr wohl mit in die Berechnung eingehen. Die Implementierung garantiert zwar in der später beschriebenen realisierten Form, dass auch unter den Instanzen derselben Aktion ein FIFO Mechanismus dies verhindert, dennoch wird es hier so zum Zwecke der Allgemeinheit und für die richtige Handhabung des Scheduling-Overhead-Anteils bei der Transformation des Zyklischen Softwareteiles (siehe Kapitel 4.1.3) gehandhabt.

4.2.3 Endzeitpunkte

Im Gegensatz zum Startzeitpunkt, wirkt sich die Wahl des verwendeten Prioritätsvererbungs-Protokolls sehr wohl auf die Berechnung des Endzeitpunktes $F_i^\tau(q)$ aus. Grundsätzlich setzt sich jedoch auch hier bei beiden vorgestellten Strategien die iterative Formel aus drei Termen zusammen.

$$F_i^\tau(q) = \min(W) \quad with$$

$$W = S_i^\tau(q) + \qquad \textbf{(4.11)}$$

$$+ \sum_{k \neq \tau} \left(\Psi_k(W) - \Psi_k^+(S_i^\tau(q)) \right) \cdot L_k(A_i^\tau) + \qquad \textbf{(4.12)}$$

$$+ \left(\Psi_i(W) - \Psi_i^+(S_i^\tau(q)) \right) \cdot L_\tau(A_i^\tau) \qquad \textbf{(4.13)}$$

Der erste Term 4.11 berücksichtigt den frühestmöglichen Startzeitpunkt $S_i^\tau(q)$ der *q-ten* Instanz der betrachteten Aktion A_i^τ der Transaktion τ.

Durch den zweiten Term 4.12 wird der Interferenz bzw. der Verdrängung durch andere Aktionen der anderen Transaktionen Rechnung getragen. Wichtig ist hierbei zu berücksichtigen, dass nur Transaktions-Instanzen nach dem Startzeitpunkt $S_i^\tau(q)$ mit in die kumulative Berechnungszeit eingehen. Daher die Subtraktion von $\psi^+(S_i^\tau(q))$, da dieser Anteil bereits bei der Berechnung des Startzeitpunktes mit eingegangen ist. Der kumulative Berechnungsaufwand einer Instanz der Transaktion *k*, also $L_J(A_l^\tau)$ wird über eine weitere rekursive Formel berechnet, die neben der Priorität der Aktionen auch die Zugehörigkeit zu Threads berücksichtigt. Während der Bearbeitung der Aktion A_i kann diese nämlich nur durch Aktionen mit höherer Priorität und unterschiedlichem Thread verdrängt werden (run-to-completion Paradigma).

Der letzte Term 4.13 beschreibt den kumulativen Berechnungsaufwand der eigenen Transaktion der ebenfalls die Verdrängung durch alle Instanzen der betrachteten Transaktion nach dem Startzeitpunkt betrachtet. Auch hier wird die Zugehörigkeit zu einem Thread bei $L_j(A_i^\tau)$ berücksichtigt.

Die kumulative Berechnungszeit einer Instanz einer bestimmten Transaktion $L_j(A_i^\tau)$ unter Berücksichtigung der augenblicklichen Priorität (BPI: $\pi(A_i)$, PT: $\gamma(A_i)$) der betrachteten Aktion A_i^τ und der Threadzugehörigkeit („run-to-completion" Paradigma) wird, wie folgt, berechnet.

Basic Priority Inheritence Protokoll

Im „worst-case" ist die zu betrachtende Priorität während der Ausführung der Aktion gleich der festen statischen Priorität der Message $\pi(A_i)$.

$$L_k(A_i^\tau) = L_k(A_i^\tau, A_j^k) \qquad \textbf{(4.14)}$$

$$L_k(A_i^\tau, A_j^k) = \begin{cases} 0 & if\ \left(\Gamma(A_j^k) = \Gamma(A_i^\tau)\right) \vee \left(\pi(A_j^k) < \pi(A_i^\tau)\right) \\ C(A_j^k) + \sum\limits_{g::A_j^k \to A_g^k} L(A_i^\tau, A_g^k) \end{cases} \qquad \textbf{(4.15)}$$

Preemption Threshold Protokoll

Im Falle der Preemption Threshold Strategie ist während der gesamten Verarbeitung der Aktion die Priorität der Aktion gleich der dynamischen Priorität bzw. der Preemption Threshold Priorität $\gamma(A_i)$.

$$L_k(A_i^\tau) = L_k(A_i^\tau, A_j^k) \qquad \textbf{(4.16)}$$

$$L_k(A_i^\tau, A_j^k) = \begin{cases} 0 & if\ \left(\Gamma(A_j^k) = \Gamma(A_i^\tau)\right) \vee \left(\pi(A_j^k) < \gamma(A_i^\tau)\right) \\ C(A_j^k) + \sum\limits_{g::A_j^k \to A_g^k} L(A_i^\tau, A_g^k) \end{cases} \qquad \textbf{(4.17)}$$

4.2.4 Reaktionszeiten

Die Reaktionszeiten $R_i^\tau(q)$ errechnet sich aus den Endzeitpunkten $F_i^\tau(q)$ der spezifischen Instanz einer betrachteten Aktion A_i^τ der Transaktion τ und den zugehörigen Ankunftszeitpunkten $Arr^\tau(q)$.

$$R_i^\tau(q) = \left(F_i^\tau(q) - Arr^\tau(q)\right) \qquad \textbf{(4.18)}$$

Wie bereits erwähnt gilt als „worst-case" Reaktionszeit die längste Reaktionszeit aller Instanzen einer Aktion in dem selben Level-i Busy-Window. Somit ergibt sich für die zu berechnende „worst-case" Reaktionszeit folgender Zusammenhang:

$$R_{i\max}^\tau = \max_{q\in[1,\ldots,m]}\left(F_i^\tau(q) - Arr^\tau(q)\right) \qquad \textbf{(4.19)}$$

Diese mathematischen Beziehungen sind wiederum unabhängig von der verwendeten Prioritätsvererbungs-Strategie, nicht aber die eingesetzten Werte für die Endzeitpunkte $F_i^\tau(q)$.

Die Umsetzung und Anwendung der hier beschriebenen Realzeitnachweis-Methode ist in Kapitel 5.2.3 anhand eines Beispiels veranschaulicht.

5 Umsetzung bei Simulink/Stateflow

Im folgenden Abschnitt wird die Umsetzung der beschriebenen SW-Architektur und des vorgestellten Realzeitnachweises im Design-Werkzeug Simulink/Stateflow präsentiert.

Dazu werden im ersten Teil die notwendigen Anpassungen und Erweiterungen der Werkzeugkette zur Realisierung der neuen Konzepte aus Abschnitt 3 auf graphischer Ebene erläutert.

Der zweite Teil befasst sich mit der Durchführung des in Kapitel 4 präsentierten Realzeitnachweises und beinhaltet die Beschreibung der dafür entwickelten Skripten (Umsetzung der iterativen und rekursiven Formeln aus Kapitel 4.2).

5.1 Software-Architektur

Das Hauptproblem der ursprünglichen SW-Architektur ist das Fehlen jeglicher Ereignispufferung und damit auch das Fehlen einer effektiven Unterstützung zur Umsetzung ereignisgetriggerter Systemanteile.

Die neu entwickelte Architektur kompensiert diese Lücke durch den Einsatz des in Abschnitt 3.2.1 beschriebenen Grundelementes, bestehend aus einer Messagequeue und einem ihr zugeordneten Server-Thread. Weiter wurde in dieser Arbeit bei der Abbildung der funktionalen Spezifikation eine neue Sicht auf die SW-Architektur vorgeschlagen, die zu einem Ereigniszentrierten System führt. Die sich daraus ergebenden Vorteile wurden bereits ausführlich in 3.2.2 beschrieben.

5.1.1 Graphische Modellierung

Zur Umsetzung der neuen Konzepte auf graphischer Ebene wurde in Simulink/Stateflow die Beschreibung der neuen Komponenten, wie

- Messagequeues mit Server-Threads und
- Ereignisse,

ermöglicht.

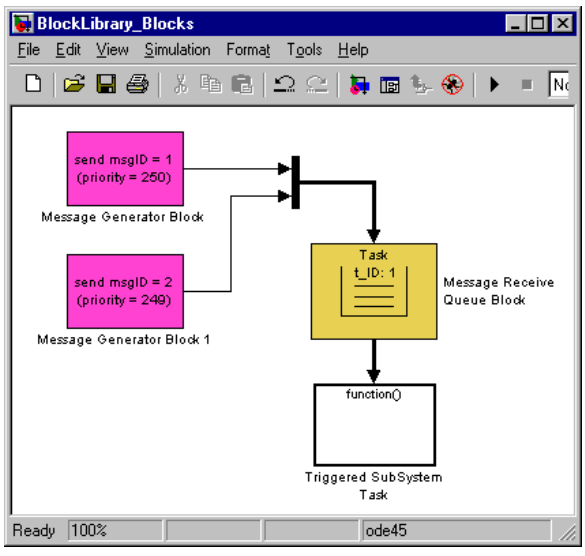

Abbildung 5.1: Beispiel eines einfachen Modells mit einem ereignisgesteuerten Server-Thread (*Triggered SubSystem Task*) der durch seine Messagequeue (*Message Receive Queue Block*) zwei unterschiedliche Ereignisse ($msgID = 1$, $msgID = 2$) empfängt.

Die standardmäßig in Simulink/Stateflow vorhandene Möglichkeit zur Modellierung der Software-Architektur einer Implementierung, beschränkt sich lediglich auf die in Kapitel 2.3.3 beschriebene Anbindung einer Task an eine ISR.

Daher mussten zur Modellierung der besagten Komponenten zwei neue graphische Blöcke definiert und entwickelt werden (siehe Abbildung 5.1):

- der *Message Generator Block* und
- der *Message Receive Queue Block*.

Message Generator Block

Der *Message Generator Block* modelliert das Versenden von Ereignissen bzw. von Berechnungswünschen auf graphischer Ebene. Er dient somit als Quelle externer und interner Ereignisse im graphischen Modell. Bei jeder Ausführung des Blocks wird genau ein Ereignis vom definierten Typ versendet.

Die Parameter des *Message Generator Blocks* (siehe Abbildung 5.2) sind:

- Message-Identifier (*msgID*)
- Priorität
- External-Event-Flag
- Sample-Rate.

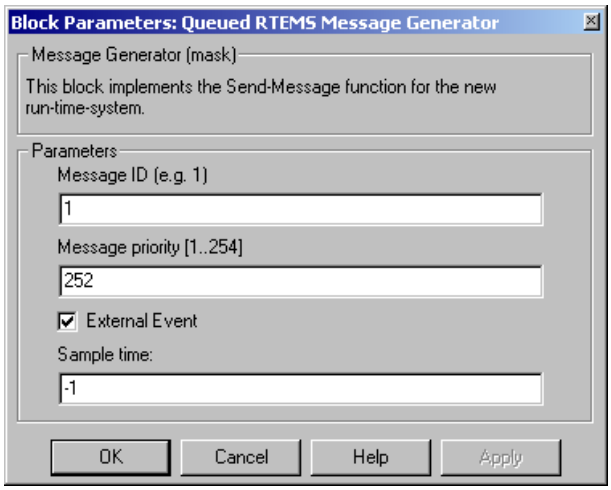

Abbildung 5.2: Parameter-Eingabefenster für *Message Generator Block*

Der Message-Identifier wird sowohl zur eindeutigen modellübergreifenden Erkennung des gesendeten Ereignisses herangezogen als auch zur Definition des Namens der bearbeitenden Aktion im System verwendet. Zum Beispiel erhält bei einem Ereignis mit der $msgID = 1$ die im System ausgelöste Aktion den eindeutigen Namen $A1$. Dadurch wird die Zuordnung der Aktionen des *Aktions-Präzedenz-Graphen* zum graphischen Modell erleichtert.

Durch die statische Priorität im *Message Generator Block* wird festgelegt, welche Priorität die versendeten Nachrichten haben sollen. Es sind Werte zwischen 1und 254 erlaubt, wobei 1 die niederste Priorität darstellt.

Durch die Check-Box „External-Event-Flag“ wird dem System mitgeteilt, dass der besagte *Message Generator Block* ein externes Ereignis versendet und somit durch eine externe Ereignis-Quelle, wie z.B. einem Interrupt, getriggert wird. Dieses Flag wird beim automatischen Parsen des graphischen Modells zur Ermittlung der Aktions-Präzedenz-Graphen des Systems verwendet (siehe Kapitel 5.2.1).

Der letzte Parameter, die Sample-Rate, ermöglicht die zyklische Ausführung des *Message Generator Blocks*. Soll zum Beispiel ein ereignisgesteuerter Systemteil durch einen Tick-Trigger zyklisch geweckt werden, wird die Abtastrate des *Message Generator Blocks* mit der gewünschten Periodendauer gesetzt.

Wird die Sample-Rate mit –1 definiert, entspricht dies einer *Inherited-Sample-Rate* und der Block erbt die Abtastrate vom restlichen Modell durch die Signal-Anbindung. Diese Option wird verwendet, falls er Teil eines Function-Call-Subsystems[9] ist und somit keine vordefinierten Abarbeitungszeitpunkte besitzt.

Im ereignisgesteuerten Modellteil werden solche Function-Call-Subsysteme mit *Message Generator Blöcke* verwendet, um bei der Verarbeitung von Aktionen Folge-Ereignisse zu versenden.

Die Signal-Leitung am Ausgang eines jeden *Message Generator Blocks* muss immer in einem *Message Receive Queue Block* enden, da nur dieser das versendete Ereignis geeignet weiterverarbeiten kann.

Message Receive Queue Block

Der *Message Receive Queue Block* modelliert die Empfangs-Seite von Ereignissen auf graphischer Ebene. Das an ihm angeschlossene Function-Call-Subsystem (siehe Abbildung 5.1) beinhaltet, in graphischer Form spezifiziert, die Funktion des Server-Threads. Dadurch kann das Grundelement der vorgestellten SW-Architektur (siehe Abbildung 3.1) sehr einfach auf graphischer Ebene modelliert werden.

Die Parameter des *Message Receive Queue Blocks* (siehe Abbildung 5.3) sind:

- Task-Name
- Task-Identifier (*taskID*)
- Task-Priorität
- Stack-Größe
- Messagequeue Länge
- Anzahl unterschiedlicher Ereignistypen
- Preemption-Threshold Priorität.

Der Task-Name wird optional beim Erzeugen des Threads im unterlagerten Laufzeitsystem verwendet. Er erleichtert ein mögliches Source-Level-Debugging der konkreten Software-Architektur.

Der Task-Identifier *taskID* wird analog zum Message-Identifier zur eindeutigen Zuordnung von Aktionen zu Threads verwendet. Er wird daher auch bei der Beschreibung von Aktionen im APG benutzt.

Die Task-Priorität im *Message Receive Queue Block* gibt die anfängliche Priorität des im Laufzeitsystem erzeugten Server-Threads an. Sie ist nur während der Initialisierung gültig, da während der Realzeitausführung der Software die Server-Task ihre Priorität durch die empfangenen Ereignisse erhält (Prioritätsvererbungs-Strategien).

Die Stack-Größe definiert die Größe des bei der Task-Erzeugung angelegten Task-Stacks. Die Messagequeue-Länge ist die Tiefe der erzeugten Warteschlange des Server-Threads. Durch die Realzeituntersuchung ist es möglich zuverlässige Aussagen über die maximal zu erwartende Anzahl an Ereignissen zu treffen, die in den jeweiligen Puffern gleichzeitig warten müssen.

---

[9] Der Inhalt eines Function-Call-Subsystems wird bei der Codegenerierung zu einer C-Funktion zusammengefügt und jeweils beim Auftreten des Trigger-Ereignisses, das am oberen Eingang des Subsystems angeschlossen ist, ausgeführt.

Die Anzahl unterschiedlicher Ereignistypen gibt an, wieviel Signale von unterschiedlichen *Message Generator Blöcken* in den *Message Receive Queue Block* münden. Diese Information wird zur Realisierung des Simulations-Verhaltens des Blockes benötigt.

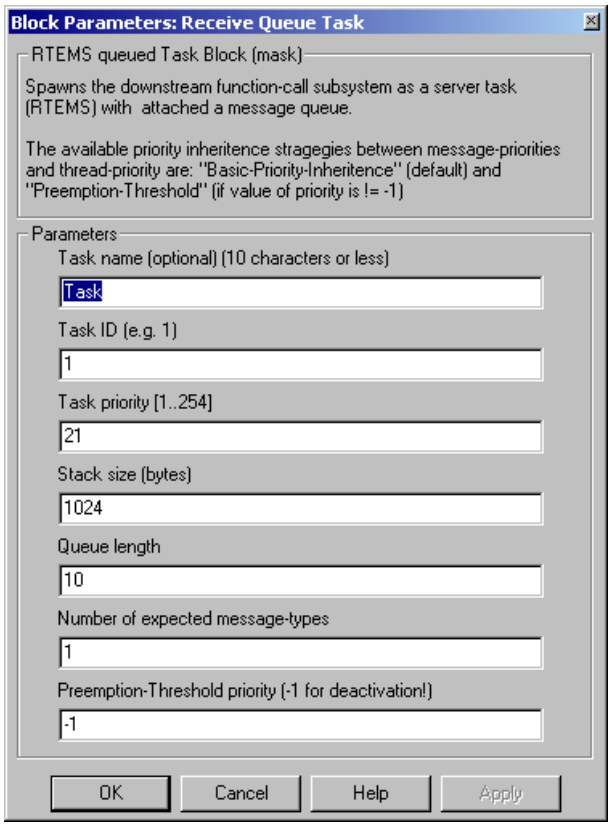

Abbildung 5.3: Parameter-Eingabefenster für *Message Receive Queue Block*

Durch den letzten Parameter, der Preemption-Threshold Priorität, ist es möglich, auf das verwendete Prioritätsvererbungs-Protokoll zwischen Nachrichten in der Warteschlange und dem Server-Thread Einfluss zu nehmen. Ist der Wert auf –1 gesetzt, wird automatisch das *Basic-Priority-Inheritence (BPI)* Protokoll verwendet. Durch einen Wert zwischen 1 und 254 wird hingegen das *Preemtption-Threshold (PT)* Protokoll aktiviert, wobei im RT-Nachweis angenommen wird, dass bei der Verwendung des *PT*-Protokolls diese Priorität der maximal zu erwartenden Ereignis-Priorität der betrachteten Warteschlange entspricht. Bei dieser Wahl der Ceiling-Priorität wird der Vorteil des PT-Protokolls bereits wirksam (Minimierung der Blockierzeit). Die Beeinflussung höherer Prioritäts-Ebenen durch Blockierungen ist jedoch minimal (nur *notwendige* Blockierungen treten auf).

Modellierung mit den neuen Blöcken

In den folgenden Abbildungen sind die in Kapitel 4.1.3 beschriebenen unterschiedlichen Grundstrukturen der Architektur auf Modellebene wiedergegeben.

Als externe Ereignisquelle wird in allen Modellen ein *External Interrupt Registration Block* verwendet, der für eine bestimmte Plattform (in diesem Fall den MPC555 und RTEMS als RTOS, siehe Kapitel 6) die Registrierung von Interrupt Service Routinen ISRs erledigt.

So werden zum Beispiel durch die graphische Spezifikation in Abbildung 5.4 die im Subsystem *ISR* enthaltenen Blöcke als Funktion des IRQ1-Handlers in der SW-Architektur umgesetzt.

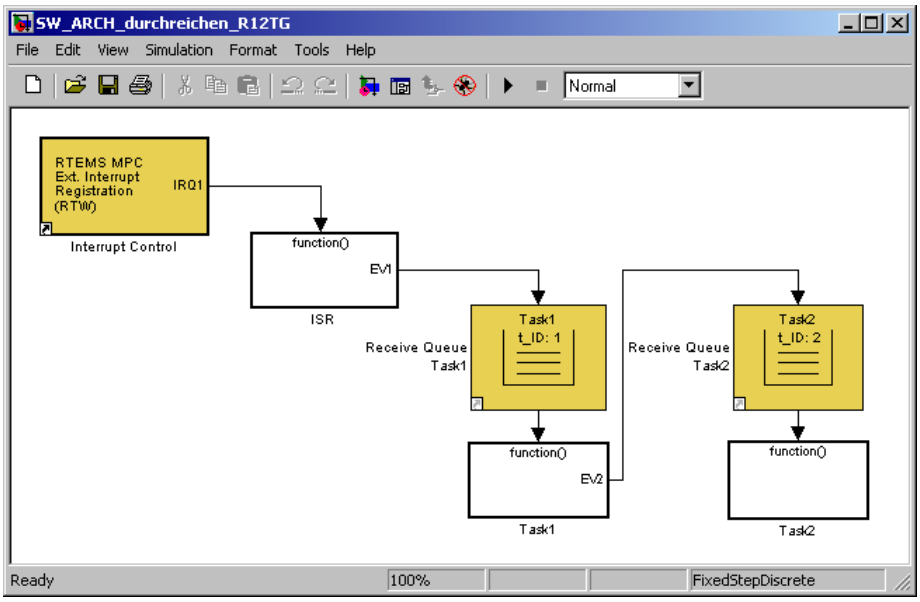

Abbildung 5.4: Modell für eine einfache Verkettung (siehe **Abbildung 4.7**)

In diesem *ISR* Subsystem ist der *Message Send Block* von *EV1* enthalten, der das Versenden des Ereignisses *EV1* beim Eintreffen eines IRQs bewerkstelligt.

Ist in einem $Task_x$ Subsystem ein weiterer *Message Send Block* enthalten, kann je nach Applikationsverhalten, bei der Bearbeitung eines Ereignisses, also bei der Ausführung einer Aktivität ein neues internes Ereignis (z.B. *EV2*) versendet werden.

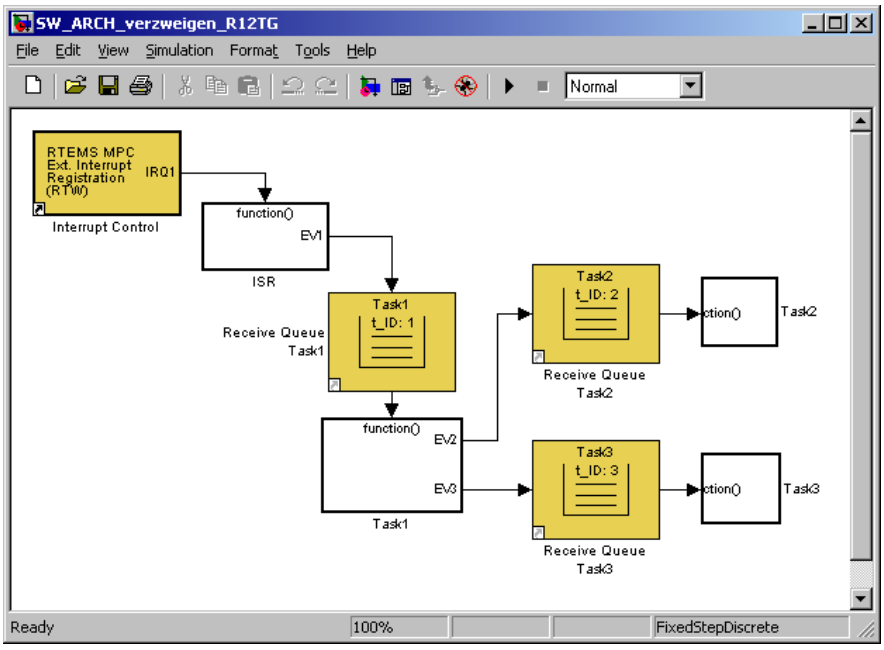

Abbildung 5.5: Modell für eine Verzweigung (siehe **Abbildung 4.9**)

Zur Abbildung des zyklischen Anteils eines graphischen Modells waren keine Erweiterungen auf Modellebene erforderlich, da nach wie vor auch in der neuen SW-Architektur die Abbildung automatisch und entsprechend der ursprünglichen SW-Architektur (siehe Abbildung 2.12) vorgenommen wird.

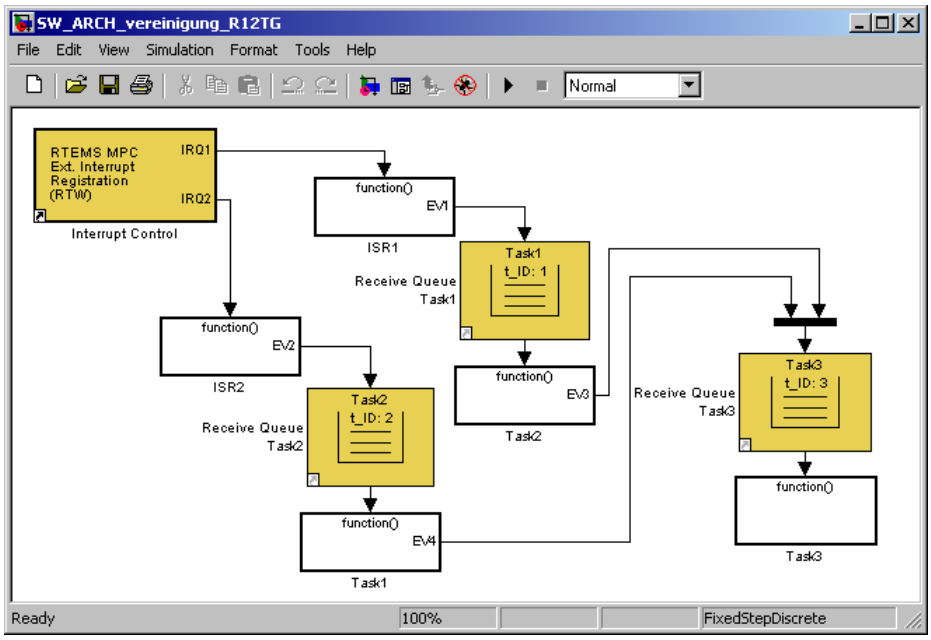

Abbildung 5.6: Modell für eine Vereinigung (siehe **Abbildung 4.14**)

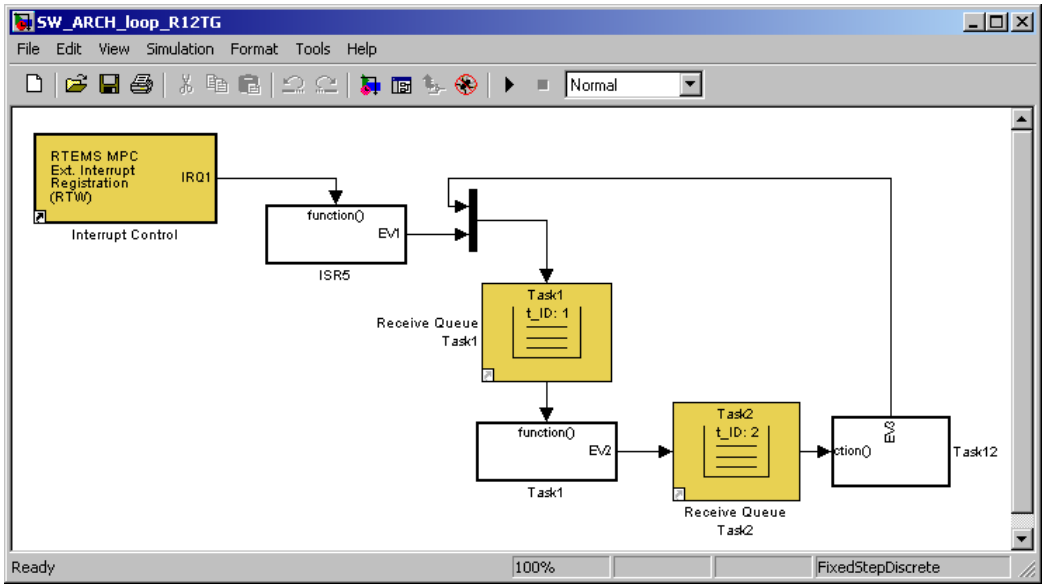

Abbildung 5.7: Modell für eine Schleife (siehe **Abbildung 4.16**)

5.1.2 RT-Implementierung (Codegenerierung)

Nachdem die neuen Blöcke vorgestellt wurden, werden nun die Anpassungen zur Abbildung eines damit spezifizierten Modells auf ein geeignetes Laufzeitsystem (siehe Kapitel 6) beschrieben.

Dafür wurde die automatische Codegenerierung der Werkzeugkette mit dem *Real-Time-Workshop*, dem *Embedded-Coder* und dem *Stateflow-Coder* (Abbildung 2.3) angepasst und erweitert.

Der Codegenerierungs-Prozess der gewählten Werkzeugkette ist) vom Hersteller sehr flexibel gestaltet worden, um die Abbildung graphischer Modelle auf eine Vielzahl unterschiedlicher Laufzeitsysteme zu unterstützen. Die Abbildung der neuen Konzepte war somit relativ einfach umsetzbar.

Message Generator Block

Der *Message Generator Block* dient zur Beschreibung von Ereignis-Quellen im graphischen Modell. Während der Codegenerierung für die Realzeitimplementierung wird sein funktionales Verhalten, d.h. sein Funktions-Code durch die Verwendung geeigneter Betriebsystem-Funktionen, wie z.B. `mq_send` der POSIX4 API umgesetzt.

Dadurch wird das Versenden von Ereignissen im graphischen Modell auf reale RTOS Nachrichten im Laufzeitsystem abgebildet. Die zur Umsetzung der Funktionsaufrufe notwendigen Parameter werden teils über die im jeweiligen *Message Generator Block* angegebenen Parameter spezifiziert (siehe Abschnitt 5.1.10: ID, Priorität) und teils aus der graphischen Zusammensetzung und Signalführung im Modell ermittelt (Referenz der Empfangs-Queue).

Die konkrete Umsetzung wurde mittels der Hersteller-spezifischen Target-Language-Compiler (TLC) Sprache [73] realisiert (siehe Anhang A), die eine abstrakte Spezifikation des zu generierenden Zielcodes (hier C-Code) zulässt.

Message Receive Queue Block

Der *Message Receive Queue Block* dient zur Beschreibung der Empfangs-Seite von Ereignissen im graphischen Modell. Bei der echtzeitfähigen Implementierung bildet jeder *Message Receive Queue Block* eine Nachrichten-Warteschlange mit einem entsprechenden Server-Thread auf dem unterlagerten Laufzeitsystem ab. Hierzu werden bei der Codegenerierung einige grundlegende Anpassungen durchgeführt.

Zum einen wird durch den Initialisierungscode des Blockes ein neuer Thread (der Server-Thread des *Message Receive Queue Blocks*) im Laufzeitsystem erzeugt. Die dafür notwendigen Parameter werden auf graphischer Ebene durch die Blockparameter angegeben. Diesem Thread wird die automatisch generierte Funktion des angebundenen Function-Call Subsystems (siehe Abbildung 5.1) zugeordnet.

Zum anderen wird durch den *Message Receive Queue Block* bei der Codegenerierung der einfache Funktionsaufruf des Function-Call Subsystems in eine Endlosschleife umgewandelt, da die Programmausführung eines RTOS-Threads nie enden darf.

Als letzte Anpassung des generierten Codes muss an einer geeigneten Stelle in der erzeugten Endlosschleife das Empfangen bzw. Entnehmen von Nachrichten aus der Warteschlange durch eine Betriebsystemfunktion, wie z.B. `mq_receive` der POSIX4 API integriert werden. Um die in der funktionalen Spezifikation definierte Zuordnung von Trigger-Ereignissen auch im RT-Code umzusetzen, wurden die Nachrichteninhalte zum Versenden von Berechnungswünschen im RTOS wie folgt definiert:

```
struct mq_message {
   int msgID;     /* message id to identify corresponding action */
   int FcnPortElement; /* FcnPortElement value */
};
```

Durch den `FcnPortElement` Parameter der jeweils empfangenen Nachricht ist es möglich, zwischen der Bearbeitung der verschiedenen empfangbaren Ereignisse in der Server-Thread Funktion zu unterscheiden.

Auch für den *Message Receive Queue Block* wurde die Codegenerierung durch TLC-Files realisiert (siehe Anhang A).

5.1.3 Simulation

Neben der Realisierung der RT-Codegenerierung für die neuen Blöcke wurde auch ein angemessenes Simulations-Verhalten entwickelt, um den Hauptvorteil eines modell-basierten Entwurfsprozesses beizubehalten.

Da sich die augenblickliche Laufzeitumgebung während der Simulation jedoch grundlegend von der Laufzeitumgebung auf dem RT-Zielsystem (RTOS) unterscheidet, war bei der Realisierung der Simulations-Funktion der Blöcke nur eine Kompromisslösung zwischen

- der Übereinstimmung von Simulations- und Implementierungs-Verhalten und
- dem erforderlichen Umsetzungsaufwand im Rahmen dieser Arbeit

möglich.

Um eine vollkommene Übereinstimmung beider Verhalten zu erreichen, wären nach eigenen Einschätzungen größere Eingriffe in die Simulationsumgebung notwendig gewesen, die neben hersteller-internem Know-How auch Ressourcen, wie den Source-Code des Werkzeuges (Simulink/Stateflow) erforderlich gemacht hätten.

Um jedoch die Auswirkungen des gewählten, abweichenden Simulationsverhaltens durch ein geeignetes Design der Modelle zu minimieren (Divergenz zwischen Simulation und Implementierung), wird dem Entwickler in Abschnitt 5.1.5 ein Leitfaden gegeben, der auf die möglichen Probleme aufmerksam macht.

In Abschnitt 5.1.6 ist zudem kurz ein Forschungsvorhaben beschrieben, durch dessen Konzepte eine Beseitigung der Unterschiede zwischen Simulations- und Implementierungs-Verhalten der hier beschriebenen SW-Architektur möglich werden könnte, ohne direkt in das Simulations-Laufzeitsystem von Simulink/Stateflow einzugreifen.

Message Generator Block

Das Simulations-Verhalten des *Message Generator Blocks* entspricht dem eines Funktion-Call-Generator Blocks der Simulink Standard-Bibliothek. Beim Versenden einer Nachricht, also beim Ausführen des Blockes, wird in der Simulation der über das Signal angebundene Empfänger-Block (immer ein *Message Receive Queue Block*) als getriggertes Subsystem in Form eines Funktions-Aufrufes sofort ausgeführt.

Da im aktuellen Simulations-Laufzeitsystem keine Möglichkeit besteht, eine asynchrone Ausführung unterschiedlicher paralleler Tasks zu beschreiben, wird die in der neuen SW-Architektur eigentlich asynchrone Berechnungswunsch-Anforderung während der Simulation als direkter Funktionsaufruf des Systems umgesetzt.

Message Receive Queue Block

Das eigentliche Simulations-Verhalten des *Message Receive Queue Blocks* ergibt sich direkt aus dem Verhalten des *Message Generator Blocks*. Der empfangene Function-Call wird im Simulations-Verhalten einfach an das angebunde Triggered-Subsystem, das die Funktionsbeschreibung des Server-Threads enthält, weitergereicht.

Um bei der Codegenerierung jedoch geeignete Eingriffspunkte im Funktionsaufruf-Fluss zu bekommen, wird der eigentliche Function-Call des *Message Generator Blocks* über ein Enabled-Subsystem und einen neuen, eigenen Function-Call-Block propagiert.

Zur Anpassung dieser im *Message Receive Queue Block* enthaltenen Function-Call Handhabung, wurde neben der Grundstruktur des Blocks in der Bibliothek, auch ein Skript (`rtems_task_rcv_msg_build.m`) programmiert, das automatisch das Innenleben des *Message Receive Queue Blocks* an die Anzahl unterschiedlicher angebundener Function-Call Eingänge anpasst (siehe Anhang A).

5.1.4 Divergenz zwischen Simulation und Implementierung

Aus der beschriebenen RT-Implementierung und der Simulations-Funktion der Blöcke ergeben sich Unterschiede zwischen dem Realzeit- und dem Simulations-Verhalten spezifizierter Modelle.

Grundsätzlich liegt die Ursache dafür in der unterschiedlichen Abarbeitungsreihenfolge der Sende-Aufrufe im System. Ein Message-Send in der RT-Implementierung kann nie zu einer Unterbrechung der aktuellen Aktion führen (siehe Bedingungen in Kapitel 4.1.3), wogegen bei der Simulation, durch die einzig mögliche Umsetzung des *Message Generator Blocks* als Function-Call-Generator, das Senden in einem direkten Funktionsaufruf umgesetzt wird (siehe Abbildung 5.8).

Die Folge ist, dass die nach dem Versenden eines Ereignisses durchgeführten Bearbeitungen zwischenzeitig durch die Behandlung des Funktionsaufrufes in der Simulation zurückgestellt werden.

Der *asynchrone* Funktionsaufruf der RT-Implementierung wird bei der Simulation durch einen *synchronen* Funktionsaufruf nachgebildet. Aus dieser Erkenntnis ergeben sich die möglichen Probleme bzw. Unterschiede zwischen Simulation und Implementierung und die notwendigen Hinweise bzw. Einschränkungen für den Entwickler.

5.1.5 Richtlinien für den Entwickler

Der Hauptgrund für eine mögliche Divergenz zwischen Simulations- und Implementierungs-Verhalten ist die unterschiedliche Abarbeitungsreihenfolge bei einer Ereignis-Triggerung.

Grundsätzlich muss daher dem Entwickler bewusst werden, welche Art der Modellierung die Auswirkungen dieser Unterschiede auf das eigentliche Modellverhalten minimiert.

Abbildung 5.8 zeigt die Abarbeitungsreihenfolge beim Senden eines Ereignisses durch den *Message Generator Block* bei

- der *Simulation* als synchronen Funktionsaufruf und bei
- der *RT-Implementierung* als asynchronen Aufruf (senden einer Nachricht mit gleicher oder niederer Priorität als aktuelle Aktion).

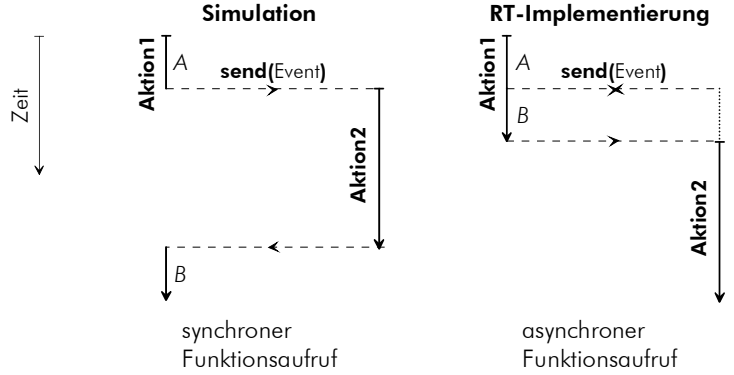

Abbildung 5.8: Unterschiede in der Abarbeitungsreihenfolge zwischen Simulation und RT-Implementierung eines *Message Generator Blocks*

Aus der Graphik wird ersichtlich, dass, die in der Simulation vorgezogene Bearbeitung der *Aktion2* keine Auswirkungen auf das Systemverhalten hat, solange eine Übereinstimmung zwischen Simulation und RT-Implementierung gegeben ist. In diesem Fall kann eine Verifikation des System-Verhaltens anhand einer Simulation durchgeführt werden.

Am einfachsten kann dies in einem Design gewährleistet werden, wenn der Inhalt des *Aktionsteils B* der *Aktion1* so gering wie möglich gehalten, oder mit anderen Worten, das Versenden

von Trigger-Ereignissen immer erst am Ende einer Aktion durchgeführt wird. Daraus ergibt sich eine klare Empfehlung für die funktionale Spezifikation auf der graphischen Modellebene.

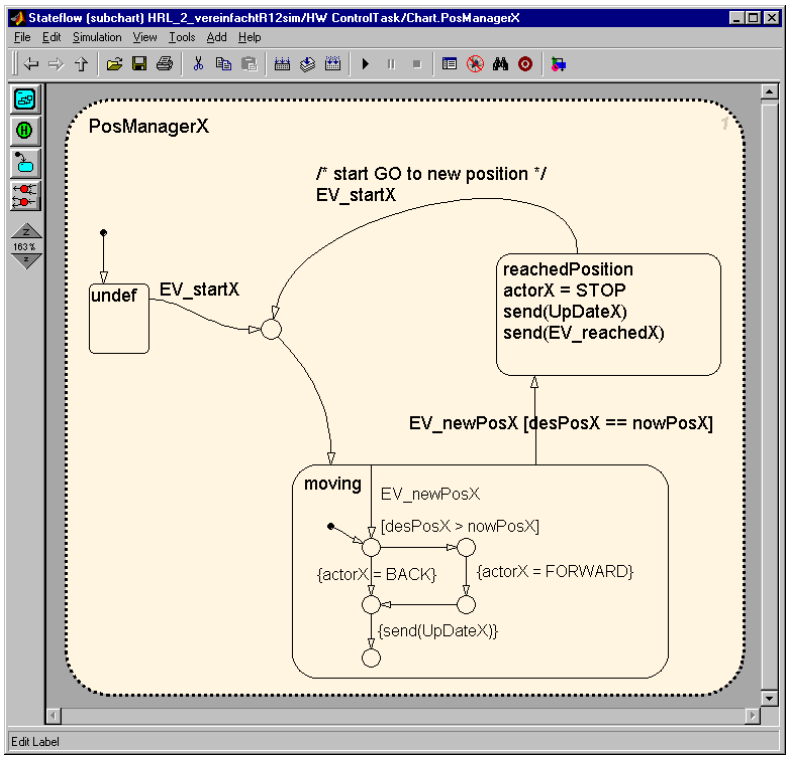

Abbildung 5.9: Beispiel eines Zustandsautomaten in Stateflow mit dem Sendevorgang als letzte Aktion der jeweiligen Bearbeitung

Die Anwendung dieser Empfehlung bei einem Zustandsautomaten in Stateflow, der Ereignisse an Nachfolge-Aktionen versendet, sieht somit wie in Abbildung 5.9 dargestellt aus. Sowohl das Ereignis *UpDateX* als auch das Ereignis *EV_reachedX* werden immer am Ende, nach allen anderen Bearbeitungen, verschickt.

Da im beschriebenen Beispiel beim Eintritt in den Zustand *reachedPosition* gleich zwei Ereignisse verschickt werden, ist es wiederum wichtig, dass eine unterschiedliche Bearbeitungsreihenfolge der beiden Nachfolge-Aktionen (möglicherweise durch unterschiedliche Prioritäten) keine funktionale Veränderung des Systems mit sich bringt. Diese Forderung wurde bereits im Kapitel 3.1.3 bei der Definition eines *guten Designs* erläutert.

Eine weitere Empfehlung für die Entwicklung von Stateflow-Automaten ist die Verlagerung von Aktionen aus den Transitionen in die Entry-Position von Zuständen. So kann vermieden werden, dass beim Versenden von Ereignissen durch eine Ereignis-Schleife, d.h. durch ein erneutes Triggern des Sende-Automaten schwer nachvollziehbare Effekte entstehen, die nur durch eine genaue Untersuchung der Laufzeitimplementierung gelöst werden können.

5.1.6 Simulation eines RT-Kernels (*TrueTime*-Simulator)

Nachdem die realisierte Umsetzung beschrieben und auch die Problematik der gewählten Simulations-Realisierung erläutert wurde, wird in diesem Abschnitt kurz das Forschungsprojekt *„Integrated Control and Scheduling“*, das eine Zusammenarbeit des *Department of Automatic Control* und des *Department of Computer Science* am *Lund Institute of Technology* ist, vorgestellt.

Durch die dort entwickelten neuen Konzepte könnte eine Beseitigung der Unterschiede zwischen Simulations- und Implementierungs-Verhalten möglich werden, ohne das Simulations-Laufzeitsystem des gewählten Werkzeuges zu verändern.

Im Rahmen des Projektes wurde der Simulator *TrueTime* [86][89][90] entwickelt, der ebenfalls auf Matlab/Simulink basiert. Dieser ermöglicht, Regelungs-Tasks in einem Realzeit-Betriebssystem (RTOS) zusammen mit kontinuierlichen Umgebungsmodellen zu simulieren. Es wurden dazu Datenstrukturen implementiert, die für ein RTOS typisch sind (Ready-Queue, Scheduler, etc.). Während einer Simulation werden die Strukturen zum Emulieren eines realen RTOS Verhaltens eingesetzt.

TrueTime ermöglicht somit die Simulation eines virtuellen Realzeit-Betriebsystems und wird im Augenblick zur Bestimmung von Jitter-Einflüssen auf Reglermodelle und zur Evaluierung neuartiger Scheduling-Strategien verwendet.

Durch eine angemessene Erweiterung der Grundfunktionalität dieses virtuellen RT-Kernels sollte es auch möglich sein, das Verhalten der in dieser Arbeit beschriebenen Konzepte umzusetzen und dadurch eine vollkommene Übereinstimmung zwischen Simulation auf dem Host-System und der letztendlichen RT-Implementierung auf dem Target zu erreichen.

5.2 RT-Nachweis

Die Simulation und Abbildung eines graphischen Modells auf die vorgestellte Software-Architektur sind nur ein Teil, der im Zusammenhang mit dieser Arbeit realisierten Umsetzung. Nachfolgend werden das Vorgehen und die semi-automatische Durchführung des Realzeitnachweises für ein gegebenes graphisches Modell erläutert.

Dabei wird nicht weiter auf die Theorie der bereits in Kapitel 4.2 vorgestellten Analyse eingegangen, sondern es werden vielmehr die verschiedenen Phasen der RT-Untersuchung im Kontext einer modell-basierten Entwicklung beschrieben.

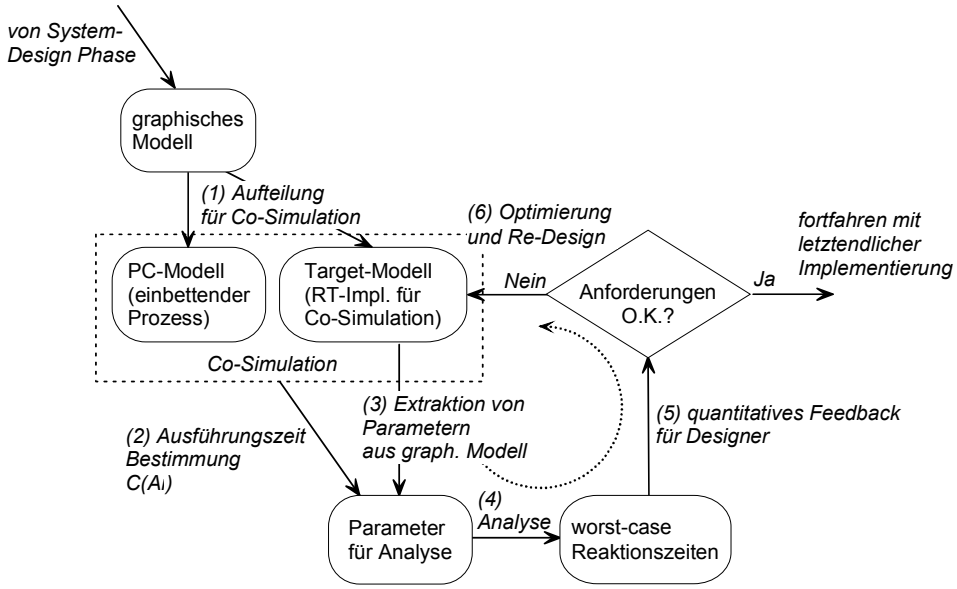

Abbildung 5.10: Flussdiagramm zur Durchführung der Realzeitanalyse für eine gegebene graphische Spezifikation

Durch die Ergebnisse des Realzeitnachweises ist ein quantitatives Feedback über die Qualität einer gewählten Umsetzung möglich, die z.B. die Wahl der Partitionierung von Funktionen auf Threads oder die Zuweisung von unterschiedlichen Prioritäten zu den Ereignissen betrifft. Damit ist die notwendige Grundlage für eine schrittweise Optimierung des Designs gegeben.

Bei einem model-basierten Entwurfsprozess (siehe Kapitel 1.2) entstehen durch die verschiedenen Phasen der Entwicklung unterschiedliche Modelle des umzusetzenden Systems. Der

Ausgangspunkt für den hier vorgestellten RT-Nachweis (siehe Abbildung 5.10) ist das endgültige Simulations-Modell, das zur funktionalen Validierung des Designs entwickelt wurde. Dieses Modell beinhaltet sowohl die zu entwickelnde Funktion als auch ein abstraktes Abbild des einbettenden Prozesses.

Ausgehend von dieser graphischen Spezifikation werden zur Parameterbestimmung durch eine Co-Simulation (siehe Abschnitt 5.2.10) vorerst zwei Modelle gebildet *(1)*, die zum einen den einbettenden Prozess (PC-Modell) und zum anderen das eigentlich zu entwickelnde System enthalten (Target-Modell).

Durch die Durchführung der Co-Simulation werden dann realistische Ausführungszeiten (nicht worst-case) für die Umsetzung der gegebenen graphischen Spezifikation auf dem Zielsystem gemessen *(2)*. Weitere für die Realzeitanalyse notwendige Parameter werden aus dem Modell der RT-Implementierung gewonnen *(3)*.

Nachdem alle Parameter ermittelt wurden, wird die Analyse *(4)* zur Bestimmung der worst-case Reaktionszeiten durchgeführt. Dabei werden für ausgewählte Aktionen iterativ alle Zeiten berechnet.

Diese quantitativen Angaben der Antwortzeiten auf bestimmte Ereignisse des umgesetzten Modells werden anschließend dem Designer als Feedback der Analyse *(5)* übergeben. Er entscheidet letztendlich über das Einhalten aller geforderten Deadlines und kann gleichzeitig Aussagen über die reale Auslastung des Systems treffen und mögliche zukünftige Engpässe oder Probleme frühzeitig erkennen.

Nachfolgend wird die Parametergewinnung beschrieben, die teilweise automatisiert abläuft. Am Ende dieses Kapitels wird die Umsetzung der Formeln als Skripten kurz wiedergegeben und die Anwendung an einem Beispiel gezeigt.

5.2.1 Parametergewinnung

Die für die hier beschriebene Analyse notwendigen Parameter lassen sich durch ihre Herkunft in drei grundlegende Kategorien einteilen. Sie werden entweder durch

- die konkrete Implementierung durch eine bestimmte Laufzeitumgebung (RTOS) und eine spezielle Hardware,
- das graphische Modell (Design), oder
- die Spezifikation des Systems (*Requirements*)

definiert.

Parameter aus der konkreten Implementierung

Die für die Realzeitanalyse relevanten Daten, welche aus der konkreten Implementierung gewonnen werden, sind die worst-case Ausführungszeiten der einzelnen Aktivitäten des Systems.

Die Bestimmung der worst-case Ausführungszeiten von Software für eine bestimmte Hardware-Architektur ist Gegenstand vieler Forschungsarbeiten [118][120][122]. Grundsätzlich können diese implementierungsabhängigen Zeiten entweder durch Messungen oder durch Modellierung der Hardware und anschließende Simulation bestimmt werden. Da diese Ansätze oft nur für kleine Programmteile konzipiert sind, gibt es einen dritten Schwerpunkt in der Forschung, der sich mit der Berechnung der Ausführungszeit von umfangreichen Programmen aus den ermittelten Bearbeitungszeiten ihrer kleineren Teile beschäftigt. Die meisten dieser Verfahren sind relativ aufwendig und meist nur für eine spezielle Architektur geeignet.

Um im Kontext dieser Arbeit aussagekräftige Bearbeitungszeiten für die weitere Analyse zu erhalten, wurde im Rahmen einer Diplomarbeit [77] die notwendige Werkzeugunterstützung

(Blockbibliothek) für eine Co-Simulation realisiert. Die Messungen werden am realen Zielsystem durchgeführt, indem die echtzeitfähige Software-Architektur (die in dieser Arbeit entwickelt wurde) um eine Simulations-Manager-Task erweitert wird, welche eine Kommunikation zwischen dem Target und einer PC-Simulation des einbettenden Systems bereitstellt. Die Software auf dem Zielsystem wird somit gezielt mit realistisch simulierten Messdaten vom PC-Modell versorgt und gewünschte Abarbeitungsschritte werden durchführen. Nach jedem Berechnungsschritt findet wiederum ein Update der Daten auf dem PC statt, um auch dort die Auswirkungen der Regleralgorithmen, die auf dem Target laufen, zu berücksichtigen.

Damit ist der Wirkungskreis geschlossen und eine Co-Simulation für bestimmte Abläufe und Situationen möglich. Da die einzelnen Ausführungsschritte ohne Unterbrechung durchgeführt werden, können realistische Ausführungszeiten der unterschiedlichen modellabhängigen Aktionen des Systems gemessen werden.

Bei einer solchen Co-Simulation wird auch zum ersten Mal das reale Laufzeitsystem (RTOS) zur Implementierung der modellierten Funktionalität verwendet, so können unerwünschte Unterschiede zwischen dem Simulations- und dem Implementierungs-Verhalten bei der Analyse der Ergebnisse erkannt werden.

In Abbildung 5.11 ist das Zusammenspiel zwischen PC und Target während der beschriebenen Co-Simulation noch einmal schematisch dargestellt.

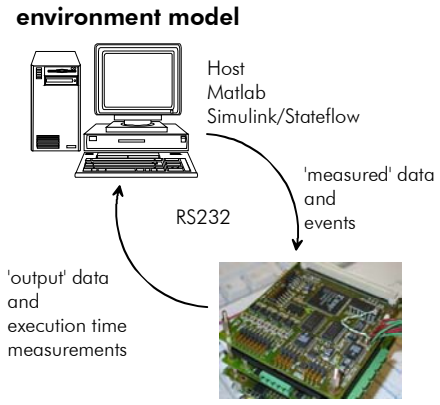

Abbildung 5.11: Schematische Darstellung der Co-Simulation zur Ausführungszeitmessung

Im beschriebenen Verfahren werden *mögliche* Ausführungszeiten der SW-Architektur gemessen und es wird nicht auf die Erzeugung von worst-case Szenarien eingegangen. Die Wahl dieses pragmatischen Ansatzes wird dadurch begründet, dass der Fokus dieser Arbeit nicht in der Gewinnung des worst-case Falles der einzelnen Ausführungszeiten bestand, sondern vielmehr in der Entwicklung der übergeordneten Vorgehensweise (SW-Architektur und RT-Nachweisverfahren für hybride graphische Spezifikationen) lag.

Kontext-Wechsel-Zeiten, die bei der Co-Simulation unberücksichtigt bleiben, werden durch gezielte Zuschläge zu den gemessenen Werten berücksichtigt. Ähnlich werden auch Treiber-Ausführungszeiten mit in die Analyse aufgenommen, da durch die Verwendung von Pseudo-Treiber-Blöcken zur Kommunikation mit dem PC bei der Simulation, diese ebenfalls nicht mitgemessen werden (siehe Diplomarbeit [77]).

Bei einem weiterführenden Einsatz der hier vorgestellten Methodik kann die Gewinnung von „realistischen" Ausführungszeiten durch eine genauere Variante zur Ermittlung garantierter wost-case Zeiten, wie z.B. in [121][122], ersetzt werden.

Parameter aus dem graphischen Modell

Aus dem graphischen Modell kann in Schritt (3) aus Abbildung 5.10 der *allgemeine APG* des Modells abgeleitet werden. Die dafür notwendigen allgemeinen Algorithmen wurden bereits in den Kapiteln 4.1.3 beschrieben.

Bei der hier realisierten Umsetzung für die Werkzeugkette Simulink/Stateflow wurde im ersten Ansatz die automatisierte Extraktion von *Aktions-Präzedenz-Graphen*, auf die Gewinnung *konkreter APG* beschränkt. Der Entwickler kann lediglich UND Bedingungsknoten zur Spezifikation der Kommunikationsbeziehungen zwischen Aktionen eines *APG* nutzen. Prinzipiell ist eine Erweiterung auf die Gewinnung der *allgemeinen APG* durchaus möglich, wurde jedoch im Rahmen dieser Arbeit nicht durchgeführt.

Zur automatisierten Extraktion des *konkreten APG* eines Modells, wurde das Skript[10] `parse_for_APG.m` (m-File) implementiert (siehe Anhang B).

Beim ereignisgesteuerten Anteil eines gegebenen Modells werden, ausgehend von den externen *Message Send Blöcken*, die durch ein spezielles Flag gekennzeichnet sind, alle internen Kommunikationsbeziehungen und Aktivitäten automatisch durchlaufen.

Dabei wird für jede ereignisgetriggerte Aktion im System eine eigene Datenstruktur `action_gl(i)` angelegt und mit folgenden konkreten Daten gefüllt:

- Action-Identifier, entspricht dem triggernden Message-Identifier *msgID*
- Priorität der Aktion, entspricht der triggernden Message-Priorität
- Ausführungszeit, die durch eine Co-Simulation ermittelt wurde
- Task-Identifier, entspricht dem Identifier des bearbeitenden Threads
- Namen der Trigger-Nachricht (zum Debuggen)
- Vektor von Action-Identifiern aller Nachfolge-Aktionen.

Die Nachfolger einer bestimmten Aktion sind dabei von der umgesetzten Applikations-Funktion abhängig und werden während des Parse-Vorgangs durch Interaktion mit dem Entwickler[11] definiert. Durch diese Informationen wird das Applikationswissen des ereignisgesteuerten Teils mit in die Analyse aufgenommen bzw. beim Aufbau des APG berücksichtigt. Damit die Daten für ein bestimmtes Modell nicht immer von neuem eingegeben werden müssen, wird im Hintergrund jede Information in eine Log-Datei (`<MODEL_NAME>_log.m`) geschrieben, die dann beim erneuten Aufruf des Parse-Skripts entweder ausgelesen oder überschrieben werden kann.

Nachdem so alle internen Kommunikationsbeziehungen und Aktivitäten definiert sind, werden die externen Aktionen des ereignisgetriggerten Modellanteils spezifiziert. Hierzu wird für jedes ISR-SubSystem, in dem sich ein externer *Message Send Block,* d.h. mit einem aktivierten External-Flag befindet, eine neue zusätzliche Aktivität definiert. Dadurch werden die Rechenzeitanforderungen, die durch die ISRs entstehen, mit in der RT-Analyse berücksichtigt. Die Priorität dieser Aktionen ergibt sich durch die bereits beschriebene Abbildung von HW-Prioritäten der IRQs auf die einheitlichen SW-Prioritäten der Aktionen im System.

Zur automatisierten Gewinnung aller *konkreten APG* des zyklischen Modellanteils ist im Gegensatz zum ereignisgesteuerten Anteil keine Interaktion mit dem Entwickler notwendig. Durch die in Kapitel 4.1.3 beschriebene Transformation ergeben sich die eindeutigen Strukturen der unterschiedlichen *konkreten APG* für die im System enthaltenen Abtastraten. Somit können

---

[10] Der entsprechende Funktionsaufruf in der Matlab-Shell ist:
`>[action_gl,action_chain_gl,Ts,f_name] = parse_for_APG('<MODEL_NAME>');`

[11] Dies geschieht in Form von gezielten Fragen, wie z.B.:
„`Can message: MsgGen1 trigger message: MsgGen3 ?[y, n, c]`“.

auch für diesen Teil des graphischen Modells alle Aktionen mit den dazugehörigen Datenstrukturen angelegt und gefüllt werden.

Zur Visualisierung der gewonnenen *konkreten APG* wurde das Skript `build_APG.m` verfasst[12], das die Daten der aus dem Parse-Vorgang gewonnenen Aktionen in ein für das Werkzeug: *daVinci Presenter* [13] lesbares Textformat konvertiert (siehe Anhang B). So können die in den Datenstrukturen enthaltenen Informationen für den Designer als Graphen dargestellt werden.

Parameter aus der Spezifikation

Die letzten, für die RT-Analyse relevanten Parameter, werden aus der Spezifikation des zu entwickelnden Systems gewonnen. Hierzu zählen zum einen die Rechenzeitanforderungen aus dem einbettenden System, die durch die in Kapitel 4.1.4 beschriebenen Ereignisströme definiert werden und zum anderen die einzuhaltenden Deadlines für bestimmte Reaktionen des Systems auf die Anforderungen durch die Umgebung. Letztere werden erst nach der Berechnung der worst-case Reaktionszeiten zur Bewertung der realisierten Implementierung verwendet.

Die Ereignisströme für den zyklischen Modellanteil können, wie bereits in Abschnitt 4.1.4 angedeutet nicht nur aus den Requirements abgeleitet, sondern auch automatisiert aus der graphischen Modellspezifikation extrahiert werden.

Im zyklischen Modellanteil sind nämlich die Rechenzeitanforderungen bereits implizit in Form von Abtastraten definiert und werden bei der automatisierten Abbildung auf den SW-Frame berücksichtigt. Durch das zyklische Laufzeitmodell (Kaptiel 3.1.10) werden auch die einzuhaltenden Deadlines festgelegt (Deadline ist gleich der Periodendauer).

Beim ereignisgesteuerten Anteil müssen die möglichen worst-case Ereignisströme durch die Analyse des physikalischen einbettenden Systems und der beschriebenen Requirements abgeleitet werden (siehe hierzu [117]). Selbes gilt für die Deadlines, falls sie nicht bereits Teil der Anforderungen sind.

5.2.2 Implementierung des RT-Nachweises

Zur Umsetzung des Realzeitnachweises eines graphischen Modells sind verschiedene Schritte notwendig. Zum einen muss zur Berechnung des jeweiligen worst-case Zeitpunktes, sei es der Startzeitpunk oder der Endzeitpunkt, aus den *allgemeinen APG* die *konkreten APG* abgeleitet werden. Hierzu wird je nach untersuchter Aktion und verwendetem Prioritäts-Vererbungs-Protokoll, bei allen unterschiedlichen ODER Bedingungspfaden der *allgemeinen APG* einer graphischen Spezifikation, immer nur der eine Pfad mit der größten kumulativen Ausführungszeit für die weitere Untersuchung im *konkreten APG* verwendet (siehe Kaptiel 4.1.3).

Zum anderen muss für die jeweilig betrachtete Aktion die maximale Blockierzeit bei der gegebenen SW-Architektur und dem verwendeten Prioritätsvererbungs-Protokoll berechnet werden.

Anschließend kann dann mit der Berechnung des worst-case Startzeitpunktes für die Bearbeitung der betrachteten Aktion bei der gegebenen worst-case Belastung durch das einbettende System und den gewonnenen *konkreten APG* begonnen werden.

Anschließend wird aufbauend auf das erhaltene Ergebnis, der worst-case Endzeitpunkt berechnet, um die unterschiedlichen Reaktionszeiten zu bestimmen.

---

[12] Der entsprechende Funktionsaufruf in der Matlab-Shell ist:
```
>build_APG(action_chain_gl, action_gl, Ts, f_name);
```

[13] *daVinci Presenter* ist ein von *b-novative GmbH* entwickeltes Werkzeug zur Visualisierung von Graphen. http://www.b-novative.de/home.html

Die Abbildung der *allgemeinen APG* auf die jeweiligen *konkreten APG* entfällt in der vorhandenen Umsetzung, da die Untersuchung von ODER Bedingungspfaden nicht unterstützt wird (siehe Kaptiel 5.2.1).

Somit ergeben sich aus der Parametergewinnungs-Phase direkt die *konkreten APG* der graphischen Spezifikation. Alle Aktionen im System werden durch die in Kapitel 5.2.1 beschriebene Datenstruktur für die weitere automatisierte Verarbeitung abgelegt.

Jeder *Aktions-Präzedenz-Graph* wird zudem durch die Angabe

- seiner ersten Aktion und
- dem dazugehörigen Ereignisstrom

definiert. Damit sind alle für die automatisierte Analyse notwendigen Informationen verfügbar.

Die in Kapitel 4 beschriebenen Formeln für die Startzeitpunkte und Endzeitpunkte sind in einer impliziten Form gegeben und bei der Berechnung der Endzeitpunkte ist die Bestimmung der kumulativen Ausführungszeit $L_j(A_l^\tau)$ als rekursive Summe über den jeweiligen *konkreten Aktions-Präzedenz-Graphen* angegeben.

Zur Implementierung der beschriebenen impliziten mathematischen Zusammenhänge wurde der bereits durch Joseph und Pandya [97] definierte iterative Ansatz angewandt. Wie bereits in Kapitel 4.2 erwähnt, konvergiert die iterative Umsetzung für alle Fälle mit einer Auslastung kleiner als 1.

Dabei wird ausgehend von einem Startwert V_0, jeweils ein neues Ergebnis V_1 der impliziten Formel berechnet. Iterativ wird dann durch Einsetzen des alten Ergebnisses V_{n-1} und der Berechnung des impliziten mathematischen Zusammenhangs ein neues Ergebnis V_n berechnet.

Die Abbruchbedingung dieser allgemeinen iterativen Berechnung ist im Erfolgsfall die Übereinstimmung zwischen zwei unterschiedlichen Iterationsergebnissen $V_m == V_{m-1}$ und im Falle einer Überlastung, also bei einer Auslastung größer 1, das Erreichen des Wertes Unendlich $V_m == \infty$.

Die iterative Umsetzung und der jeweils gewählte Anfangswert werden nachfolgend kurz zur Vollständigkeit angegeben. Die jeweils vollständigen Versionen der verwendeten Algorithmen sind im Anhang C als Skripten mit den Namen

- `responsetime_BPI.m`[14] für das *Basic Priority Inheritence* Protokoll und
- `responsetime_PT.m`[15] für das *Preemption Threshold* Protokoll

wiedergegeben.

Nachfolgende iterative Formeln sind unabhängig von den angewandten Prioritätsvererbungs-Strategien und wurden bereits in Kapitel 4.2 beschrieben.

[14] `>responsetime_BPI(ES,Ts,action_chain_gl,action_gl,TransactNr,ActNr);`

[15] `>responsetime_PT(ES,Ts,action_chain_gl,action_gl,TransactNr,ActNr);`

Startzeitpunkte

Initialwert:

$$S_i^\tau(q)_0 = B(A_i) \qquad \textbf{(5.1)}$$

Iterative Implementierung der Startzeitpunkt Berechnung:

$$S_i^\tau(q)_n = B(A_i) + \qquad \textbf{(5.2)}$$

$$+\sum_{k\neq\tau}\left[\Psi_k^+(S_i^\tau(q)_{n-1})\cdot\sum_l\left(C(A_l^k)\mid\pi(A_l^k)\geq\pi(A_i^\tau)\right)\right]+ \qquad \textbf{(5.3)}$$

$$+(q-1)\cdot\sum_l\left(C(A_l^\tau)\mid\pi(A_l^\tau)\geq\pi(A_i^\tau)\right)+ \qquad \textbf{(5.4)}$$

$$+\left(\Psi_\tau^+(S_i^\tau(q)_{n-1})-q+1\right)\cdot\sum_l\left(C(A_l^\tau)\mid\neg\left(A_i^\tau\rightarrow A_l^\tau\right)\wedge\pi(A_l^\tau)\geq\pi(A_i^\tau)\right) \qquad \textbf{(5.5)}$$

Endzeitpunkte

Initialwert:

$$W_0 = S_i^\tau(q)_{final} \qquad \textbf{(5.6)}$$

Iterative Implementierung der Endzeitpunkt Berechnung:

$$F_i^\tau(q) = \min(W) \quad with$$

$$W_n = S_i^\tau(q)_{final} + \qquad \textbf{(5.7)}$$

$$+\sum_{k\neq\tau}\left(\Psi_k(W_{n-1})-\Psi_k^+(S_i^\tau(q)_{final})\right)\cdot L_k(A_i^\tau)+ \qquad \textbf{(5.8)}$$

$$+\left(\Psi_i(W_{n-1})-\Psi_i^+(S_i^\tau(q)_{final})\right)\cdot L_\tau(A_i^\tau) \qquad \textbf{(5.9)}$$

Die Berechnung des kumulativen Berechnungsaufwandes $L_j(A_l^\tau)$ wird dabei als rekursive Funktion implementiert.

5.2.3 Konkretes Anwendungs-Beispiel

Die konkrete Vorgehensweise wird anhand eines kleinen Beispiels dargestellt. Hierzu wurde ein einfaches Modell gewählt, das sowohl zyklische als auch ereignisgesteuerte Modellteile beinhaltet. Die funktionale Spezifikation entspricht dabei keiner realen Anwendung, sondern erzeugt lediglich durch Vektor-Multiplikationen eine gut reproduzierbare Rechnerauslastung.

Das hier untersuchte Simulations-Modell ist in Abbildung 5.12 dargestellt. Die Funktionalität des zu entwickelnden Systems (*controller model*) ist aus zwei Abtastraten (T_{s1} = 0.01sec, T_{s2} = 0.1 sec) und einem extern getriggerten ereignisgesteuerten Anteil zusammengesetzt. Dieser wird durch den IRQ2 angestoßen und stellt eine einfache Verkettung (siehe Kapitel 4.1.3) dar. Der Rest des Modells bildet auf sehr einfache Weise den einbettenden Prozess (*embedding system*) nach.

Diese zwei Teile werden zur Parametergewinnung bei der beschriebenen Co-Simulation auf zwei getrennte Modelle aufgeteilt (entsprechend Abbildung 5.10 *(1)*).

In Abbildung 5.13 ist der Teil des eigentlich zu entwickelnden Systems in einem eigenen Modell (Target-Modell) wiedergegeben.

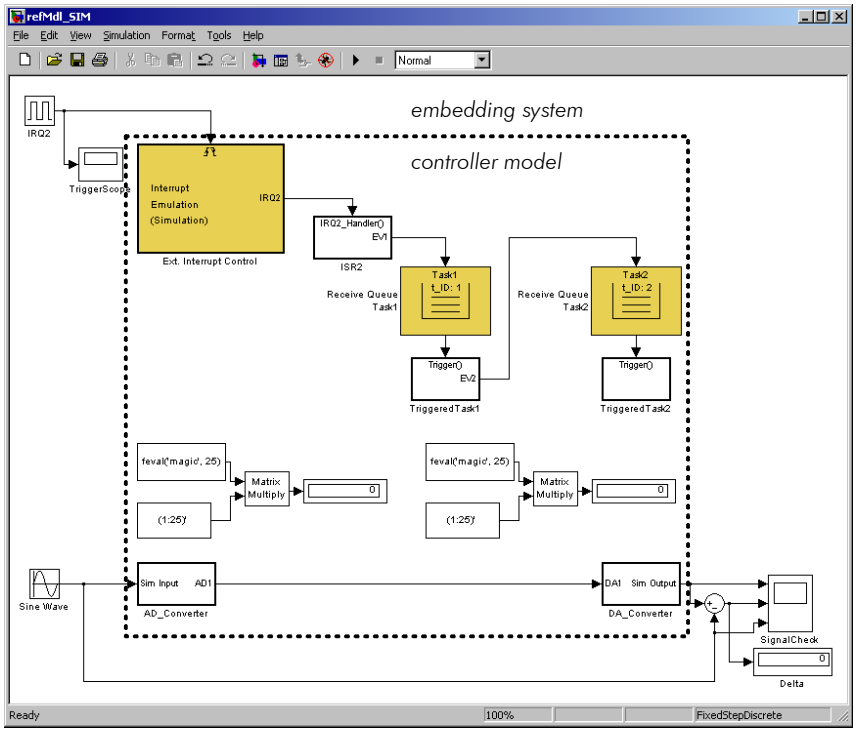

Abbildung 5.12: Simulations-Modell des hier umgesetzten Beispiels, beinhaltet sowohl Controller-Modell als auch Teile des einbettenden Systems

Um während der Co-Simulation die Anbindung der Signale beider Modelle zu ermöglichen, werden spezielle Blöcke anstelle der entgültigen Hardware-Treiberblöcke verwendet. Auch die Interrupt-Anbindung wird durch einen für die Co-Simulation geeigneten Block ersetzt.

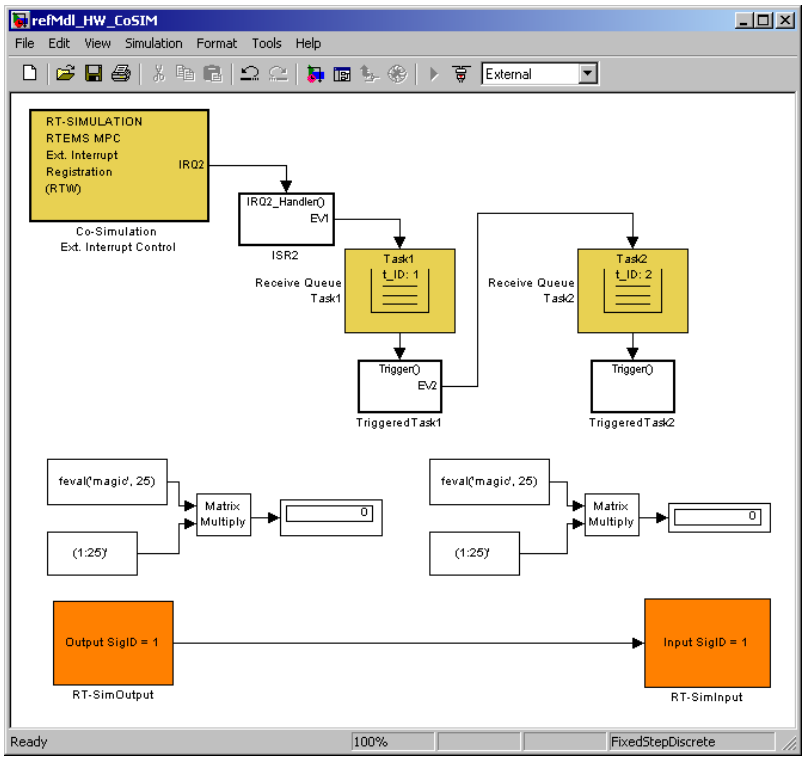

Abbildung 5.13: Controller-Modell mit Anpassungen für die Co–Simulation (Target-Modell)

Nun kann die automatische Codegenerierung für das RT-Simulations-Target gestartet werden. Das dabei gebildete ausführbare Programm besitzt die in Abbildung 5.14 dargestellte SW-Architektur. Durch die *SimManager* Task wird die Co-Simulation mit dem PC-Modell realisiert. Das erzeugte Programm wird auf das Zielsystem (hier ein MPC555 Evaluation-Board) geladen und gestartet.

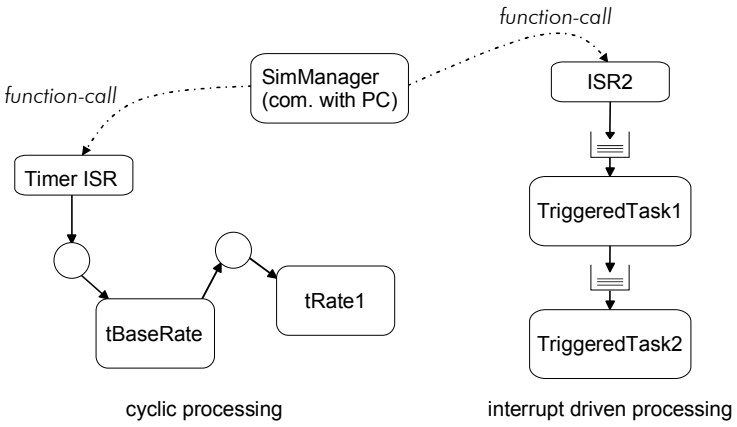

Abbildung 5.14: Software-Architektur des untersuchten Modells bei der Co-Simulation

Die *SimManager* Task wartet nun auf den Beginn der Co-Simulation durch das PC-Modell, das in Abbildung 5.15 dargestellt ist und bildet das einbettende System nach.

Durch das Starten eines Simulations-Laufes dieses Modells in Simulink wird die Co-Simulation angestossen. Vor jedem Ausführungsschritt der Target-Software übermittelt das PC-Modell der Simulink-Simulation über eine Serielle-Kommunikation die jeweiligen neuen Messdaten (z.B. emulierte A/D-Wandler-Werte oder digitale Eingangssignale) und die für diesen Ausführungs-schritt notwendigen Ereignisse (z.B. PIT oder IRQx).

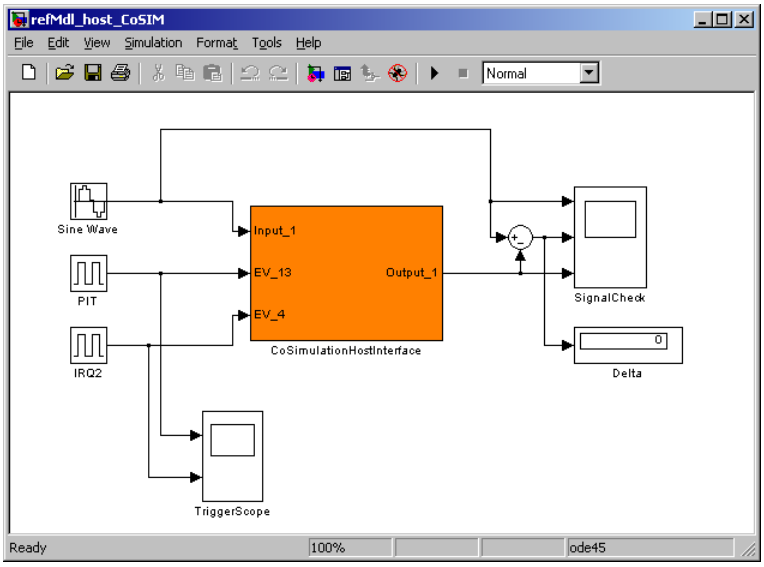

Abbildung 5.15: Simulations-Modell der einbettenden Umgebung auf dem PC (PC-Modell)

Das Target bearbeitet daraufhin die neuen Informationen und sendet die Ergebnisse mit den jeweiligen gemessenen Ausführungszeiten zurück zum PC-Modell, das dann wiederum seinen Simulationsschritt beenden kann.

Im hier vorgestellten Beispiel sind genau zwei unterschiedliche Ereignis-Quellen enthalten, der periodische Interrupt des PIT mit dem Identifier *13* und der externe Hardware-Interrupt IRQ2 mit dem Identifier *4*.

Am Ende eines erfolgreich durchgeführten Simulations-Laufes können nicht nur die Reaktionen des Target-Systems auf gegebene Stimuli durch das PC-Modell evaluiert, sondern auch die on-line gemessenen, realen Ausführungszeiten für die relevanten Aktionen der graphischen Umsetzung ausgewertet werden. Diese Daten werden während der laufenden Simulation in eine ASCII-Datei (`<MODEL>_exeTime.m`) geschrieben, die anschließend automatisiert ausgewertet werden kann.

Dies geschieht mit dem Skript: `parse_for_APG.m`. Hier werden die gemessenen Zeiten eingelesen und um die jeweils notwendigen, modellunabhängigen Betriebsystemzeiten erweitert.

Das Skript parst das gegebene Modell, um die *konkreten APG* für die anschließende Analyse zu ermitteln. Dabei wird durch Interaktion mit dem Entwickler Applikationswissen in die Analyse bzw. den *APG* aufgenommen. Letztendlich gibt das Skript alle ermittelten Aktionen des Systems mit den aus dem Modell und der Co-Simulation gewonnenen Parametern zurück.

```
>[action_gl,action_chain_gl,Ts,f_name] = parse_for_APG('refModel');
```

Die gewonnenen konkreten Daten sind dann als Datenstruktur im Workspace von Matlab enthalten, können aber vom Entwickler nur mühsam interpretiert werden. Zur Vereinfachung der Analyse wurde ein weiteres Skript implementiert, das die Darstellung der Daten in einer graphischen Form erlaubt (`build_APG.m`). Das Skript stellt die gegebenen Daten in unterschiedlichen Graphen dar. Das Ergebnis für das hier beschriebene Beispiel ist in Abbildung 5.16 dargestellt.

```
>build_APG(action_chain_gl,action_gl,Ts,f_name);
```

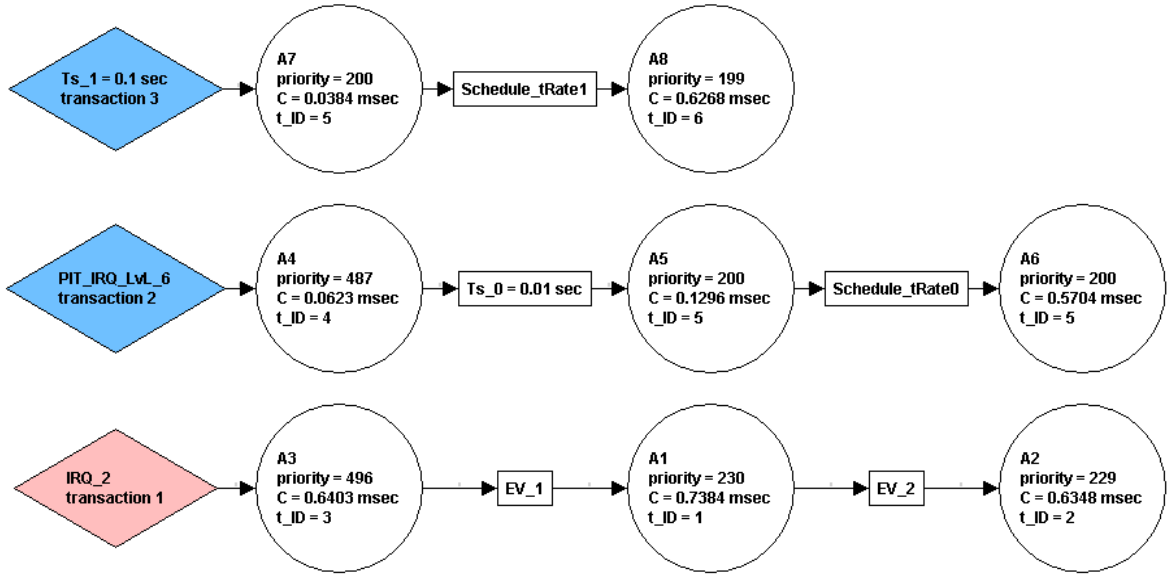

Abbildung 5.16: *Konkrete APG* des Beispiel-Modells mit realen Parametern aus der graphischen Spezifikation und der Co-Simulation

Nun kann gezielt die Realzeit-Analyse des Modells durchgeführt werden. Dazu werden für die unterschiedlichen Aktionen der *konkreten APG* die worst-case Reaktionszeiten berechnet.

Dabei wird das Skript `responsetime_BPI.m` (bzw. `responsetime_PT.m`) wie folgt verwendet.

```
>responsetime_BPI (ES, Ts, action_chain_gl, action_gl, T_Nr, A_Nr);
```

Die zwei ersten übergebenen Parameter ES und Ts spiegeln die Belastung durch das einbettende System wieder. ES gibt die Ereignisströme der externen Ereignisse wieder (hier nur $IRQ2$), wogegen Ts die Abtastraten des zyklischen Anteils darstellen.

Die letzten zwei skalaren Parameter *T_Nr* und *A_Nr* geben die untersuchte Transaktion und die darin betrachtete Aktion wieder.

Wird z.B. die worst-case Reaktionszeit der Aktion *A6* in der Transaktion *2* untersucht, wird folgender Aufruf getätigt:

```
>responsetime_BPI (ES, Ts, action_chain_gl, action_gl, 2, 6);
```

Um die Genauigkeit der im Rahmen dieser Arbeit entwickelten RT-Analyse und ihre Umsetzung abzuschätzen, wurde für das hier beschriebene Beispiel die maximale Interrupt-Frequenz *IRQ2* berechnet, für die alle zyklischen Deadlines (d.h. die Periodendauern) eingehalten werden.

Anschließend wurde das in Abbildung 5.17 dargestellte Modell, das dem RT-Modell des untersuchten Beispiels (Abbildung 5.13) entspricht, automatisch auf das Zielsystem abgebildet. Das resultierende System wurde anschließend durch einen entsprechenden Hardware-Aufbau belastet, indem die Interrupt-Frequenz *IRQ2* mittels eines Frequenzgenerators beliebig eingestellt werden konnte.

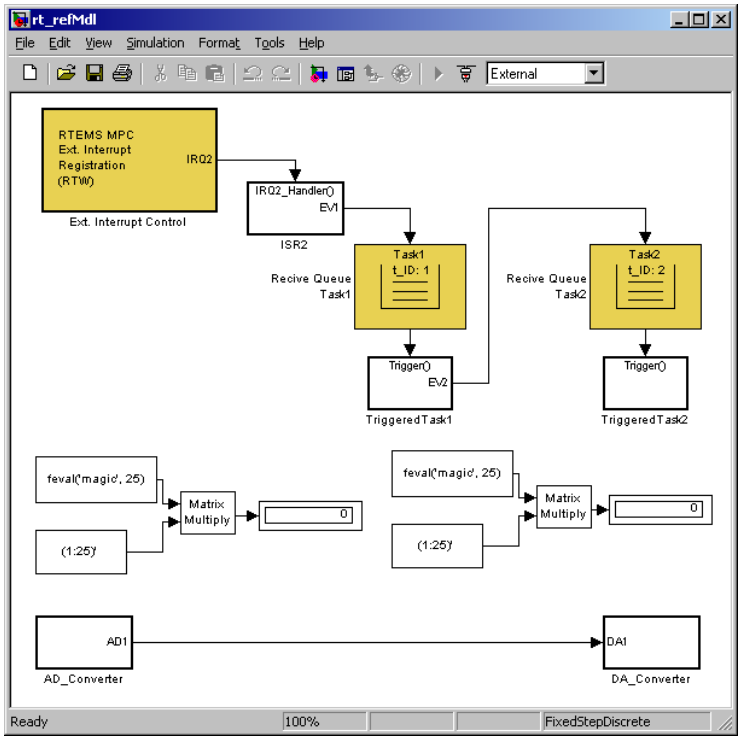

Abbildung 5.17: Realzeit-Modell des beschriebenen Beispiels mit realen Treiber-Blöcken

Die Ergebnisse der Realzeit-Analyse für unterschiedliche periodische Interrupt-Frequenzen sind in Tabelle 5.1 dargestellt.

Dabei wird ersichtlich, dass bei einer Interrupt-Frequenz von 451Hz die Reaktionszeit von Aktion *A6* noch unter der Periodendauer von T_{s1} = 0.01sec liegt, bei einer Frequenz von 452Hz jedoch bereits zwei Instanzen (q = 1 und 2) der Aktion *A6* im selben Busy-Window bearbeitet werden. Dies weist auf eine Deadline-Verletzung der höchsten Abtastrate des zyklischen Modellanteils hin.

Bei der Validierung der berechneten Ergebnisse wurde die RT-Implementierung des Modells durch einen Frequenzgenerator mit der jeweilig gewünschten Interrupt-Wiederholungsfrequenz belastet. Dabei wurde bewiesen, dass die berechnete obere Grenze von 451 Hz die SW-Architektur nicht überlastet. Durch schrittweise Erhöhung der Interrupt-Frequenz wurde auch eine Abschätzung für die reale obere Grenzfrequenz ermittelt, die zwischen 463 und 464Hz

liegt. Die somit bestimmbare Abweichung zwischen realer und berechneter worst-case Interrupt-Belastung liegt bei ca. 13Hz oder 2.9%.

Tabelle 5.1: Berechnete Reaktionszeiten für das gegebene konkrete Beispiel bei einer zyklischen Interrupt-Belastung

IRQ2 Ereignisstrom *ES*	**Reaktionszeiten von Aktion 6 in Transaktion 2 [sec]**	**Reaktionszeiten von Aktion 8 in Transaktion 3 [sec]**
$\left\{\begin{pmatrix} 1/450Hz \\ 0 \end{pmatrix}\right\}$	0.0088547	0.03996
$\left\{\begin{pmatrix} 1/451Hz \\ 0 \end{pmatrix}\right\}$	0.0088547	0.04877
$\left\{\begin{pmatrix} 1/452Hz \\ 0 \end{pmatrix}\right\}$	0.01093 für q = 1 0.00767 für q = 2	0.05960

6 Realisierung der Laufzeitsystemunterstützung

6.1 Anforderungen und Wahl eines RTOS

Wie bereits im Abschnitt 3.3 beschrieben ist für die Umsetzung der hier vorgestellten SW-Architektur grundsätzlich ein Realzeitbetriebsystem (RTOS) mit den folgenden Grundfunktionalitäten erforderlich:

- Preemptives Multi-Tasking
- Prioritätsbasiertes Scheduling mit FIFO Verhalten bei Threads mit gleichen Prioritäten
- Möglichkeit der Veränderung von Prioritäten bei Threads während der Laufzeit
- Möglichkeit der Interrupt-Anbindung in Form von Interrupt Service Routinen (ISRs)
- Versenden von Messages mit Prioritäten
- Messagequeues mit Berücksichtigung der Message-Prioritäten beim Sortieren.

Diese Funktionen sind in vielen am Markt erhältlichen Realzeitbetriebsystemen bereits enthalten. In den letzten Jahren hat eine zunehmende Vereinheitlichung des Funktionsumfanges unterschiedlicher Implementierungen, besonders durch die Standardisierung von Programmierungs-Schnittstellen (*Application Programming Interface*, API), stattgefunden. Erwähnenswert ist in diesem Kontext die Verbreitung des POSIX 4 Standards [123][124], der bereits von vielen Realzeitbetriebsystemen unterstützt wird.

Eine weitere für die Umsetzung der vorgestellten neuen Konzepte unerlässliche Funktionalität ist die Bereitstellung von

- ❖ Prioritätsvererbungs-Protokollen zwischen den Messages in den Queues und den bearbeitenden Threads.

Solche Prioritätsvererbungs-Strategien werden jedoch nur zum Teil und von vereinzelten kommerziellen Produkten (z.B. QNX [123] Seite 103) unterstützt. Um genau diese Protokolle im unterlagerten Laufzeitsystem zur Verfügung zu stellen, waren Anpassungen in den Kernel-Funktionen der Message-Queue-Verwaltung notwendig, die nachfolgend genauer erläutert werden.

Bei der Wahl eines geeigneten Ausgangspunkts (RTOS) für die notwendigen Erweiterungen, wurde versucht, auf einem sich etablierenden Standard aufzubauen. Aus diesem Grund kamen für die Auswahl nur Realzeitbetriebsysteme mit einer POSIX 4 bzw. POSIX 1003.1 konformen API in Frage, um einen möglichen zukünftigen Portierungsaufwand zu minimieren.

Als weitere Anforderung stand fest, dass das gesamte Betriebsystem in Form von Quell-Code verfügbar sein muss, um die notwendigen Erweiterungen auf Source-Code-Ebene einfügen zu können. Dies führte unweigerlich zum Einsatz eines Open-Source RTOS.

Durch diese zwei Randbedingungen

- POSIX 1003.1 API und
- Verfügbarkeit der Kernel-Sourcen

wurde für die Umsetzung des Laufzeitsystems das Realzeitbetriebsystem RTEMS[16] gewählt.

[16] www.OARcorp.com oder www.RTEMS.com

RTEMS [125][126] steht für *Real-Time Executive for Multiprocessor Systems* und ist ein typisches Realzeitbetriebsystem für eingebettete Anwendungen. Es ist in Form von Quellcode frei verfügbar und kann für viele Prozessorarchitekturen eingesetzt werden. Ursprünglich wurde es für das amerikanische Militär entwickelt, heute wird es jedoch auch in zahlreichen zivilen Projekten eingesetzt. Es bietet eine, für die hier notwendige Implementierung ausreichende POSIX 1003.1 Unterstützung [127] mit Message-Prioritäten und sortierten Warteschlangen und erlaubt eine einfache Erweiterung der für die angesprochene Prioritätsvererbung notwendigen Betriebsystem-Funktionen.

6.2 Notwendige Anpassungen und Implementierung

Grundsätzlich ergibt sich bei der Implementierung der erwähnten Prioritätsvererbungs-Protokolle eine begrenzte Anzahl von Kernel-Funktionen, die verändert bzw. erweitert werden müssen (siehe Tabelle 6.1 und Tabelle 6.2).

Im Rahmen dieser Arbeit wurden die konkreten Erweiterungen für das *Basic-Priority-Inheritence* Protokoll und das *Preemption-Threshold* Protokoll zur Vererbung der Message-Priorität an den Server-Thread in den Sourcen umgesetzt.

6.2.1 Basic Priority Inheritence (BPI) Protokoll

Die Funktionsweise des *Basic-Priority-Inheritence* Protokoll wurde bereits in Kapitel 3.2.2 beschrieben.

Zur Implementierung des BPI-Protokolls zwischen Message-Prioritäten und der Priorität des Server-Threads wurde als Vorlage die bereits im verwendeten RTOS (RTEMS 4.5.0) vorhandene Realisierung des BPI-Verhaltens bei binären Semaphoren bzw. Mutexes verwendet. Nachfolgend werden die veränderten Datenstrukturen und Funktionen kurz beschrieben.

Die Datenstruktur der Message-Queue `CORE_message_queue_Control` in der Datei `coremsg.h` wurde erweitert, um eine eindeutige Zuordnung des Server-Threads zur Queue zu ermöglichen. Durch diese Referenz wird es erst möglich, den jeweiligen Thread, der die Message-Queue bedient, zu referenzieren und gegebenenfalls dessen Priorität zu verändern.

Zur richtigen Initialisierung dieser neuen Datenfelder wurde die neue Funktion `mq_initialize_holder()` in der Datei `mqueueopen.c` implementiert. Diese Funktion wird im jeweiligen Server-Thread während der Initialisierung einmalig aufgerufen und konfiguriert dabei die Datenstruktur mit den eigenen Referenzen.

Eine Veränderung der Server-Thread-Priorität kann beim Eintragen einer neuen Nachricht in die Queue durch einen Sender-Thread oder beim Entnehmen einer Nachricht aus der Queue durch den Server-Thread erforderlich werden. Zu diesem Zweck wurden die nachfolgend beschriebenen Funktionen um geeignete Code-Fragmente erweitert.

Senden einer neuen Nachricht an eine Message-Queue

Beim Senden einer neuen Nachricht durch den RTOS Befehl `mq_send()` wird als erstes die Möglichkeit einer korrekten Abbildung der Message-Priorität auf eine Thread-Priorität überprüft. Dies wird in der Funktion `_POSIX_Message_queue_Send_support()` realisiert, gültige POSIX Thread-Prioritäten bei RTEMS sind [1 ... 254], wobei höhere Zahlen eine höhere Priorität wiedergeben.

Handelt es sich um eine gültige Nachricht, wird sie mit dem Befehl `_CORE_message_queue_Submit()` weiterverarbeitet. Abhängig davon, ob an der Empfangs-Queue bereits ein Thread wartet oder nicht, wird die Nachricht sofort dem Empfangs-

Thread übergeben, oder die Nachricht mit der Funktion `_CORE_message_queue_Insert_message()` in die Empfangswarteschlange eingefügt.

Beim direkten Übergeben in `_CORE_message_queue_Submit()` wird die Priorität der Message *immer* an den Empfangs-Threads vererbt, da dieser bereits auf eine Nachricht wartet und daher blockiert ist.

Wird die Nachricht aber durch `_CORE_message_queue_Insert_message()` in die Warteschlange eingereiht, findet dabei eine Sortierung nach Prioritäten statt. Für den Fall, dass die neue Nachricht an die erste Stelle der Queue eingereiht wird, sie somit die höchste Priorität unter den wartenden Nachrichten besitzt, ist möglicherweise eine Vererbung der Priorität an den Server-Thread notwendig. Ist die neue Nachrichten-Priorität größer als die bereits gehaltene Priorität des Server-Threads, wird durch die Vererbung letztere angehoben.

Empfangen einer neuen Nachricht durch den Server-Thread

Beim Empfangen einer Nachricht im Server-Thread durch den Befehl `mq_receive()` können zwei Fälle auftreten. In dem Fall, dass keine Nachricht in der Warteschlange enthalten ist wird der Thread blockiert und gibt die Kontrolle über den Prozessor ab. Wird durch einen anderen Thread im System eine neue Nachricht an die leere Queue gesendet, wird, wie bereits erwähnt, direkt die Nachricht an den Server-Thread übergeben und seine Priorität an die neue Nachrichten-Priorität angepasst.

Entnimmt jedoch ein Thread durch `mq_receive()` eine wartende Nachricht aus der Warteschlange, wird die Kernel-Funktion `_CORE_message_queue_Seize()` aufgerufen, welche ebenfalls *immer* die Priorität des Server-Threads an die neue Message-Priorität anpasst.

In Tabelle 6.1 sind die unterschiedlichen Datenstrukturen und Funktionen, die angepasst werden mussten, noch einmal aufgelistet (siehe Anhang D.1).

Tabelle 6.1: Geänderte Datenstrukturen und Funktionen des RTOS (RTEMS 4.5.0) zur Implementierung des *BPI*-Protokolls zwischen Nachrichten-Prioritäten und Server-Thread-Prioritäten

Datenstruktur	`_CORE_message_queue_Control`
Bemerkungen Erweiterung der Message-Queue Datenstruktur um Queue-Holder Felder.	
Funktion	`mq_initialize_holder()`
Bemerkungen Funktion zur Initialisierung der neuen Daten-Felder der Queue-Struktur. Sie wird *einmal* im Server-Thread während der Initialisierung aufgerufen.	
Funktion	`_CORE_message_queue_Initialize_holder()`
Bemerkungen Implementiert die Kernel-Funktion, die aus `mq_initialize_holder()` für die Initialisierung der neuen Daten-Felder aufgerufen wird.	
Funktion	`_POSIX_Message_queue_Send_support()`
Bemerkungen Erweiterung um den Nachrichten-Prioritäts-Check, zur Überprüfung einer möglichen Abbildung von Message-Prioritäten auf die Thread-Prioritäten.	

Funktion	`_CORE_message_queue_Submit()`
Bemerkungen Implementiert das Senden von Nachrichten auf Kernel-Ebene.	
Funktion	`_CORE_message_queue_Insert_message()`
Bemerkungen Implementiert das Einfügen von Nachricht in Warteschlangen auf Kernel-Ebene.	
Funktion	`_CORE_message_queue_Seize()`
Bemerkungen Implementiert das Entnehmen einer Nachricht aus einer Warteschlange auf Kernel-Ebene.	

6.2.2 Preemption Threshold (PT) Protokoll

Um das in Kapitel 3.2.2 beschriebene *Preemption Threashold Protokoll* in das verwendete RTOS mit zu integrieren, musste, ähnlich dem Priority-Ceiling-Protokoll bei binären Semaphoren, die Queue-Daten-Struktur `CORE_message_queue_Control` um eine Ceiling-Priorität (`threshold_priority`) erweitert werden.

Wie beschrieben, wird diese Ceiling-Priorität durch die Konstellation des konkreten Systems definiert (z.B. höchste Message-Priorität, die von der Queue empfangen werden kann) und während der Initialisierungs-Phase im Server-Thread durch die neu implementierte Funktion `mq_initialize_threshold_priority()` konfiguriert.

Zur Laufzeit wird dann bis zum Starten des Server-Threads durch den Scheduler diese Ceiling-Priorität der Message-Queue nicht berücksichtigt, sondern lediglich die Priorität der zu bearbeitenden Nachrichten. Dieses Verhalten entspricht dem bereits umgesetzten *BPI*-Protokoll und wird auch durch dessen Erweiterungen (siehe Abschnitt 6.2.1) realisiert. Erst beim Starten des Server-Threads durch den Scheduler wird seine Priorität von der in Arbeit befindlichen Message-Priorität auf die Ceiling-Priorität angehoben.

Dies wird durch eine gezielte Erweiterung der POSIX 4 API-Funktion bewerkstelligt `_POSIX_Message_queue_Receive_support()`.

Zur Umsetzung des *PI*-Protokolls waren daher nur geringfügige Erweiterungen zum *BPI*-Protokolls notwendig. Die geänderten Datenstrukturen und Funktionen sind in Tabelle 6.2 zusammengefasst (siehe Anhang D.1).

Tabelle 6.2: Geänderte Datenstrukturen und Funktionen des RTOS (RTEMS 4.5.0) zur Implementierung des *PT*-Protokolls zwischen Nachrichten-Prioritäten und Server-Thread-Prioritäten

Datenstruktur	`_CORE_message_queue_Control`
Bemerkungen Erweiterung der Message-Queue Dataenstruktur um die Threshold-Priorität	
Funktion	`mq_initialize_threshold_priority()`
Bemerkungen Funktion zur Initialisierung der neuen Daten-Felder der Queue-Struktur	

Funktion	`_CORE_message_queue_Initialize_threshold_prio()`
Bemerkungen Implementiert die Kernel-Funktion zur Initialisierung der neuen Daten-Felder	
Funktion	`_POSIX_Message_queue_Receive_support()`
Bemerkungen Erst beim Starten des Server-Threads durch den Scheduler wird in dieser Funktion die Priorität auf die der Queue zugeordnete Threshold-Priorität gesetzt.	

7 Anwendungsbeispiele

Anhand der hier beschriebenen konkreten Anwendungsbeispiele wird die Anwendbarkeit der vorgestellten Konzepte und der realisierten Werkzeugumsetzung demonstriert.

7.1 Hochregallager (HRL)

Im folgenden Abschnitt wird die modell-basierte Entwicklung einer Hochregallager-Steuerung präsentiert. Dabei wird weniger auf den eigentlichen Steueralgorithmus eingegangen sondern vielmehr auf die Vorgehensweise bei der Entwicklung. Die korrekte Funktionsweise des entgültigen Designs wurde anhand eines realen Hochregal-Lager Modellaufbaus am Lehrstuhl für Realzeit-Computersysteme validiert. Das verwendete Fischertechnik-Modell (siehe Abbildung 7.1) wird in einer Lehrveranstaltung[17] als Versuchsaufbau eingesetzt.

Die gemessenen Ausführungszeiten der echtzeitfähigen Implementierung des graphischen Modells sind beim gewählten Zielsystem (RTEMS auf MPC555) so klein, dass in keinem Fall eine Realzeitverletzung auftreten kann. Die beschriebenen Szenarien sind jedoch insofern wichtig, da sie bei Systemen mit höherer Rechnerbelastung zu Problemen führen können und mittels der neuen Konzepte durch die hier geschilderte Vorgehensweise gelöst werden.

7.1.1 Systembeschreibung

Das zu steuernde Hochregallager verfügt über 21 Lagerplätze in drei Etagen, die mit einer Bedieneinheit angefahren werden können. Diese kann gleichzeitig sowohl horizontal (x-Achse) als auch vertikal (y-Achse) bewegt werden. Die Position wird dabei über Taster, die entlang der Bewegungsachsen angebracht sind, bestimmt.

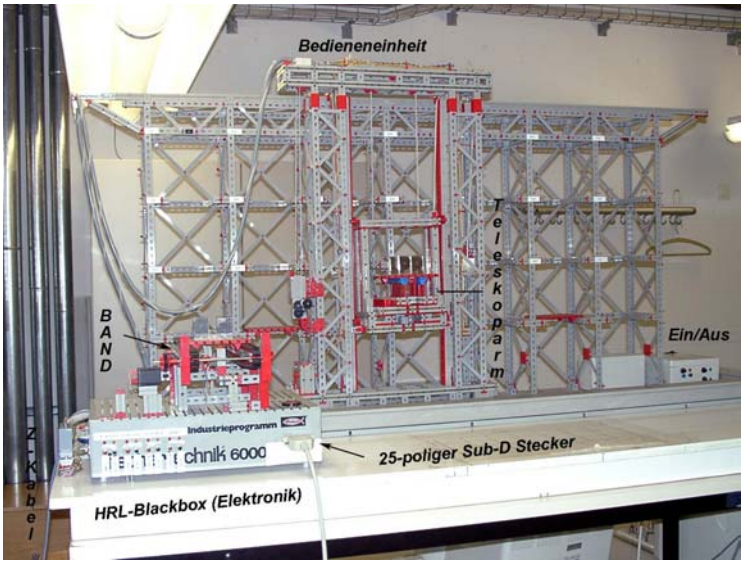

Abbildung 7.1: Fischertechnik-Anlage des Hochregallagers am RCS

[17] Praktikum Realzeitprogrammierung am Lehrstuhl für Realzeit-Computersysteme

Außerdem kann die Bedieneinheit einen Teleskoparm sowohl nach vorn als auch nach hinten (z-Achse) herausfahren. Auch die Bewegungen des Teleskops werden mittels Taster überwacht.

Zusätzlich verfügt das Hochregallager noch über eine Einlegestation, von der aus die Paletten durch die Bedieneinheit entweder eingelagert oder ausgelagert werden können.

Alle Sensoren sind bei diesem Aufbau so verschaltet, dass sie, sobald sie betätigt werden, einen von drei Interrupts entsprechend ihrer Achse (x-Achse, y-Achse, z-Achse) auslösen (siehe Tabelle 7.1). Die jeweils aktuelle Position kann dann durch Auslesen der entsprechenden Digital-Signale ermittelt werden.

Tabelle 7.1: HW-Interrupt-Zuordnung der Anlagenelektronik

Beschreibung	Interrupt Quelle
x-Achse	IRQ3
y-Achse	IRQ2
z-Achse	IRQ4
neuer Befehl	IRQ5

Die Bedienung der Anlage erfolgt über ein Tastenfeld, an dem unterschiedliche Befehle codiert werden können. Das Vorhandensein eines neuen Befehls wird der Steuerung durch einen weiteren Interrupt signalisiert.

7.1.2 Modell-basierter Entwurf

Bei dem in Kapitel 1.2 beschriebenen modell-basierten Entwurfsprozess werden unterschiedliche Phasen durchlaufen. Die Ergebnisse dieser Phasen bei der Entwicklung der HRL-Steuerung sind nachfolgend beschrieben.

Einbettender Prozess

Als erstes wurde der einbettende Prozess, hier das physikalische Verhalten des Hochregallagers, graphisch modelliert. Das resultierende Modell ist zur funktionalen Verifikation des Steuerungs-Systems unerlässlich.

Die drei unterschiedlichen Bewegungsrichtungen des Hochregallagers (x, y und z, siehe Abbildung 7.2) wurden zur Vereinfachung nicht mittels linearen Differentialgleichungen (mechanischen Bewegungsgleichungen) sondern durch einen Zustandsautomaten beschrieben.

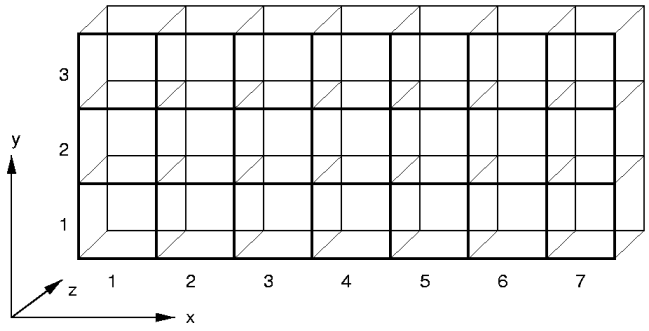

Abbildung 7.2: Abstraktes Koordinatensystem mit Positions-Nummern des HRL

Dieser zeitgesteuerte Zustandsautomat (siehe Ausschnitt für x-Achse in Abbildung 7.3) gibt bei entsprechenden Aktor-Signalen nach konstanten Zeitabständen (hier jeweils 3 Ticks) diskrete Positions-Änderungen (*newPosX*) aus und generiert die jeweiligen Interrupt-Signale (*IRQ_X*) für den ereignisgetriebenen Steuerungs-Modellteil.

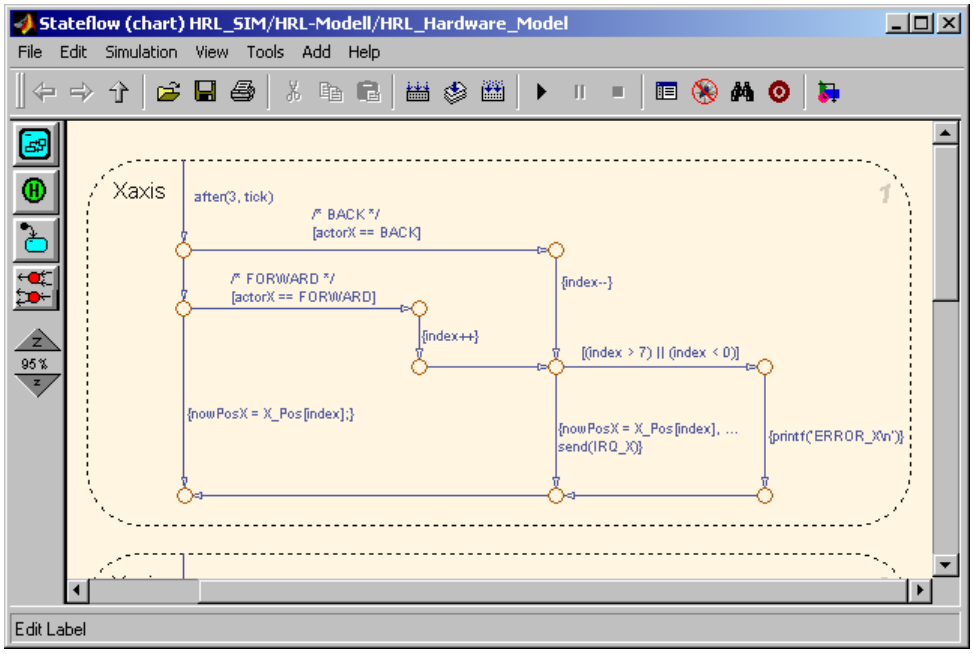

Abbildung 7.3: Hardware-Modell der Bewegung in x-Richtung des Hochregallagers

Steueralgorithmus

Bei der Entwicklung des funktionalen Modells zur Steuerung des Hochregallagers wurde iterativ vorgegangen. Ausgehend von einem abstrakten und sehr einfachen Funktionsmodell der Steuer-Logik hat der Entwickler durch stetige Verfeinerungen sukzessive immer mehr Funktionen konkret realisiert. Die Korrektheit aller inkrementellen Änderungen konnte durch jeweils geeignete Simulationsläufe verifiziert werden.

Die höchste Hierarchie-Ebene der entstandenen Steuerungs-Logik ist in Abbildung 7.4 dargestellt und kann in zwei Teile gegliedert werden.

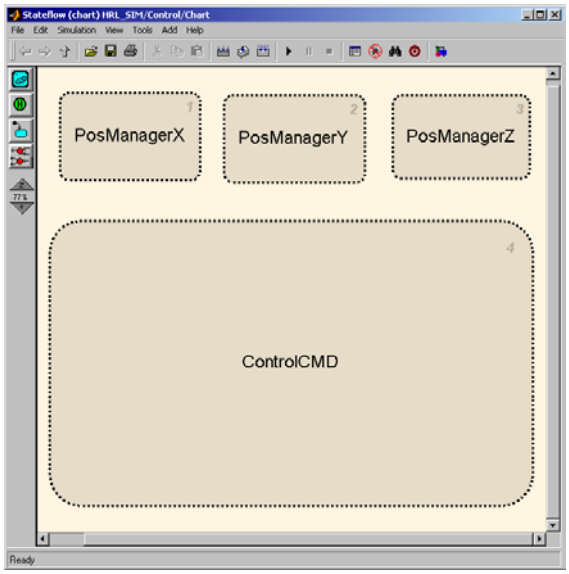

Abbildung 7.4: Stateflow-Chart mit der Steuer-Logik des HRL

Die verschiedenen Bewegungsrichtungen werden jeweils durch einen eigenen Zustandsautomaten (*PosManagerX*, *PosManagerY*, *PosManagerZ*) gesteuert. Dabei wird unabhängig voneinander das zugeordnete Aktor-Signal entsprechend der aktuell ermittelten Position (infolge eines Interrupts der Achse) und der gewünschten entgültigen Soll-Position bestimmt und ausgegeben. Dieser Teil ist durch die direkte Interaktion mit dem physikalischen System zeitkritisch.

Zusätzlich gibt es eine übergeordnete Kontroll-Logik (*ControlCMD*), die für die Ausführung von komplexen Bewegungsfolgen, wie z.B. „lagere Palette von Einlegestation in Position (x_1, y_1) ein“ oder „hole Palette aus Position (x_2, y_2)“, zuständig ist. Diese Kontroll-Logik gibt je nach gewünschtem Befehl unterschiedliche Soll-Positionen für die drei unabhängigen Bewegungsrichtungen vor und synchronisiert diese untereinander. Dabei werden alle physikalischen Randbedingungen der Anlage berücksichtigt. Zum Beispiel können x und y Bewegungen gleichzeitig ausgeführt werden, sofern sich der Teleskoparm in Mittelposition befindet. Während der Inbetriebnahme der Steuerung können Beschädigungen des Hochregallagers durch Fehler in der Steuer-Logik durch vorheriges gezieltes Simulieren von Abläufen vermieden werden. Dieser Teil ist nur beschränkt zeitkritisch, da lediglich die vom Bediener erwartete Performance beeinflusst wird. Schäden für Menschen und Anlage können aber nicht entstehen.

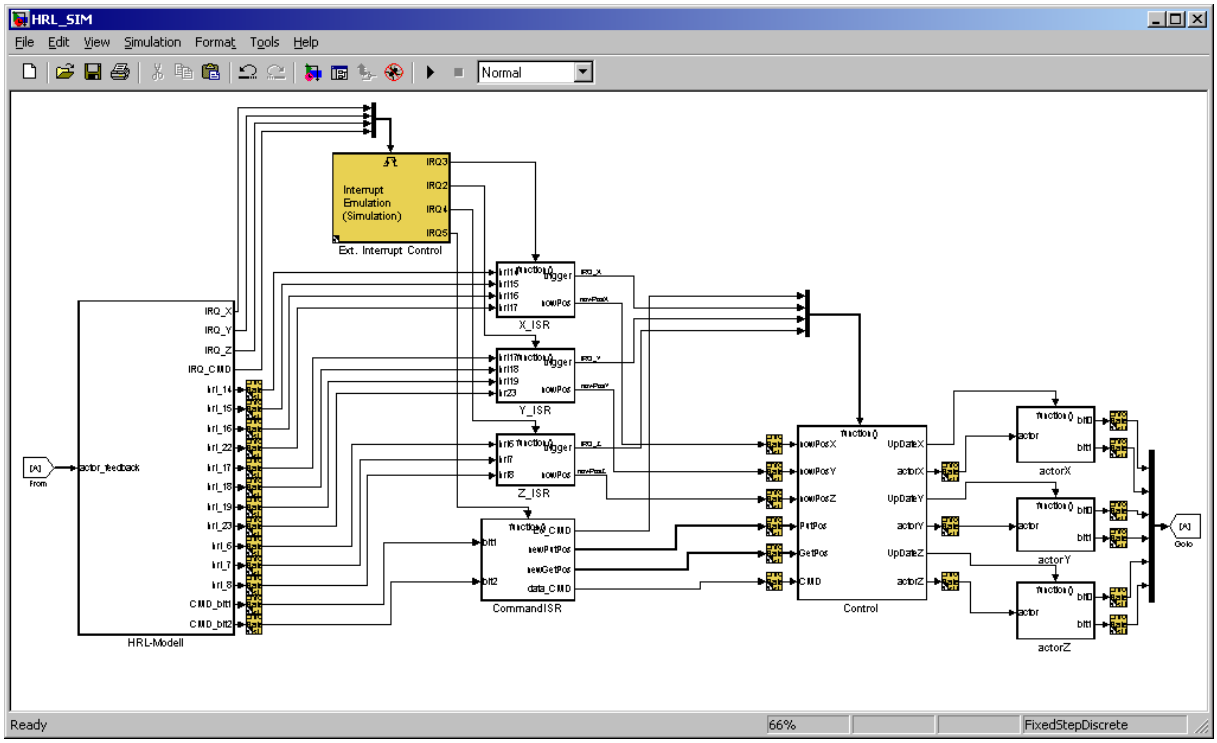

Abbildung 7.5: Simulations-Modell der HRL-Steuerung

Die Kommunikation zwischen den zwei Teilen erfolgt über globale Daten und Ereignisse. So triggert die übergeordnete Kontroll-Logik *ControlCMD* z.B. durch ein Ereignis *EV_startX* den Zustandautomaten *PosManagerX*, damit dieser die neue x Soll-Position *desPosX* anfährt. Der Hardware-Interrupt IRQ3 signalisiert von außen durch das Ereignis *EV_newPosX* dem *PosManagerX*, dass infolge der Bewegung eine neu x-Position erreicht wurde. Ist die Bedieneinheit am Ziel angekommen, deaktiviert *PosManagerX* den Aktor, stoppt somit die x-Bewegung und informiert durch das Ereignis *EV_reachedX* die Kontroll-Logik über das Erreichen der gewünschten x-Position. Nachdem der Steuerungs-Algorithmus zu Ende entwickelt war, wurden die unterschiedlichen HW-Anbindungen modelliert. Die Positions-Decodierungen der unterschiedlichen Bewegungsrichtungen wurden als Flussdiagramme in Stateflow entworfen und dem jeweiligen ISR-Function-Call-Subsystemen (*X_ISR*, *Y_ISR*, *Z_ISR*, CommandISR) zugeordnet. Analog wurde die Ansteuerung der Aktoren (*actorX*, *actorY*, *actorZ*) durch die entspre-

chenden digitalen Signale modelliert. Das endgültige Simulations-Modell des Entwurfs ist in Abbildung 7.5 dargestellt.

Verifikation durch Simulation

Zur reproduzierbaren Anregung des Designs wurde neben dem Hardware-Verhalten des Hochregallagers auch das Benutzerverhalten modelliert.

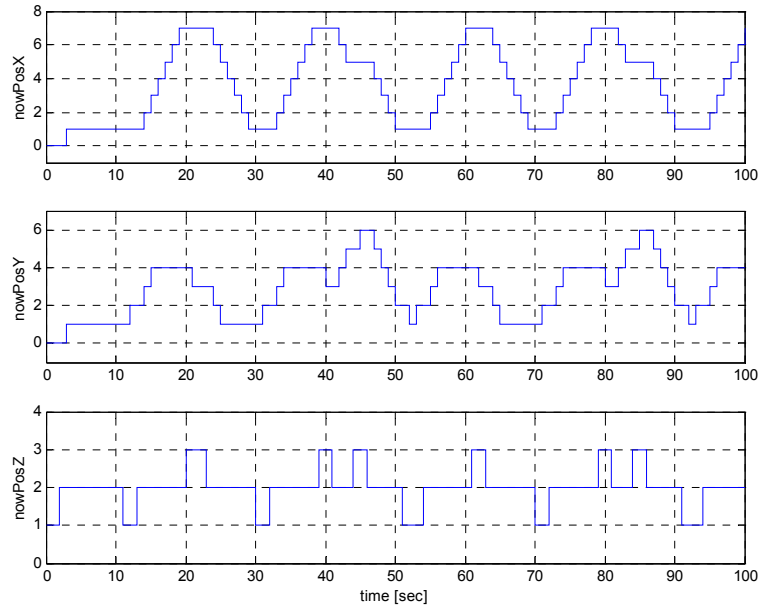

Abbildung 7.6: *nowPosX*, *nowPosY* und *nowPosZ* der Bedieneinheit beim Simulationslauf

Durch die gewählte Simulations-Anregung werden der Steuer-Logik unterschiedliche Sequenzen von Einlagerungs- und Auslagerungs-Befehle übergeben und das resultierende Verhalten untersucht. Dieselbe Anregung wird auch bei der späteren Co-Simulation zur Gewinnung der Ausführungszeiten der unterschiedlichen Aktionen angewandt.

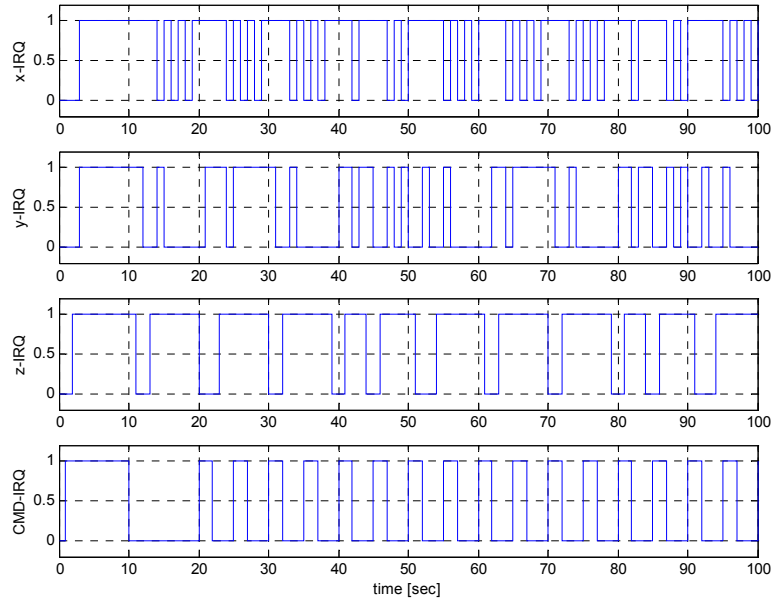

Abbildung 7.7: Interrupts im System beim gegebenen Simulationslauf

Ein typischer Bewegungsablauf ist in Abbildung 7.6 gegeben. Dabei sind über die Zeit auf drei unterschiedliche Achsen die jeweiligen aktuellen Positionen der Bedieneinheit aufgetragen. Da die Hochregallager-Hardware als Zustandsautomat modelliert wurde, ist auch der Übergang zwischen den unterschiedlichen Positionen diskret.

Im Diagramm gibt es für *nowPosY* sechs gültige Werte (1 bis 6), da bei der Modellierung eine lineare Abbildung der *unter* bzw. *ober* y-HRL-Position (1 bis 3) gewählt wurde. Generell gilt, dass alle ungeraden *nowPosY*-Werte unterhalb und alle geraden *nowPosY*-Werte oberhalb einer gewünschten y-HRL-Positionen liegen.

In Abbildung 7.7 sind die beim konkret gewählten Simulationslauf entstehenden Interrupts wiedergegeben, wobei jede Flanke einem IRQ entspricht.

Während der Simulation kann das Verhalten des Designs zudem durch die graphische Animation der Zustandsautomaten sehr einfach und komfortabel untersucht werden. Durch die Möglichkeit der Ablaufsteuerung, z.B. mittels Breakpoints an bestimmten Transitionen, können Fehler bei der Simulation sehr schnell erkannt und beseitigt werden.

7.1.3 Implementierung und Realzeit-Analyse

Nachdem die Funktion des graphischen Modells der HRL-Steuerung mittels Simulation getestet wurde, wird nun mit der echtzeitfähigen Umsetzung auf ein eingebettetes Zielsystem fortgefahren. Zur Veranschaulichung wesentlicher Sachverhalte werden nachfolgend drei unterschiedliche Implementierungs-Modelle dargestellt.

Beginnend mit der ursprünglichen Form der Implementierung durch die vorhandene Software-Architektur der Werkzeugkette, werden dessen Nachteile und Risiken kurz beschrieben.

Bei der zweiten Implementierungsvariante wird der Einsatz der neuen Konzepte und die realisierte Werkzeugunterstützung eingesetzt. Daraus ergeben sich mittels Co-Simulation und RT-Analyse Rückschlüsse über die getroffenen Implementierungsentscheidungen. Engpässe bzw. Abhängigkeiten der SW-Architektur werden ermittelt und können durch eine Optimierung bzw. ein Re-Design gelöst werden.

Dies geschieht im dritten Implementierungs-Modell, das ebenfalls unter Verwendung der Werkzeugerweiterungen eine komplexere, dem Problem angepasste Software-Architektur zur Abbildung des graphischen Modells umsetzt.

Ursprüngliche Implementierung

In Abbildung 7.8 ist ein mögliches Implementierungs-Modell der HRL-Steuerung unter Zuhilfenahme der standardmäßig vorhandenen Konstrukte von Simulink/Stateflow dargestellt.

Die Funktion der gesamten Steuerung ist in acht Function-Call-Subsystemen integriert, die durch genau vier externe Interrupts getriggert werden. Da die Datenstrukturen der Zustandsautomaten nicht reentrent sind, müssen alle Interrupt-Service-Routinen ununterbrechbar (nicht preemptive) ausgeführt werden, um Dateninkonsistenzen zu vermeiden.

Die wichtigsten Nachteile dieser Umsetzung können wie folgt zusammengefasst werden:

- Interrupts werden ausschließlich in der Hardware gepuffert. Da übliche Interrupt-Kontroller je Interrupt-Level jeweils nur ein Interrupt zwischenpuffern können, ist nicht auszuschließen, dass in ungünstigen Situationen Interrupts verloren gehen.
- Die worst-case Reaktionszeiten der Software auf die vorhandenen externen Interrupts sind, wegen der Ununterbrechbarkeit, stark von allen vorhandenen Ausführungszeiten des ereignisgetriggerten Software-Anteils abhängig.

- Dadurch ist bei Reaktionszeit-Problemen (bestimmte Deadlines können nicht eingehalten werden) eine Entkopplung der unterschiedlichen Dringlichkeit-Levels nicht möglich.

Zum Beispiel können durch die inherente Parallelität der HRL-Hardware zwei unterschiedliche Interrupts (*IRQ5* mit *CommandISR* und *IRQ3* mit *X_ISR*) in einem kurzen Zeitinterval nacheinander die Steuer-Logik triggern.

Die gewählte Implementierung würde den nieder-prioren Hardware *IRQ5* bearbeiten (da er als erstes aufgetreten ist), während der höher-priore *IRQ3* in der Hardware zwischengepuffert werden würde.

Die worst-case Reaktionszeiten des Systems auf *IRQ3* ist somit auch von der Ausführungszeit der *CommandISR* und ihren Nachfolgeaktionen abhängig. Die Trennung der beiden Ebenen (Hardware-Ansteuerung und Bedienerinteraktion) ist nicht möglich. Es wäre also durchaus möglich, dass eine zeitintensive Aktion, die durch den Bediener ausgelöst wird, die Reaktion der Software auf ein Positions-IRQ so lange verzögert, dass ein gefährlicher Anlagenzustand (z.B. das Überfahren einer Endposition) erreicht wird. Wenn dieser Zusammenhang durch eine geeignete Analyse erkannt wird, kann nur durch eine für das eigentliche Problem (Steuerung eines HRL) überdimensionierte Zielhardware Abhilfe geschafft werden, obwohl durch eine problemangepasste Software-Architektur mit einer viel geringeren Rechenleistung der Zielhardware das Problem gelöst werden könnte (siehe nachfolgende Implementierungsmodelle).

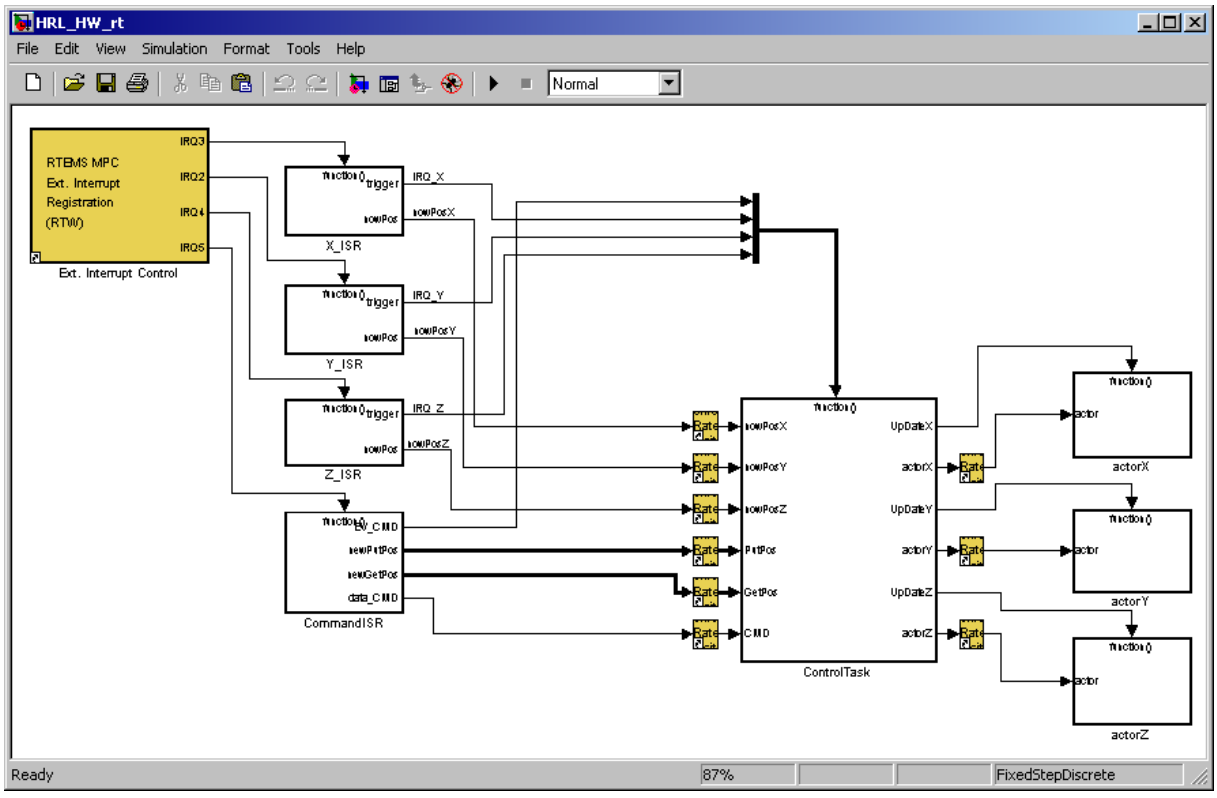

Abbildung 7.8: Ein mögliches Implementierungs-Modell durch die ursprüngliche Werkzeugunterstützung

Durch die beschriebenen Nachteile dieser konkreten Umsetzung ist der Einsatz nur bei Systemen mit sehr überdimensionierten Rechenleistungen gegenüber einem optimierten Zielsystem möglich. Die Anwendung bei zukünftigen eingebetteten Zielsystemen somit nicht geeignet.

Neue Implementierung

Nachfolgend wird die Umsetzung der beschriebenen HRL-Steuerung unter Verwendung der neuen Konzepte vorgestellt. Hierzu wird, ausgehend von dem angepassten Simulations-

Modell, die Aufspaltung auf die zwei für die Co-Simulation notwendigen Modelle (PC- und Target- Modell) beschrieben.

Aus der anschließend durchgeführten Realzeit-Analyse ergeben sich erste Rückschlüsse über die gewählte Implementierungs-Architektur und konkrete Optimierungsschritte für ein Re-Design des Modells können abgeleitet werden.

Das Ergebnis des Re-Designs wird im anschließenden Abschnitt beschrieben.

Simulations-Modell

Das erste Implementierungs-Modell mit den neuen Konzepten ist dem bereits beschriebenen Modell sehr ähnlich. Es wurden lediglich die Function-Call-Generatoren der Interrupt-Service-Routinen (*X_ISR*, *Y_ISR*, *Z_ISR*, *CommandISR*) durch *Message Generator* Blöcke ersetzt und die Empfangsseite mittels *Message Receive Queue* Block in einen Server-Thread umgewandelt. Das so entstandene Simulations-Modell ist in Abbildung 7.9 wiedergegeben. Es liefert bei einer Simulation dieselben Ergebnisse wie das bereits in Abschnitt 7.1.2 beschriebene ursprüngliche Modell.

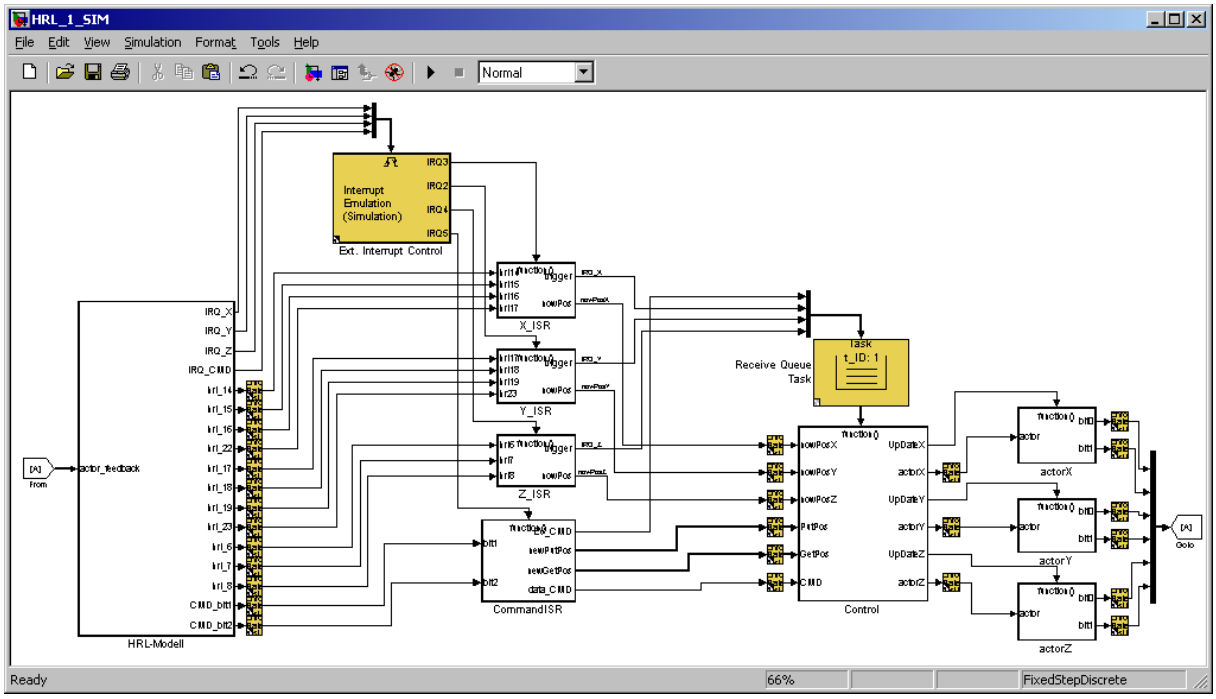

Abbildung 7.9: Erweitertes Simulations-Modell der HRL-Steuerung

Die Prioritäten der *Message Generator Blöcke* sind vorerst unwichtig, da sie während der Simulation nicht berücksichtigt werden (siehe Abschnitt 5.1.4).

Dieses Simulations-Modell ist der Ausgangspunkt für die Realzeitanalyse, die nachfolgend beschrieben wird. Erster Schritt dabei ist die Co-Simulation zur Parameterbestimmung.

Co-Simulation

Zur Co-Simulation wird das Simulations-Modell (siehe Abbildung 7.9) in das PC-Modell zur Emulation der HRL-Hardware (siehe Abbildung 7.10) und in das Target-Modell, das die eigentliche Steuer-Logik umsetzt (siehe Abbildung 7.11), aufgeteilt.

Dabei werden die bereits in Abschnitt 5.2.1 beschriebenen Blöcke zur Bereitstellung der Kommunikation während der Co-Simulation in die Modelle integriert.

Die Stimulation der Software-Architektur erfolgt während der Co-Simulation ebenfalls durch das PC-Modell, das dieselben Testvektoren erzeugt, die bereits zur Validierung des Designs während der modell-basierten Entwicklung eingesetzt wurden.

Die aufgezeichneten Ergebnisse eines Co-Simulations-Laufs können dann mit den Ergebnissen aus den früheren Simulationen verglichen werden, um etwaige Unterschiede zu erkennen. Diese können wie bereits mehrmals erläutert (siehe Abschnitt 5.1.4) durch ein schlechtes Design oder Abweichungen im Laufzeitsystem (z.B. Prioritätsüberholungen) entstehen.

Ein möglicherweise notwendiges Re-Design aufgrund dieser unerwünschten Diskrepanzen wird somit zu diesem Zeitpunkt im Entwurfsprozess erkannt.

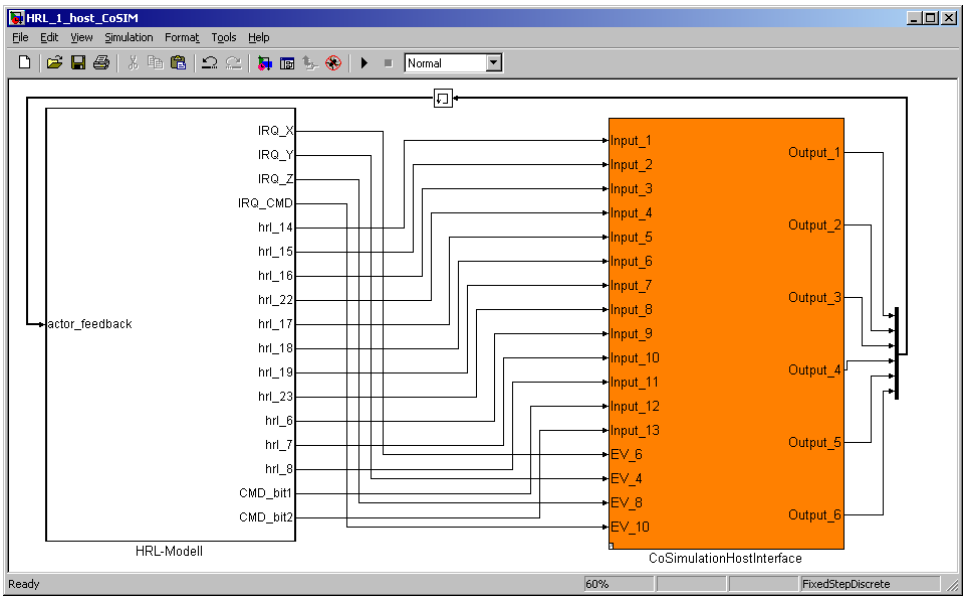

Abbildung 7.10: PC-Modell zur Emulation der HRL-Hardware während der Co-Simulation

Nachfolgend wird die Auswertung der ebenfalls während der Co-Simulation gewonnenen Ausführungszeiten beschrieben.

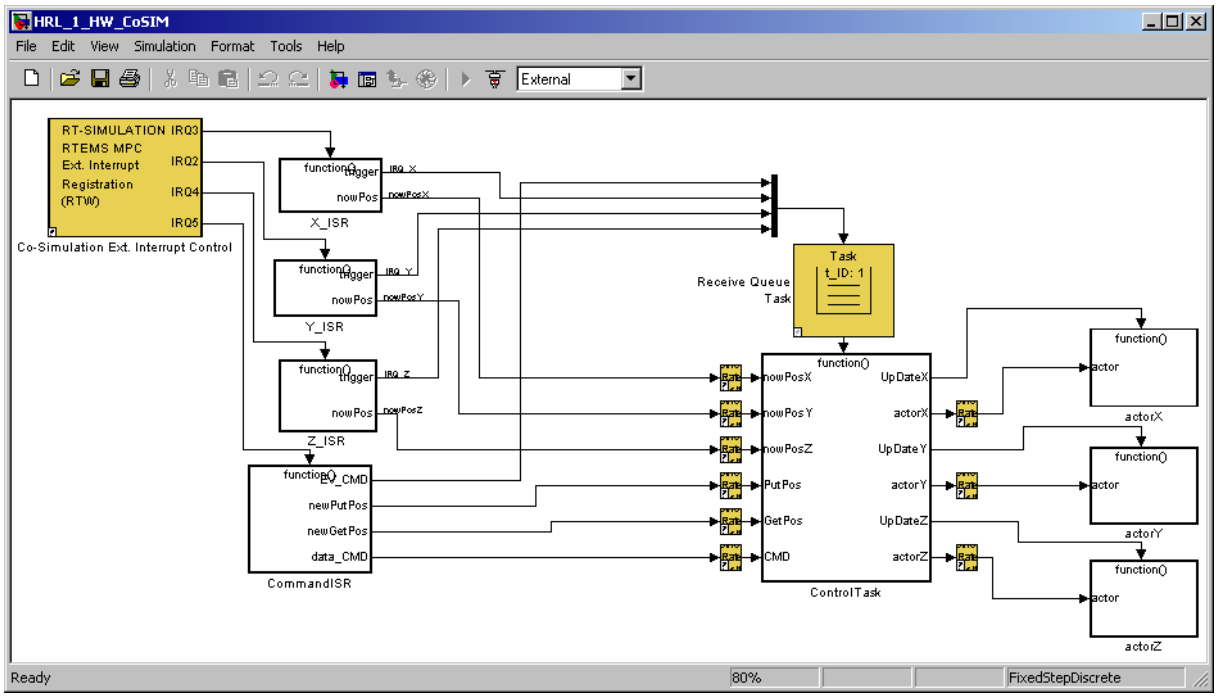

Abbildung 7.11: Target-Modell zur Abbildung der HRL-Steuer-Logik für die Co-Simulation

Realzeit-Analyse

Die gemessenen Ausführungszeiten werden einfach mittels der in Abschnitt 5.2.2 beschriebenen Skripten ausgewertet.

Als Beispiel sind hier in Abbildung 7.12 die gemessenen Ausführungszeiten der Interrupt Service Routine *Z_ISR* dargestellt. Das obere Diagramm gibt die Auftrittshäufigkeit einer bestimmten Ausführungszeit wieder. Die Auflösung des Messverfahrens liegt durch den gewählten Hardware-Timer-Takt bei 0,4 µsec und entspricht der Breite der Balken im Histogramm.

Da die Interrupts unabhängig von der Zeit eintreffen, sind im zweite Diagramm die Ausführungszeiten nicht über die absolute Simulations-Zeit, sondern nach ihrer jeweiligen Auftrittsreihenfolge aufgetragen.

Zur Ermittlung der im *konkreten APG* verwendeten Ausführungszeiten der Aktionen werden die Maxima der gemessenen Ausführungszeiten verwendet (hier 0,0656 msec) und um die bei der Co-Simulation nicht gemessenen RTOS-Zeiten (hier die IRQ Vor- und Nachverarbeitung) vergrößert (siehe Abschnitt 5.2.1).

Dadurch ergibt sich für die Ausführungszeit von Aktion $A8$ im *konkreten APG* (entspricht *Z_ISR*) aus Abbildung 7.13 eine Ausführungszeit von 0,0935 msec.

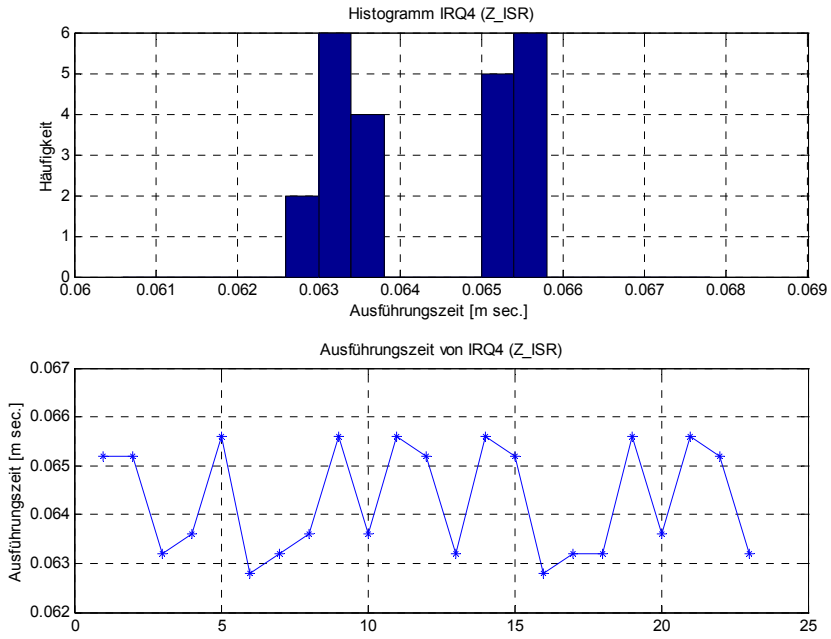

Abbildung 7.12: Gemessene Ausführungszeiten für die Interrupt-Service-Routine Z_ISR

Der gesamte resultierende *konkrete Aktions-Präzedenz-Graph* des ersten Implementierungs-Modells ist in Abbildung 7.13 dargestellt. Die Aktionen $A5$, $A6$, $A7$ und $A8$ spiegeln die jeweiligen Interrupt-Service-Routinen der SW-Architektur mit ihren Ausführungszeiten wieder. Die Reaktion des Server-Threads ($t\_ID = 1$) auf die internen Ereignisse sind Aktionen $A1$ bis $A4$.

Bei der Betrachtung der Ausführungszeiten wird klar, dass Aktion $A4$ mit 2,282 msec den Server-Thread am längsten blockieren kann. Die mit dem Skript `responsetime_BPI.m` oder `responsetime_PT.m` berechnete worst-case Reaktionszeit von z.B. Aktion $A1$ ist folglich 3,01 msec.

Die dabei angenommenen Ereignisströme[18] für die Rechenzeitanforderung durch das einbettende System waren wie folgt:

$$ES_{IRQ5}^{transaction1}: \left\{\begin{pmatrix}2\text{sec}\\0\end{pmatrix}\right\},\quad ES_{IRQ3}^{transaction2}: \left\{\begin{pmatrix}1\text{sec}\\0\end{pmatrix}\right\},\quad ES_{IRQ2}^{transaction3}: \left\{\begin{pmatrix}1\text{sec}\\0\end{pmatrix}\right\},\quad ES_{IRQ4}^{transaction4}: \left\{\begin{pmatrix}1\text{sec}\\3\end{pmatrix}\right\}$$

In Matlab wird diese Rechenzeitanforderung wie folgt spezifiziert:

```
>ES = {[struct('z', 2, 'a', 0)], [struct('z', 1, 'a', 0)],...
>      [struct('z', 1, 'a', 0)], [struct('z', 1, 'a', 3)]}
```

Neben der maximalen Blockierung durch *A4* wurde bei der Berechnung auch die Interferenz durch den höher-prioren und gleichzeitig auftretenden *IRQ2* (*Y_IRQ*) berücksichtigt.

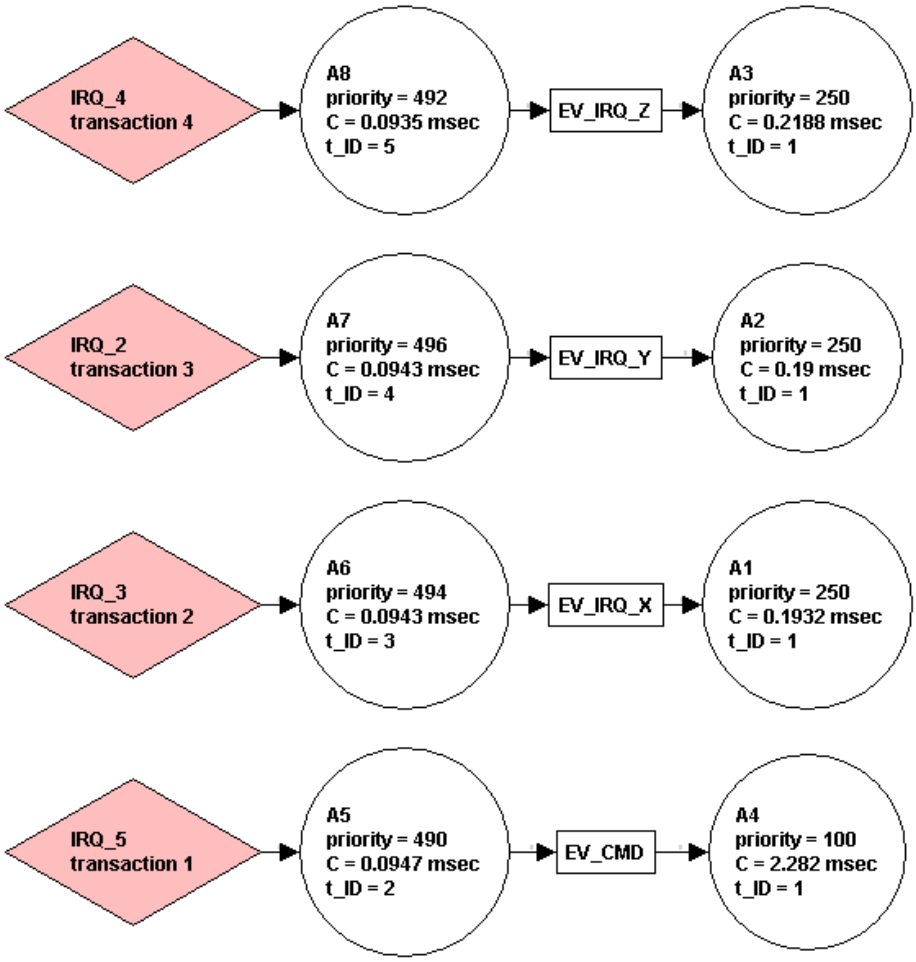

Abbildung 7.13: *Aktions-Präzedenz-Graphen* des ersten Implementierungs-Modells aus Abbildung 7.11

Die Blockierung des Server-Threads durch *A4* ist für die betrachteten Zeitanforderungen im gegebenen mechanisch sehr trägen Beispiel kaum ein Problem. Trotzdem können in anderen zeitkritischeren Beispielen ähnliche Probleme auftreten, die eine Entkopplung unumgänglich werden lassen.

[18] Diese Ereignisströme wurden so gewählt, dass bestimmte Zusammenhänge erklärbar sind. Sie entsprechen keiner realen Rechenzeitanforderung durch das einbettende System. Dies ist gerechtfertigt, da nicht die Applikation im Vordergrund steht, sondern die Anwendbarkeit der beschriebenen Konzepte.

Diese Trennung unterschiedlicher Levels, die im ursprünglichen Implementierungs-Modell nicht möglich ist, wurde im gerade untersuchten Implementierungs-Modell noch nicht durchgeführt. Sie soll nachfolgend realisiert werden.

Die ungünstige Verkopplung der Aktionen *A1*, *A2* und *A3*, die harte zeitkritische Anforderungen besitzen (Reaktion auf IRQ aus Hardware) mit der weniger zeitkritischen Aktion *A4* (Bedienereingaben) wurde aus der durchgeführten RT-Analyse des ersten Implementierungs-Modells ersichtlich.

Dadurch kann nun in einem neuen Iterations-Schritt das erste Implementierungs-Modell zielführend optimiert werden. Das Ergebnis ist das nachfolgend vorgestellte optimierte Implementierungs-Modell der HRL-Steuerung.

Optimierte Implementierung

Ziel der Optimierung ist eine Entkopplung der unterschiedlichen Dringlichkeits-Ebenen im graphischen Modell. Hierzu wurde die gegebene graphische Spezifikation auf zwei separate Server-Threads mit jeweils eigener *Message Receive Queue* umgesetzt.

Simulations-Modell

Das resultierende Simulations-Modell ist in Abbildung 7.14 wiedergegeben.

Aus der Realzeit-Analyse des einfachen Implementierungs-Modells ergab sich eine ungünstige Verkopplung von Aktionen mit unterschiedlichen Dringlichkeits-Levels in ein und demselben Thread. Dies führte zu unerwünschten großen Blockierzeiten und entsprechend langen worst-case Reaktionszeiten der Software-Implementierung.

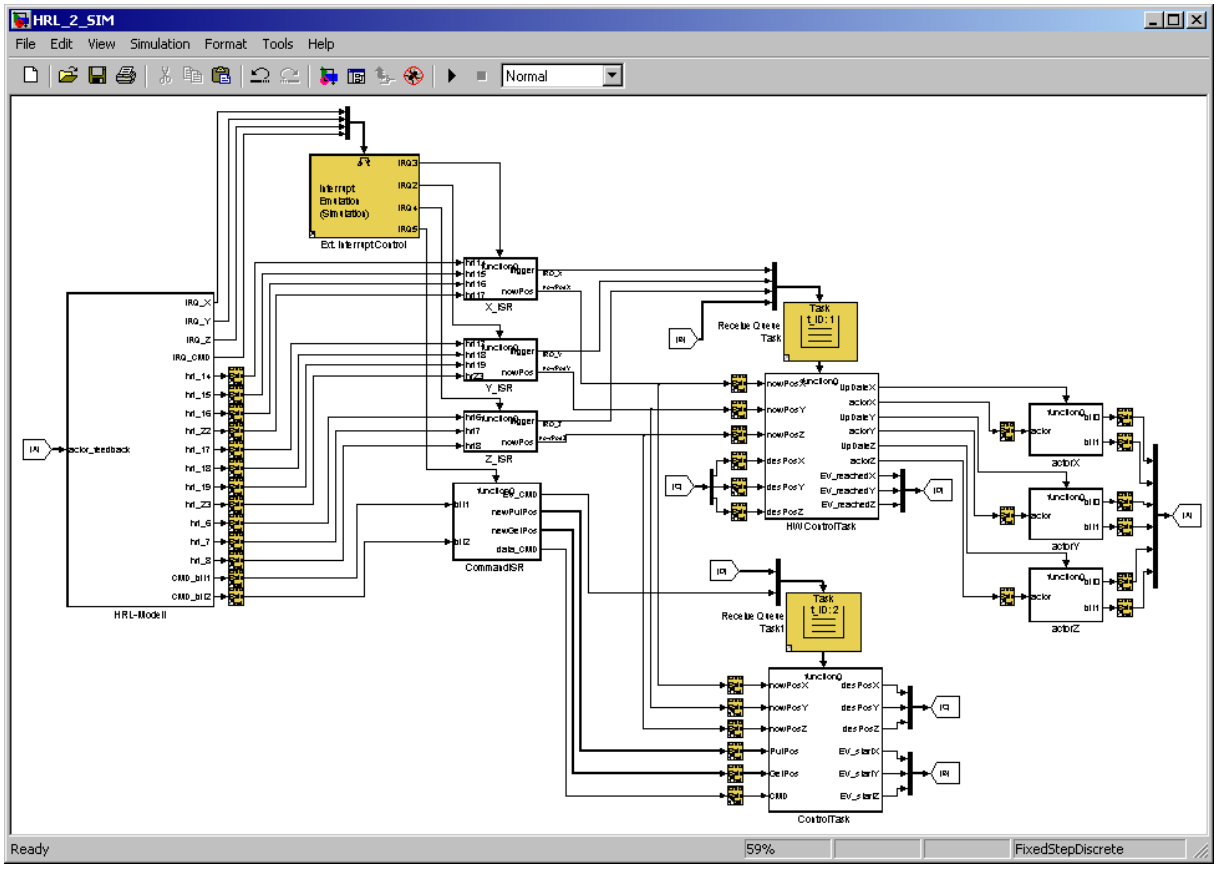

Abbildung 7.14: Simulations-Modell der optimierten Implementierung

Durch das Erkennen der Ursache für diese langen Reaktionszeiten konnte eine Aufteilung der zwei Steuer-Logik-Teile (drei Positions-Manager und Kontroll-Logik, siehe Abschnitt 7.1.2) auf separate Server-Threads vorgenommen werden. Das Verkopplungs-Problem wurde so gelöst.

In Abbildung 7.15 sind die zwei Server-Threads *HWControlTask* und *ControlTask* gut erkennbar. Der obere Thread empfängt die höher-prioren Ereignisse aus den ISR der Bewegungsachsen und die Start-Ereignisse der übergeordneten Kontroll-Logik. Letztere werden durch den unteren Thread versendet. Dieser verarbeitet die Synchronisations-Ereignisse der unterschiedlichen Positions-Manager und die eingegebenen Befehle des Bedieners der Anlage.

Dieses Simulations-Modell verhält sich bei der funktionalen Simulation genauso wie die bereits vorher erklärten Modelle. Der grundlegende Unterschied ist, dass hier die Ereignisse *EV_startX, Y, Z* und *EV_reachedX, Y, Z* in der SW-Architektur sichtbar werden und die Synchronisation zweier Server-Threads bereitstellen. Im vorigen Implementierungsmodell waren die genannten Ereignisse lokal nur im Steuer-Automaten sichtbar und ihre Aktionen daher nicht über Prioritäten untereinander priorisierbar.

Co-Simulation

Das aus dem Simulations-Modell gewonnene Target-Modell für die Co-Simulation ist in Abbildung 7.15 dargestellt. Die Hardware-Anbindung stimmt mit dem vorherigen Implementierungs-Modell überein. Die Steuer-Logik ist jedoch auf zwei unterschiedliche Threads aufgeteilt, um die Antwortzeit auf die Bewegungs-IRQs zu verbessern.

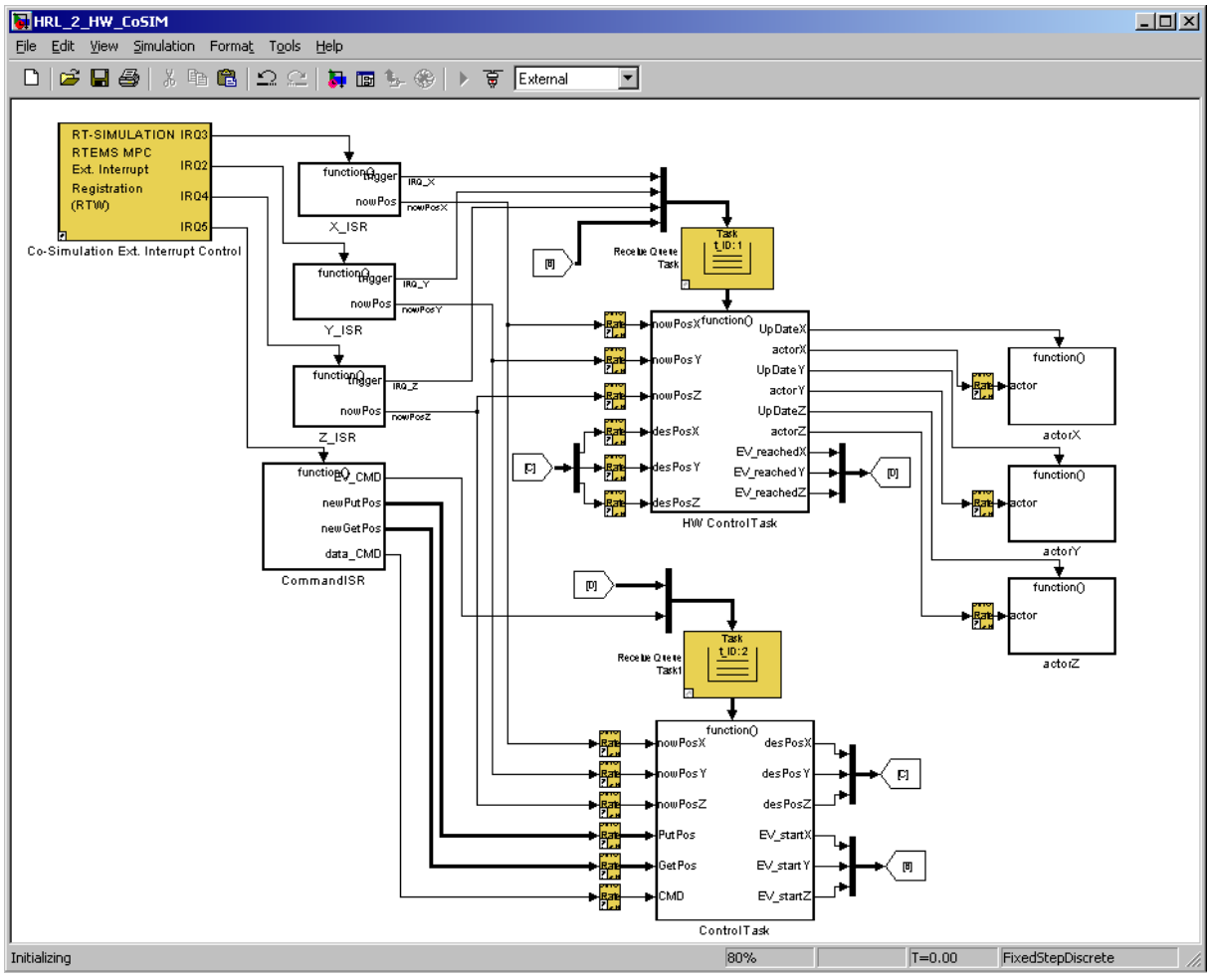

Abbildung 7.15: Target-Modell zur Co-Simulation der optimierten Implementierung

Das entsprechende PC-Modell für die Co-Simulation entspricht dem bereits in vorherigen Abschnitt vorgestellten PC-Modell, da sich durch die beschriebene Änderung des Implementierungs-Modells keine Rückwirkungen auf das Verhalten der HRL-Hardware ergeben.

Realzeit-Analyse

Durch die Aufspaltung der Steuer-Logik auf zwei Server-Threads ergeben sich hier vierzehn unterschiedliche Aktionen im System. Jede Aktion wurde durch die Wahl der Stimuli während eines Co-Simulations-Laufes wenigstens einmal ausgeführt, um realistische Ausführungszeiten für die RT-Analyse zu erhalten.

In Abbildung 7.16 sind die sich aus dem Parse-Vorgang ergebenden *konkreten APG* des optimierten Implementierungs-Modells dargestellt. Durch die Aufteilung der Steuer-Logik und die damit verbundene Einführung von zusätzlichen Ereignissen in das System wird eine feinere Aufteilung von Reaktionen auf bestimmte Ereignisse möglich.

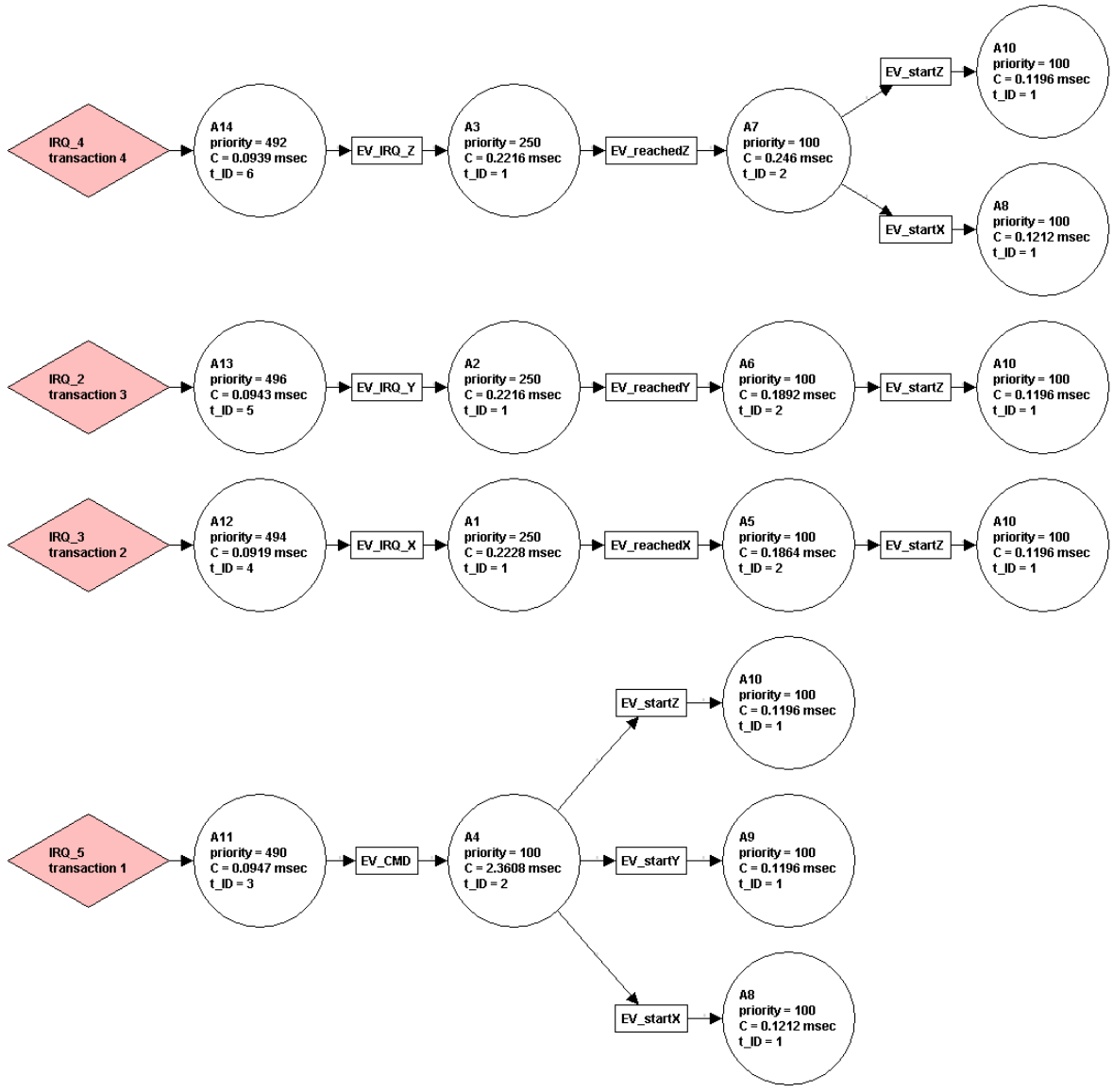

Abbildung 7.16: *Konkrete APG* des optimierten Implementierungs-Modells (zwei Server-Threads) der HRL-Steuerung

So ist zum Beispiel die Reaktion auf eine neue Position in x-Richtung, die den x-Aktor betrifft, bereits nach Bearbeitung von Aktion $A1$ zu Ende, obwohl der durch IRQ_3 angestoßene Trans-

aktions-Graph erst mit $A10$ endet. Durch die etwas niedriger gewählte Priorität des Nachfolge-Ereignisses $EV\_reachedX$ und die Verlagerung der zugeordneten Aktion $A5$ in den anderen Server-Thread ($t\_ID = 2$) kann trotz Weiterbearbeitung der besagten Transaktion ohne direkte Blockierung auf neue HW-Interrupts, die durch die Bewegung der Bedieneinheit entstehen, reagiert werden.

Diese feinere Granularität bei Aktionen und die Möglichkeit, diese zu priorisieren, ist der Hauptvorteil des optimierten Implementierungs-Modells.

Wie bereits zuvor lassen sich aus den gewonnenen Daten mit Hilfe der umgesetzten Formeln für beliebige Aktionen die worst-case Reaktionszeiten berechnen. Für die hier angeführten Zahlenwerte wurde wieder dieselbe Rechenzeitanforderung wie im vorigen Abschnitt angenommen.

Diesmal ergibt sich für die worst-case Reaktionszeit der Software vom Eintreffen eines externen x-Interrupts bis zum Stoppen der Bewegung, also bis zur Aktion A1, ein Wert von: 0,85 msec. Der dabei berücksichtigte Anteil durch Blockierung (Aktion A8) ist in dieser Implementierungs-Variante auf 0,1212 msec geschrumpft. Diese Zahlenwerte sind sowohl für das BPI- als auch das PT-Protokoll identisch, da durch die gewählten Prioritäten der Ereignisse keine Verkopplung der Blockierung zwischen den zwei Server-Threads eintreten kann (siehe Gleichungen 4.4 4.5 4.6 aus Abschnitt 4.2.1).

RT-Modell

Nachdem erfolgreich der Nachweis über die Einhaltung aller geforderten Antwortzeiten der erhaltenen RT-Implementierung durchgeführt wurde, kann die Implementierung der Steuer-Logik für den endgültigen Realzeit-Einsatz durchgeführt werden.

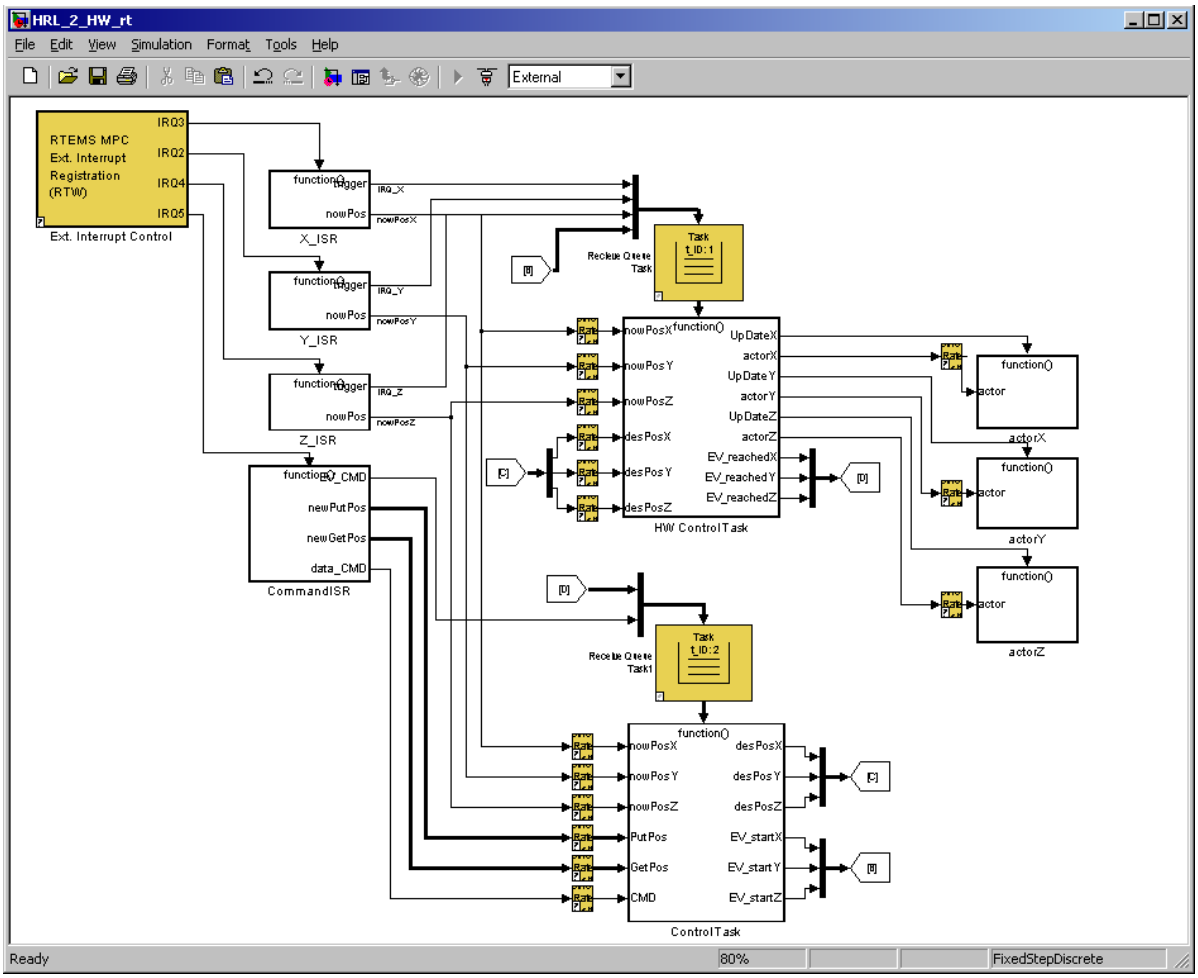

Abbildung 7.17: RT-Modell zur Umsetzung des optimierten Implementierungs-Modells

Hierzu werden die für die Co-Simulation notwendigen Blöcke im Target-Modell durch geeignete Treiberblöcke (IRQ-Block und digital Ein-Ausgabe-Blöcke) ersetzt und die finale Codegenerierung gestartet. Das resultierende RT-Modell ist zur Vollständigkeit in Abbildung 7.17 wiedergegeben.

7.1.4 Ergebnisse und Erfahrungen

Am Beispiel der Hochregallager-Steuerung wurde ein mögliches Szenario bei der modell-basierten Entwicklung eingebetteter Realzeit-Software dargestellt. Obwohl die konkret ausgewählte Aufgabe durch das verwendete Zielsystem keine wirkliche realzeittechnische Herausforderung darstellt, konnte das Vorgehen bei der Entwicklung solcher Systeme dargestellt werden.

Ausgehend von einer relativ einfachen SW-Architektur (neue Implementierung siehe Abschnitt7.1.3) wurde die Gewinnung der Ausführungszeiten mittels Co-Simulation erklärt. Dabei wurde die Umwandlung des Simulations-Modells in ein PC- und ein Target-Modell dargestellt.

Aus der anschließenden RT-Analyse mit dem Parsen des Target-Modells zur Gewinnung der konkreten APG und der Weiterverarbeitung der gewonnenen realistischen Ausführungszeiten konnte ein Feedback über die Auslastung des Systems bei den gewählten Implementierungs-Entscheidungen gewonnen werden.

Dieses Feedback hat auf die ungünstige Verkopplung unterschiedlicher Dringlichkeits-Level der Steuer-Logik bei der gewählten Implementierungs-Variante hingewiesen. Durch eine Analyse der Ergebnisse konnte eine Lösung (Entkopplung der Ebenen) erarbeitet werden.

Dieser Lösungsvorschlag führte dann durch ein Re-Design des vorliegenden Modells zu dem neuen, optimierten Implementierungs-Modell, das ebenfalls mittels Co-Simulation und RT-Analyse untersucht wurde. Die Ergebnisse bestätigten die gewünschte Entkopplung und die daraus resultierenden optimierten Reaktionszeiten.

Das erhaltene Modell wurde somit zur endgültigen Implementierung weiter verwendet. Dabei konnte gezeigt werden, dass sowohl die reine PC-Simulation als auch die Co-Simulation bei dem erhaltenen funktionalen Design das identische Verhalten aufwiesen.

Die wichtigsten gewonnenen Erfahrungen werden wie folgt zusammengefasst:

- Die Bereitstellung einer Simulations-Funktionalität ist ein wesentlicher Bestandteil eines modell-basierten Ansatzes.
- Die Co-Simulation bietet nicht nur ein realistisches Feedback über die Auslastung des Target-Systems (wenn auch nicht worst-case Fall),
- sondern ermöglicht auch Unterschiede im funktionalen Verhalten zwischen reiner PC-Simulation und RT-Implementierung auf einem RTOS zu erkennen. So können diese Unterschiede frühzeitig durch ein geeignetes Re-Design beseitigt werden.
- Durch die Visualisierung der Aktionen in den *konkreten Aktions-Präzedenz-Graphen* werden Zusammenhänge bzw. Abhängigkeiten leichter sichtbar. Fehler bei der Wahl von Prioritäten oder bei der Zuordnung von Funktionen zu Server-Threads können einfacher ermittelt werden und mögliche Optimierungen sind intuitiv erkennbar.

7.2 Hydrostatische Vorderachs-Antriebs-Steuerung

Als zweites Anwendungsbeispiel wird nachfolgend die Umsetzung eines Steuerungs-Systems für einen hydrostatischen Vorderachsantrieb (HVA) vorgestellt. Die Spezifikation der Steuerung wurde zusammen mit MAN [74] entwickelt.

Im Gegensatz zum vorherigen Beispiel handelt es sich hier um ein hybrides graphisches Modell, da sowohl zeitgesteuerte als auch ereignisgesteuerte Anteile auf graphischer Ebene zu spezifizieren sind.

7.2.1 Systembeschreibung

Die zu entwickelnde Steuerung hat die Aufgabe bei einem Nutzfahrzeug einen hydrostatischen Vorderachsantrieb ein- und auszuschalten. Zum Ein- und Aus-Schalten muss die Steuerung mehrere Magnetschaltventile (ZP1-ZP3) und die Kupplung der Haupt- und Speise-Pumpe in einem zeitlich festen Ablauf ansteuern. Anhand eines Kennlinienfeldes zwischen Geschwindigkeit und Öltemperatur wird entschieden, ob der Antrieb ein-, aus-, automatisch ein- oder automatisch ausgeschalten wird. Die Steuerung muss auch Fehler (A/D-Wandlerausfall, Bruch der Hydraulikleitung oder Ausfall des CAN-Busses) erkennen und den Antrieb daraufhin abschalten.

Für die Umsetzung der Steuerung wurde während der Entwicklung ein *By-Pass* System umgesetzt. In einem separaten Steuergerät (nicht Ziel der Entwicklung) werden alle relevanten Daten des LKWs (z.B. Ein/Aus-Schalter, Geschwindigkeit, etc.) in zwei CAN-Nachrichten verpackt und dem zu entwickelnden Steuergerät geschickt. Dieses Steuergerät verarbeitet die erhaltenen Nachrichten (Daten) entsprechend der entwickelten Funktion und liefert über zwei andere CAN-Nachrichten die Ergebnisse zurück. So konnte im ersten Ansatz die Anbindung der physikalischen Umwelt (LKW) erleichtert werden, da nur über den CAN-Bus Daten mit dem zu entwickelnden Steuergerät ausgetauscht werden.

Bricht die Kommunikation zu den Sensoren für länger als eine definierte Zeitgrenze ab, wird die Steuerung ebenfalls in einen definierten Zustand gebracht.

7.2.2 Modell-basierter Entwurf

Das Ergebnis des modell-basierten Entwurfs, das Simulations-Modell der HVA-Steuerung, ist in Abbildung 7.18 dargestellt. Es besteht aus drei Server-Threads. Der oberste Thread *Data-Check* bereitet die über CAN-Nachrichten empfangenen Daten auf. Er wird nur durch die CAN-Interrupts getriggert.

Die beiden verbleibenden Threads verarbeiten sowohl Zeit als auch Interrupt getriggerte Ereignisse (hybride Komponente).

Grundsätzlich gilt jedoch, dass der Server-Thread *EventDriven* hauptsächlich Ereignisse aus den CAN-Nachrichten verarbeitet und somit keinen direkten Bezug zur absoluten Zeit besitzt. Treffen zum Beispiel keine CAN-Nachrichten mehr ein, würde dieser Thread nicht mehr getriggert.

Aus diesem Grund wurde ein zweiter Server-Thread, *TimeDriven,* umgesetzt, der hauptsächlich ein zyklisches Ereignis durch einen Taktgeber verarbeitet. Er verwaltet eine interne Zeitbasis, die auch die Generierung der benötigten Verzögerungs-Signale bei den definierten Ein- und Ausschalt-Zyklen ermöglicht.

Ein normaler Ablauf wäre z.B. beim manuellen Einschalten, dass über die CAN-Nachricht die Betätigung des Schalters der Steuerung gemeldet wird. Der Server-Thread *EventDriven* führt daraufhin alle sofort notwendigen Aktionen durch und meldet dem Server-Thread *TimeDriven* den Wunsch, in definierten Zeitabständen bestimmte Signale (definierte Verzögerungen) zu erhalten. Nach dem Ablaufen der gewünschten Verzögerungen wird jeweils das geeignete Signal versendet und der Thread *EventDriven* kann den Einschalt-Vorgang zu Ende führen.

Durch diese Architektur ist es möglich, eine Trennung auf Modellebene zwischen Teilen ohne Zeit-Wissen und einem Teil zur Zeit-Verwaltung durchzuführen.

Zur Simulation der graphisch spezifizierten Steuer-Logik wurde zu Beginn der Entwicklung ein abstraktes Abbild des einbettenden Prozesses modelliert. Hierbei wurden nur die wirklich wesentlichen Aspekte berücksichtigt.

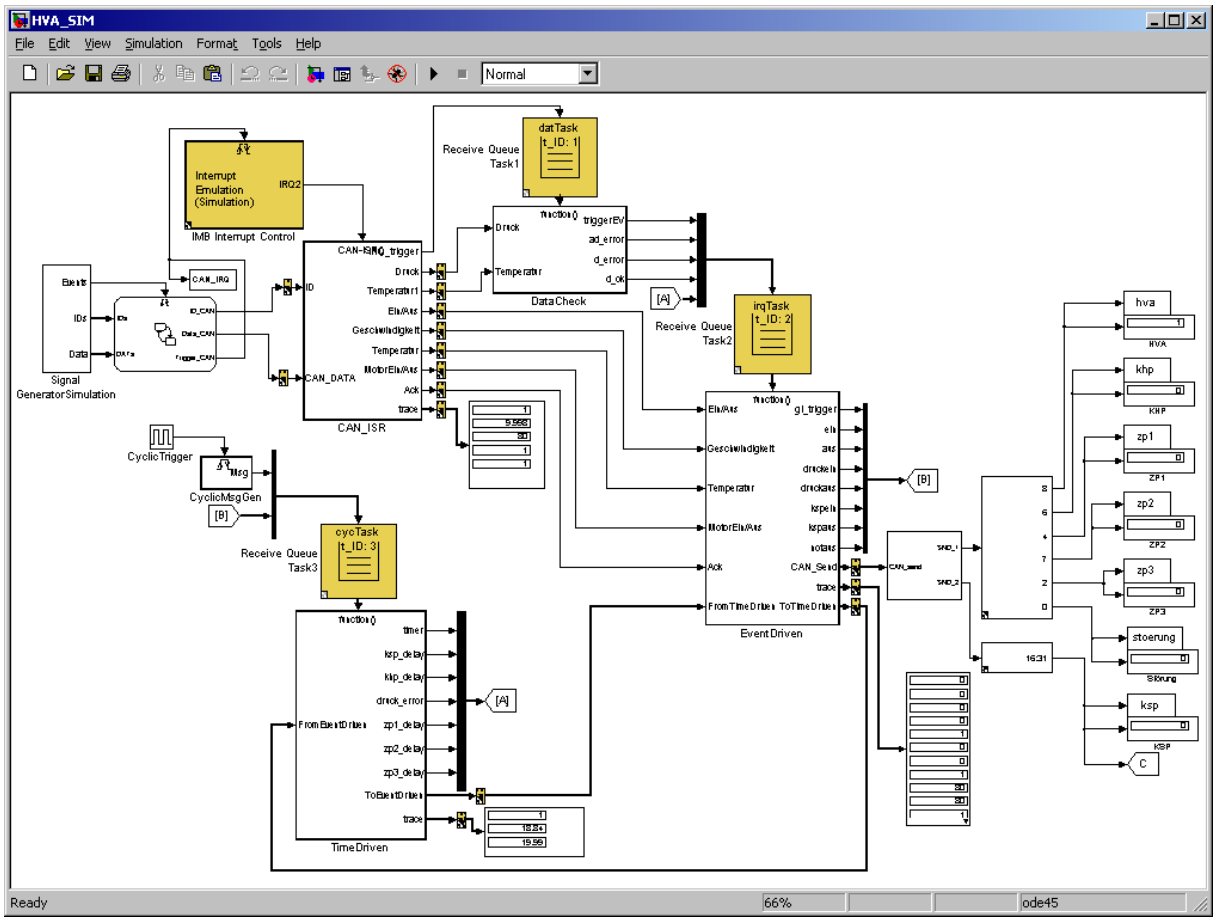

Abbildung 7.18: Simulations-Modell der HVA-Steuerung

Neben der Modellierung des Öldruck-Signals, das dem zeitverzögerten Speisepumpen-Signal der Steuer-Logik entspricht, wurde auch die Bereitstellung aller notwendigen Umgebungs-Signale realisiert, wie z.B. CAN-Interrupt, Ein/Aus Schalter, Acknowledge, Motor Ein/Aus, Temperatur und Geschwindigkeits-Signale.

Durch eine geeignete Anregung kann eine reproduzierbare Ansteuerung des zu entwickelnden Modells erreicht werden, in der alle spezifizierten und notwendigen Funktionen getestet werden können. Dieselbe Ansteuerung kann dann während der Co-Simulation zur Bestimmung der realistischen Ausführungszeiten eingesetzt werden.

In Abbildung 7.19 und Abbildung 7.20 sind jeweils die Anregung und die durch die graphisch spezifizierte Steuer-Logik resultierenden Antwort-Signale wiedergegeben.

Im Zeitintervall von 0 bis ca. 2 sec findet ein gewöhnlicher Einschalt-Zyklus statt, dessen Ende durch das Abschalten des Ventils *zp3* gekennzeichnet ist.

Mit der steigenden Flanke von *zp3* bei ungefähr 2,4 sec der Simulationszeit wird ein automatischer Abschaltvorgang eingeleitet, da die Geschwindigkeit des LKWs über den Grenzwert des hydrostatischen Vorderachsantriebs gestiegen ist. Der automatische Abschalt-Zyklus endet mit dem Abschalten von *zp3* bei ca. 3,8 sec.

Nach 3,8 sec erfolgt ein automatischer Einschalt-Zyklus, nachdem die LKW-Geschwindigkeit wieder unter einen zweiten Grenzwert (Hysterese) gesunken ist.

Zum Zeitpunkt 7 sec wird ein manueller Ausschalt-Zyklus gestartet (siehe Ein/Aus-Signal), der bei ca. 8 sec Simulationszeit zu Ende ist.

In der 9. sec wird ein Sprung der Öltemperatur simuliert, um die Erkennung einer solchen Störung zu testen. Durch das Signal Acknowledge zum Zeitpunkt 9,8 sec wird die gespeicherte Störung der Steuer-Logik zurückgesetzt.

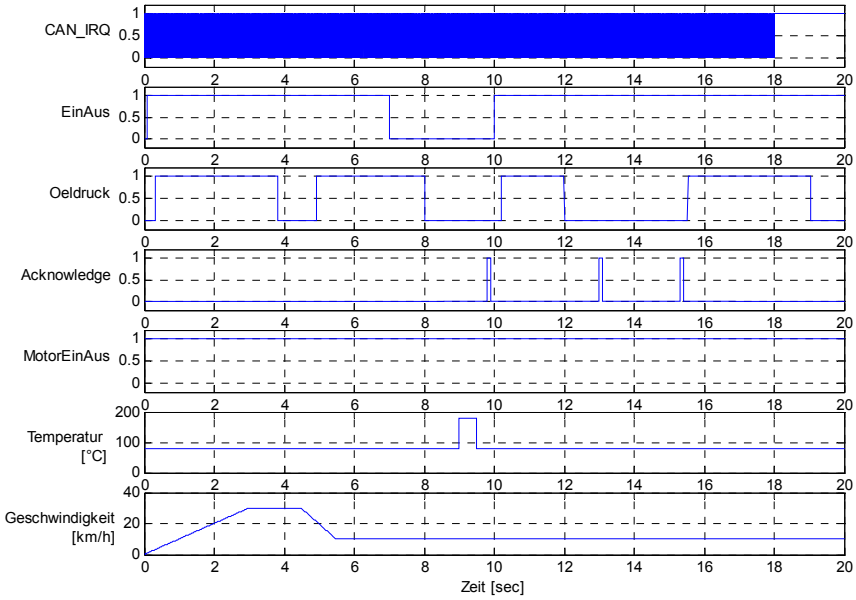

Abbildung 7.19: Signale zur Stimulation der entwickelten Steuer-Logik (mehrere Szenarien)

Dann erfolgt ein gewöhnlicher Einschalt-Zyklus (bei 10 sec), der durch einen abrupten Druckabfall zum Zeitpunkt von 12 sec die Steuer-Logik wieder in den Störzustand überführt. Alle Ventile und Kupplungen werden sofort deaktiviert.

Nach dem Acknowlege Signal bei 13 sec erfolgt ein erneuter automatischer Einschalt-Zyklus, der jedoch bei ca. 15 sec der Simulationszeit zu einer Störung führt.

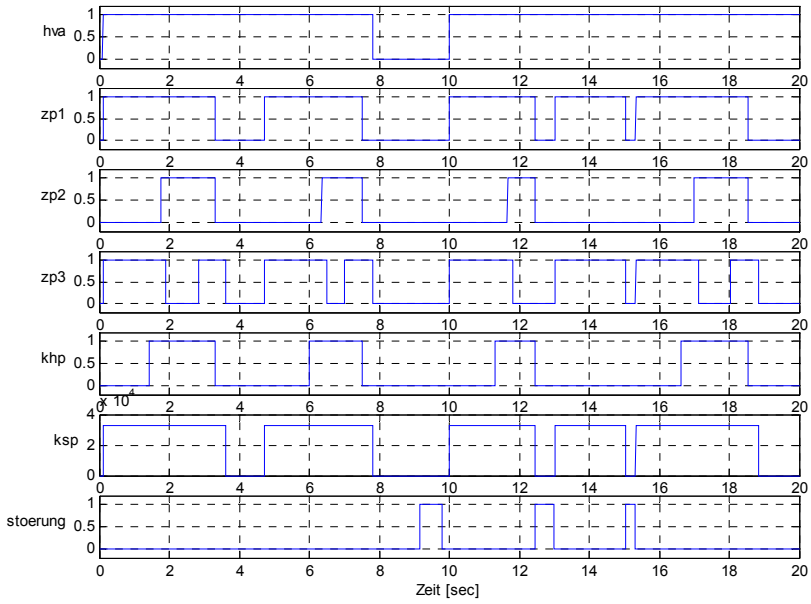

Abbildung 7.20: Signale zur Stimulation der entwickelten Steuer-Logik (mehrere Szenarien)

Dies erfüllt die Anforderung, dass nach dem Aktivieren der Speisepumpe-Kupplung innerhalb einer bestimmten oberen Zeitgrenze der Druck aufgebaut sein muss, ansonsten wird ein Fehler im Drucksystem vermutet und signalisiert.

Durch ein erneutes Acknowlege Signal erfolgt letztendlich ein erfolgreicher Einschalt-Zyklus.

Nach 18 sec Simulationszeit wird der Abbruch der CAN-Kommunikation simuliert, indem das CAN-Interrupt-gesteuerte System nicht mehr getriggert wird (keine Flanken des Signals *CAN_IRQ*). Entsprechend der Spezifikation soll in diesem Fall ein automatischer Abschalt-Zyklus eingeleitet werden, da keine Aussagen über mögliche Grenzwertüberschreitungen der Temperatur oder der Geschwindigkeit getroffen werden können. Da durch Überlastung des CAN-Busses auch nur kurzzeitige Verdrängungen bzw. Verzögerungen entstehen können, soll keine Störung in der Steuerung signalisiert werden, sondern beim Eintreffen neuer Daten die Funktion wieder aufgenommen werden.

7.2.3 Implementierung und Realzeit-Analyse

Nachfolgend wird die Implementierung des beschriebenen Simulations-Modells der HVA-Steuerung beschrieben.

Dabei werden zuerst mittels Co-Simulation die Ausführungszeiten der verschiedenen Aktionen im System bestimmt, um eine anschließende Realzeit-Analyse durchzuführen.

Als letzter Schritt der Umsetzung wird die Anpassung des Target-Modells in das endgültige RT-Modell beschrieben.

Da das realisierte Design bei dem gewählten Zielsystem keine hohe Auslastung verursacht, treten keine Realzeit-Verletzungen auf.

Co-Simulation

Zur Co-Simulation wird das Simulations-Modell aus Abbildung 7.18 in ein PC-Modell (Abbildung 7.21) und ein Target-Modell (Abbildung 7.22) aufgeteilt.

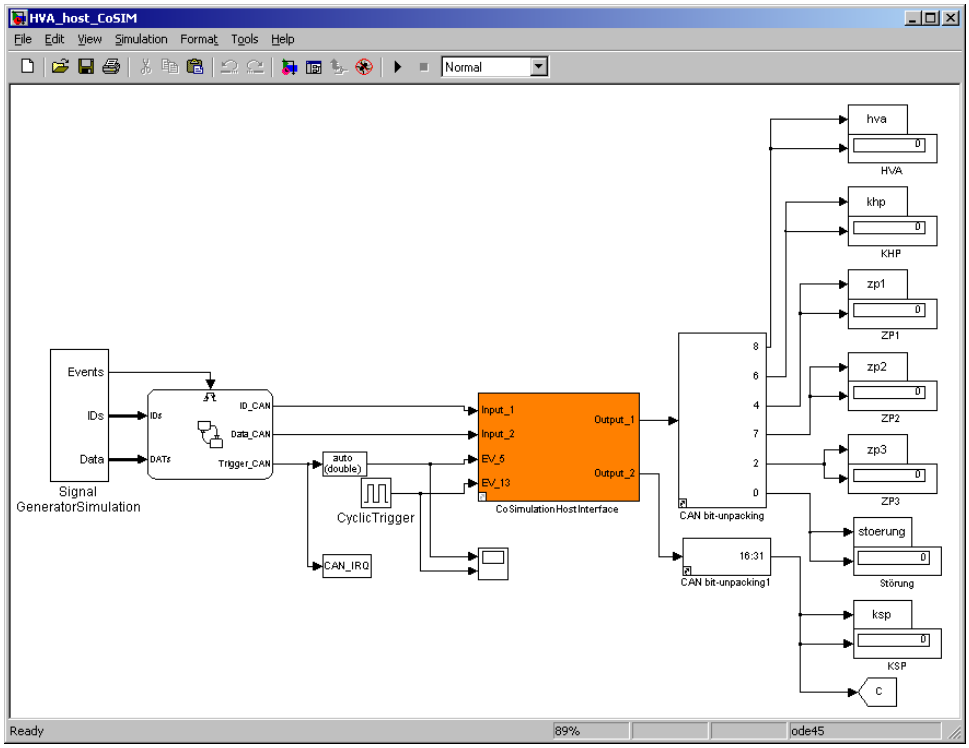

Abbildung 7.21: PC-Modell zur Co-Simulation der HVA-Steuerung

Da die Daten des einbettenden Prozesses (LKW) über CAN-Nachrichten ausgetauscht werden, sind vier Signale für die Kommunikation notwendig. *Input_1* entspricht dem jeweils empfangenen Identifier und *Input_2* gibt den dazugehörigen Daten-Teilen wieder. *Output_1* und *Output_2* sind die Daten-Inhalte der zwei CAN-Nachrichten an die Aktoren des LKWs. Alle relevanten Signale können während der Co-Simulation aufgezeichnet werden, um die Übereinstimmung mit der simulierten Funktion zu überprüfen.

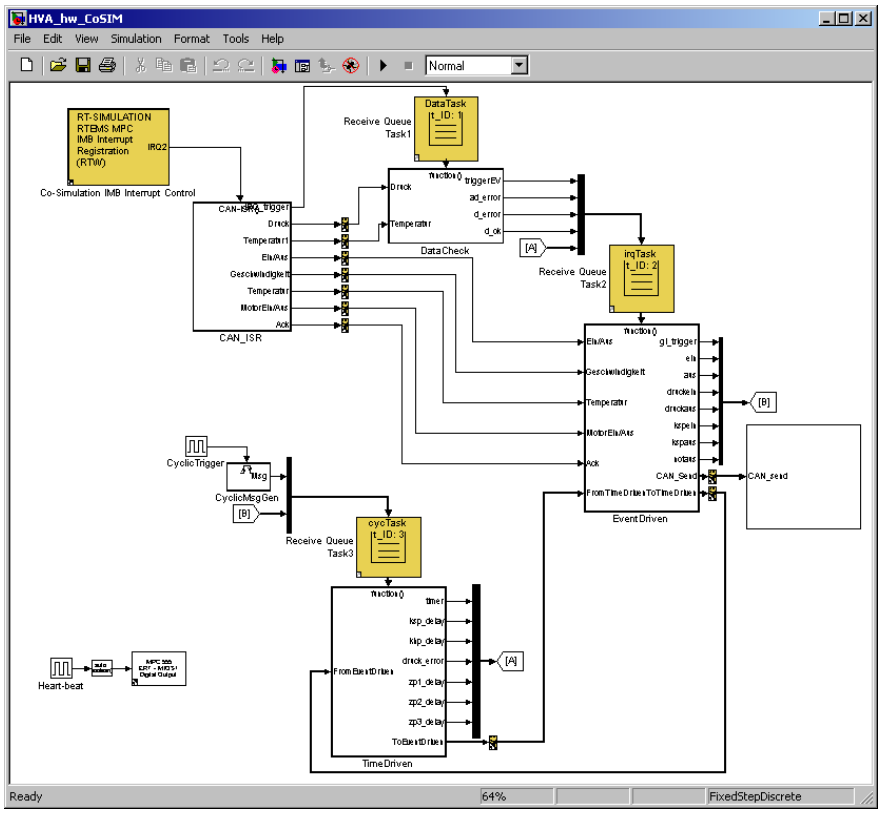

Abbildung 7.22: Target-Modell zur Co-Simulation der HVA-Steuerung

Realzeit-Analyse

Für die Realzeit-Analyse wurden mit den gemessenen Ausführungszeiten und durch Parsen des Target-Modells die folgenden *konkreten Aktions-Präzedenz-Graphen* gewonnen. Das System besitzt zwei externe Ereignisse und zwar den zyklischen Trigger für den *TimeDriven* Server-Thread und den IRQ-Trigger des CAN-Kontrollers (beim Eintreffen einer neuen Nachricht).

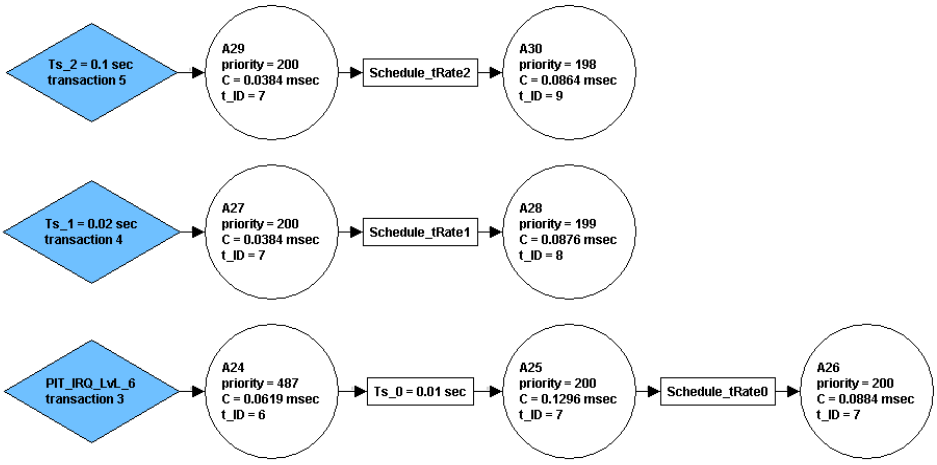

Abbildung 7.23: Konkrete APG der reinen zyklischen Modell-Teile

Die automatisch extrahierten *konkreten APG* des zyklischen Anteils sind in Abbildung 7.23 wiedergegeben. Der zyklische Teil der HVA-Steuerung ist aus drei unterschiedlichen Abtastraten zusammengesetzt. Zum einen wird mit einer Frequenz von 100 Hz (*Ts_0 = 0,01 sec*) der Takt für den *TimeDriven* Server-Thread generiert. Die dazugehörige Message wird in der Aktion *A26* versendet. Zum anderen werden mit einer Frequenz von 50 Hz (*Ts_1 = 0,02 sec*) die CAN-Nachrichten zu den Aktoren gesendet und mit der langsamsten Wiederholungsrate von 10 Hz (*Ts_2 = 0,1 sec*) wird ein Blink-Signal generiert, das als Lebenszeichen der Software dient.

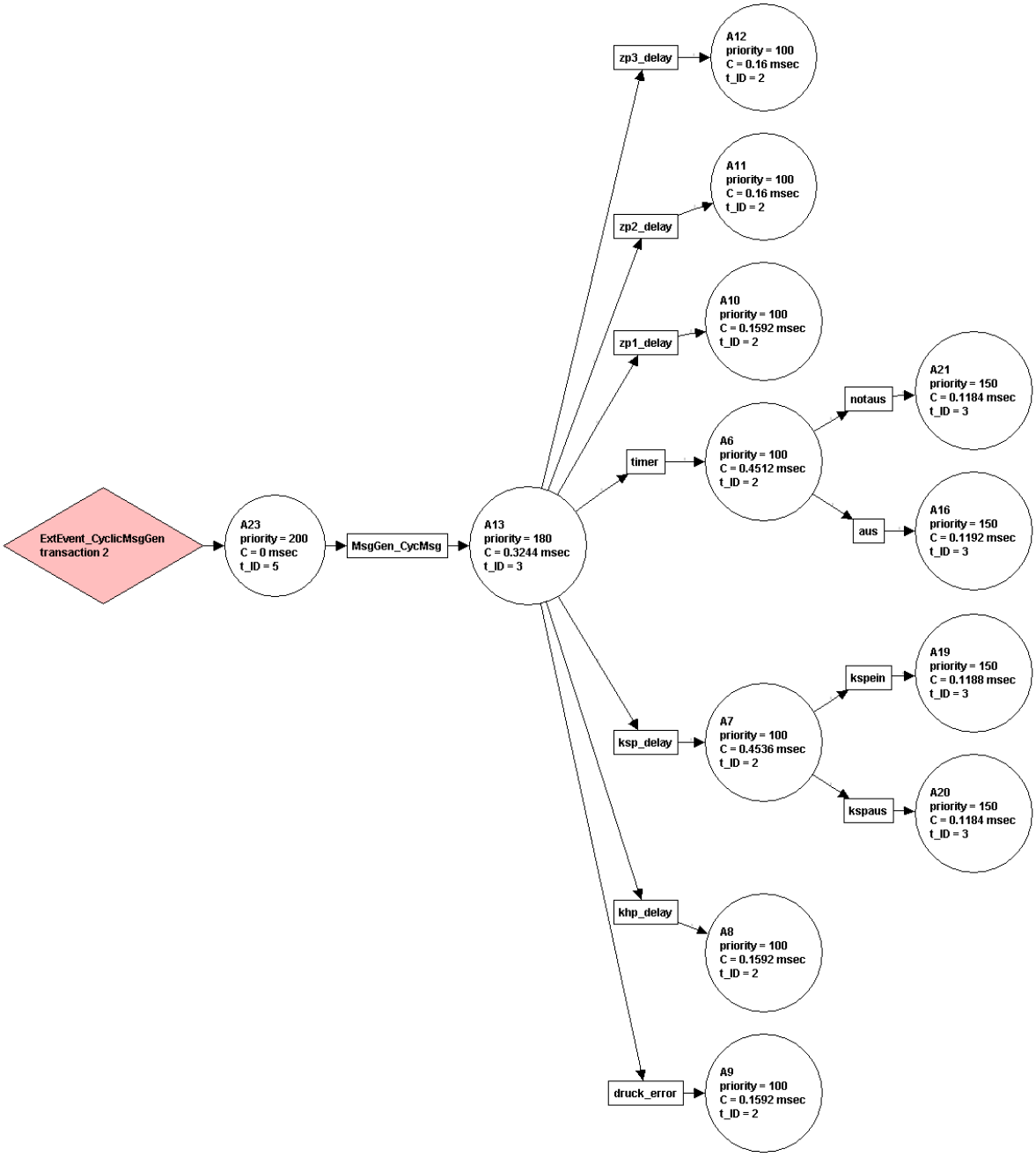

Abbildung 7.24: Konkreter APG des CyclicMsgGen externen Events (zyklisch getriggert)

Der *konkrete APG* des zyklischen Triggers für den *TimeDriven* Server-Thread ist in Abbildung 7.24 dargestellt. Die Ausführungszeit der ersten Aktion *A23* ist mit 0 msec definiert, da die notwendigen Berechnungen zum Versenden des Ereignisses bereits im zyklischen Anteil des Modells bei Aktion *A26* berücksichtigt werden. Man kann sehr schön erkennen, dass beim Eintreffen des zyklischen Takts (*MsgGen_CycMsg*) jeweils das Ablaufen einer bestimmten Ver-

zögerung dem Server-Thread (*t_ID = 2*) durch ein bestimmtes Nachfolge-Ereignis mitgeteilt wird.

Abbildung 7.25 beinhaltet den *konkreten APG* zur externen Ereignis-Quelle *IRQ_2* (CAN-IMB-Interrupt). Abhängig von den Daten generiert der *DataCheck* Server-Thread (*t_ID = 1*) entsprechende Nachfolge-Ereignisse, die durch den *EventDriven* Server-Thread weiterverarbeitet werden.

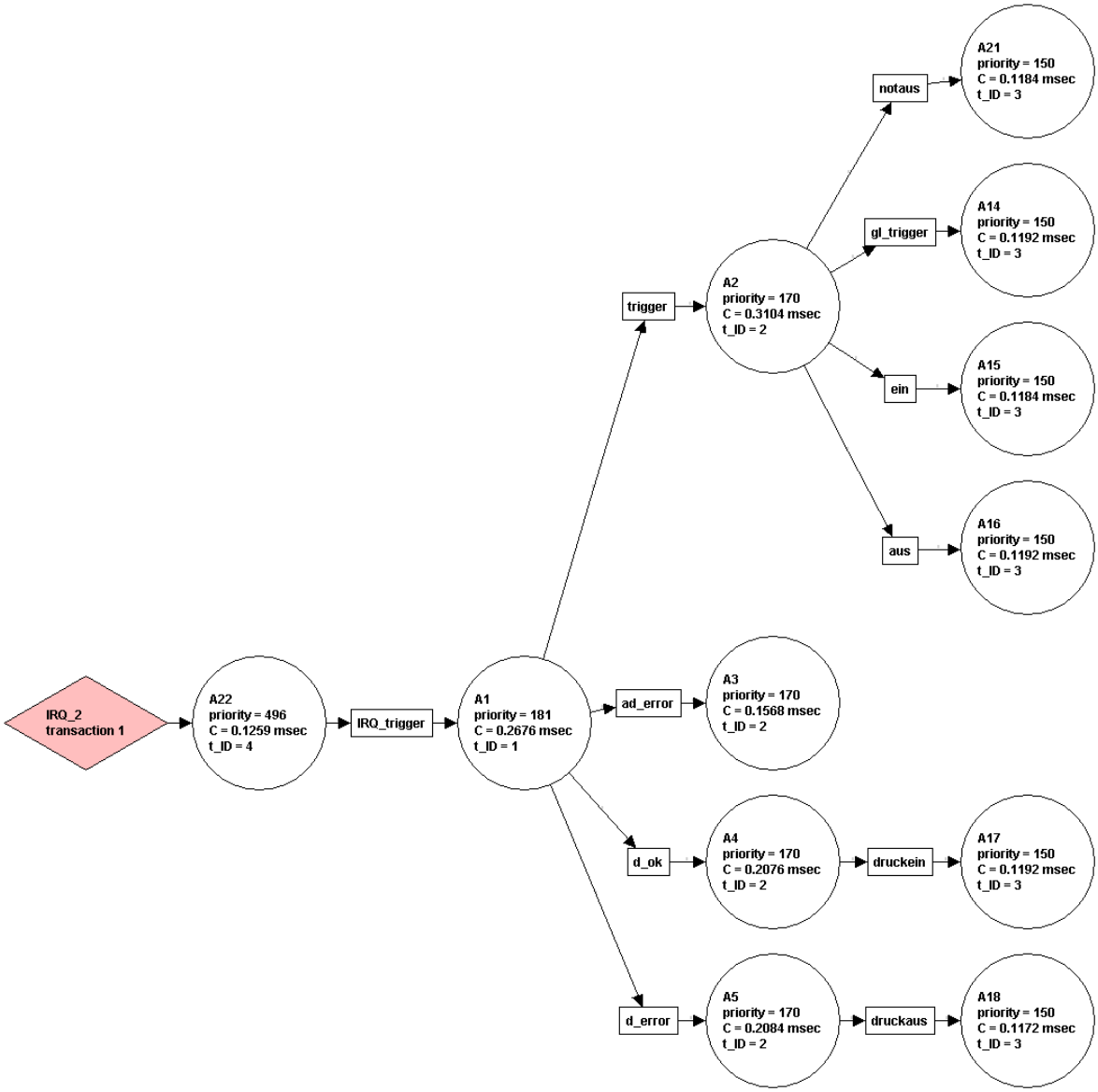

Abbildung 7.25: *Konkreter APG* des *CAN_ISR* getriggerten externen Events

Dieser sendet wiederum die Start-Signale für die verzögerten Rückmeldungen an den *TimeDriven* Thread, da nur dieser durch ein zeitgesteuertes Ereignis getriggert wird, auch wenn keine CAN-Interrupts mehr auftreten.

So kann gewährleistet werden, das der Großteil der HVA-Steuerung (*DataCheck* und *EventDriven*) nur dann bearbeitet wird, wenn auch tatsächlich Daten über den CAN-Bus eingetroffen sind, oder ein Timeout bzw. eine gewünschte Verzögerung abgelaufen ist.

Zur Berechnung der worst-case Reaktionszeiten müssen für das einbettende System noch die Rechenzeitanforderungen in Form von Ereignisströmen definiert werden.

Aus der Spezifikation der CAN-Nachrichten kann entnommen werden, dass die zwei Empfangs-Nachrichten mit einer Periode von 10 msec versendet werden, der zeitliche Abstand zwischen den Nachrichten ist mit 1 msec spezifiziert (Burst von CAN-Messages über den aktuellen Status des LKWs). Die Abbildung dieses Sachverhaltes auf die eine Ereignisstromfunktion sieht wie folgt aus:

$$ES_{IRQ2}^{transaction1} : \left\{ \begin{pmatrix} 0.01\text{sec} \\ 0 \end{pmatrix}, \begin{pmatrix} 0.01\text{sec} \\ 0.001\text{sec} \end{pmatrix} \right\}$$

Die resultierende Ereignisfolge und Ereignisfunktion der *Transaktion 1* sind in Abbildung 7.26 dargestellt. Dabei werden die Vorteile dieser Schreibweise sichtbar, da beliebige Anregungs-Muster beschrieben werden können.

Durch den zyklischen Takt von 100 Hz des *ExtEvent_CyclicMsgGen* Ereignisses ist dessen Ereignisstromfunktion wie folgt definiert

$$ES_{CyclicMsg}^{transaction2} : \left\{ \begin{pmatrix} 0{,}01\text{sec} \\ 0 \end{pmatrix} \right\}$$

Ähnliche Ereignisstrom-Beschreibungen ergeben sich für die anderen drei zyklischen Transaktionen aus Abbildung 7.23 (siehe Gleichung 4.3 in Kapitel 4.1.4).

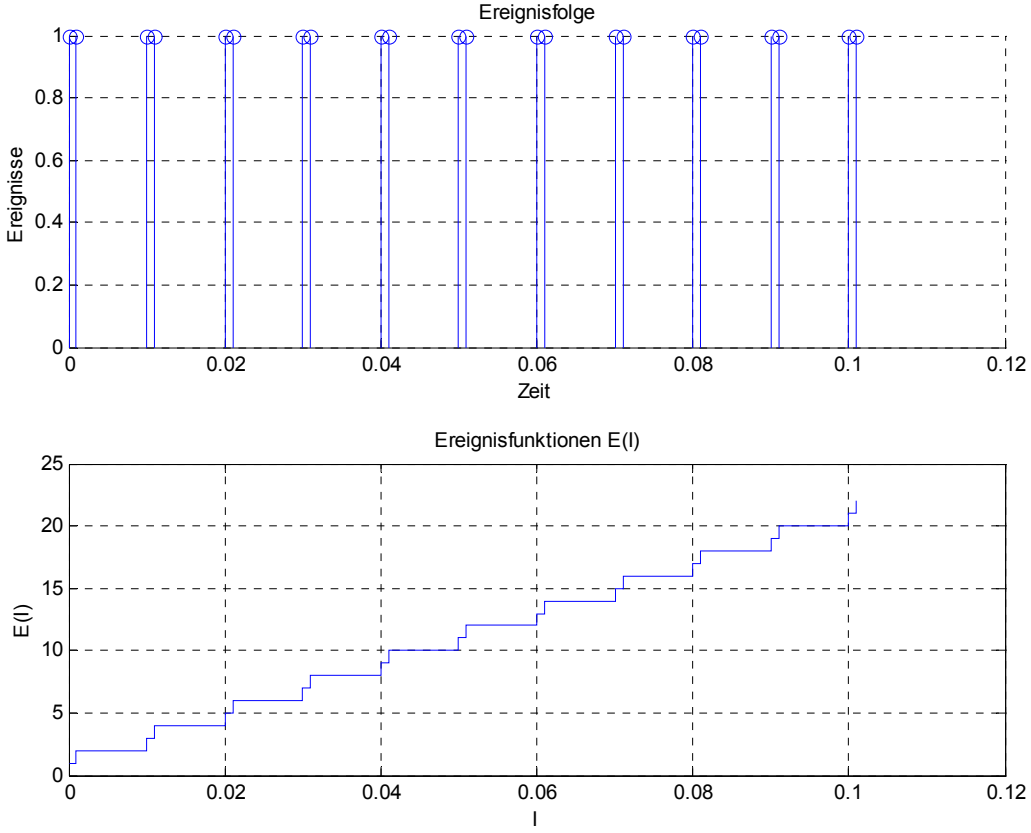

Abbildung 7.26: Rechenzeitanforderung für *Transaktion 1* des einbettenden Systems

Die Ergebnisse der worst-case Reaktionszeit-Berechnung für Aktion *A21* (Notaus) sind in Tabelle 7.2 zusammengefasst.

Die Ergebnisse sind in diesem Beispiel unabhängig vom gewählten Prioritätsvererbungs-Protokoll, da beim BPI-Protokoll keine „ungünstigen" Verkettungen von Blockierzeiten über Thread-Genzen hinweg auftreten können.

Tabelle 7.2: Berechnete Reaktionszeiten für Aktion A21 (Notaus) der umgesetzten HVA-Steuerung

Ereignisstrom *ES*	**Reaktionszeiten von Aktion *A21* [sec]**	
	Transaktion 1	**Transaktion 2**
$ES_{IRQ2}^{transaction1} : \left\{\begin{pmatrix}0.01\text{sec}\\0\end{pmatrix}, \begin{pmatrix}0.01\text{sec}\\0.001\text{sec}\end{pmatrix}\right\}$ $ES_{CyclicMsg}^{transaction2} : \left\{\begin{pmatrix}0.01\text{sec}\\0\end{pmatrix}\right\}$ zyklischer Anteil wird automatisch durch die Abtastraten (Ts) berücksichtigt	0.00564 für q = 1 0.00476 für q = 2 worst-case: *0.00564*	*0.00576*

RT-Modell

Das endgültige RT-Modell der HVA-Steuerung ist in Abbildung 7.27 wiedergegeben. Im Unterschied zu Abbildung 7.22 wurden der Block zur Interrupt-Anbindung (rechts oben: IMB Interrupt Control) und die Blöcke zum Empfangen und Senden von CAN-Nachrichten in den entsprechenden Subsystemen durch reale HW-Treiber-Blöcke ersetzt.

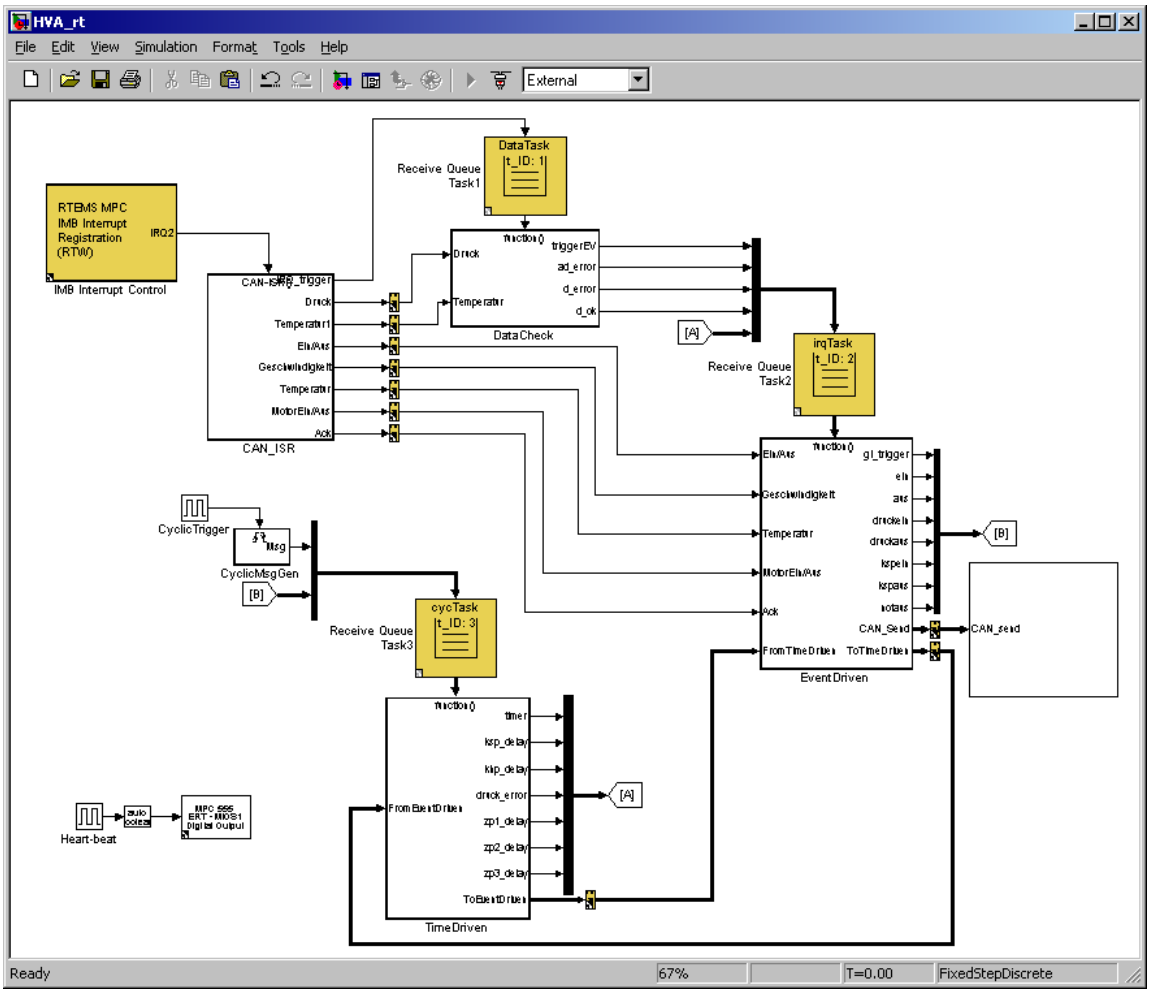

Abbildung 7.27: RT-Modell der HVA-Steuerung

Die Grundstruktur der Steuerung wurde nach eingehenden Simulationen im realen LKW getestet. Für Demonstrationszwecke wurde dem Lehrstuhl für Realzeit-Computersysteme ein speziell programmiertes Steuergerät zur Verfügung gestellt, mit dem die Emulation der LKW-Signale mittels Schalter und Potentiometern möglich ist.

Das Steuergerät sendet wie im LKW die vereinbarten Nachrichten über den CAN an das MPC555 Zielsystem, auf dem das automatisch abgebildete graphische RT-Modell läuft. Die Aktor-Signale, die von der entwickelten Steuer-Software zurückkommen, werden durch Leuchtdioden am Steuergerät angezeigt. Damit waren weitere Tests auch ohne realen LKW möglich.

7.2.4 Ergebnisse und Erfahrungen

Die Umsetzung der HVA-Steuerung hat bewiesen, dass auch umfangreichere Applikationen realisiert und untersucht werden können. Durch die Notwendigkeit der Anregung aller Aktionen der Software-Architektur während der Co-Simulation wurde auch die Wichtigkeit vollständiger Test-Szenarien sichtbar. Diese Szenarien müssen bereits bei der Modellierung und dem Design der gewünschten Funktionalität als integraler Teil des modell-basierten Entwurfsprozesses entstehen. Nur so kann die geforderte Übereinstimmung der ausführbaren Spezifikation mit der gewünschten Funktionalität gewährleistet werden.

Die Co-Simulation stellt in einem solchen konsequent durchgeführten modell-basierten Entwurfsprozess keinen Mehraufwand dar, da sowohl beide benötigten Modell-Teile (PC- und Target-Modell) im endgültigen Simulations-Modell enthalten sind als auch die zur Stimulation aller Aktionen notwendigen Test-Szenarien zur Verfügung stehen.

Mögliche Lücken und Unstimmigkeiten bei den Abhängigkeiten zwischen Ereignissen und Aktionen können durch die neue Sicht auf das System, die durch die Generierung der Aktions-Präzedenz-Graphen entsteht, erkannt und durch Änderungen des Designs beseitigt werden.

All diese Erfahrungen haben gezeigt, dass durch das Einfügen der Co-Simulation und der Realzeit-Analyse im modell-basierten Entwurfsprozess nicht nur Aussagen zur Auslastung des Systems getroffen werden können, sondern auch tiefere Einsichten in die Zusammenhänge z.B. von bestimmten Reaktionszeiten entstehen. Dieses Wissen ist eine grundlegende Voraussetzung für eine Optimierung.

Ausblick

In der vorliegenden Arbeit wurden die Grundlagen für eine weitgehend automatisierte Abbildung hybrider graphischer Spezifikationen auf eine echtzeitfähige Implementierung vorgestellt. Dazu wurde eine neue Software-Architektur erarbeitet, die neben der Berücksichtigung von zustandsbasierten und signalflussorientierten Laufzeitmodellen auch die Abbildung auf zyklische und auf ereignisgesteuerte Software-Teile unterstützt. Damit kann eine effiziente, ressourcenschonende Abbildung auf Zielsysteme mit beschränkter Rechenleistung durchgeführt werden.

Zur Berechnung des zeitlichen worst-case Verhaltens einer konkreten Umsetzung eines graphischen Modells wurde für die entwickelte SW-Architektur und die im Laufzeitsystem umgesetzten Vererbungsstrategien (*Basic-Priority-Inheritence* und *Preemption-Threshold*) auch der notwendige mathematische Formalismus entwickelt. Er ermöglicht eine weitgehend automatisierte Realzeit-Analyse.

Die Kommunikationsstrukturen zwischen Aktionen in einem zu analysierenden System werden durch *allgemeine* und *konkrete Aktions-Präzedenz-Graphen (APG)* beschrieben. Der derzeitige Implementierungsstand erlaubt die Beschreibung *konkreter APG,* unterstützt aber nicht die Definition von ODER Pfaden. Bei einer Weiterentwicklung der Werkzeugunterstützung wäre daher wünschenswert, beim Parse-Vorgang des Modells auch ODER Bedingungen beschreibbar zu machen, um dem Entwickler die Möglichkeit der Spezifikation exklusiver Pfade zu bieten. Die Transformation der *allgemeinen APG* auf *konkrete APG* könnte dann bei der Analyse automatisiert durchgeführt werden.

Ein weitere Verbesserung des Parse-Vorgangs ist durch die Analyse der konkret modellierten Funktion erreichbar. Die Fragen zu möglichen Nachfolge-Ereignissen könnten dann gezielter durch das Werkzeug im Vorfeld ermittelt werden und den Entwickler entlasten. Dies würde jedoch eine eingehende Analyse der Syntax und der Semantik der Werkzeuge erforderlich machen.

Zur Ermittlung konkreter Ausführungszeiten der letztendlichen Implementierung wurde im Rahmen der Arbeit eine Co-Simulation umgesetzt. Neben der Laufzeitmessung werden während einer Co-Simulation auch Auswirkungen konkret getroffener Implementierungsentscheidungen (z.B. Prioritätenvergabe, etc.) auf die spezifizierte Funktion untersucht. Diese Art der Verifikation wird zur Zeit manuell durchgeführt und sollte in Zukunft durch ein zusätzliches Werkzeug automatisiert werden. Im Rahmen einer solchen Weiterentwicklung wäre auch eine weiterführende Analyse der ermittelten Ausführungszeit-Daten erstrebenswert, da nicht nur die absoluten Werte der einzelnen Messungen Informationen enthalten, sondern auch ihre Auftrittsreihenfolge Rückschlüsse auf die Abarbeitungsreihenfolge der Aktionen in einem konkreten Ausführungsschritt ermöglicht. Dadurch können Unstimmigkeiten im Laufzeitsystem oder mögliche unerwünschte Effekte erkannt werden. Außerdem könnten Optimierungsvorschläge für das untersuchte Modell abgeleitet werden.

Eine Verbesserung der Genauigkeit der Realzeit-Analyse kann hingegen durch die Verwendung zuverlässiger worst-case Ausführungszeiten (WCET), anstelle der im Augenblick verwendeten realistischen Ausführungszeiten, erreicht werden. Hierzu ist die Entwicklung eines Analyse-Werkzeuges denkbar, das aus einer gegebenen graphischen Spezifikation die worst-case Anregungen ermittelt. Diese Anregungen können dann als spezielle Testfälle mit in die Co-Simulation aufgenommen werden und die Messung konkreter worst-case Ausführungszeiten ermöglichen.

Ein Anhaltspunkt über die kleinste zulässige Tiefe der Warteschlangen in einem konkreten Software-Design könnte in Zukunft durch die weiterführende Auswertung der Ergebnisse des RT-Nachweises bringen. In der augenblicklichen Umsetzung wird die Anzahl der Instanzen einer untersuchten Aktion im *Busy-Window* isoliert zur Ermittlung der worst-case Reaktionszeit verwendet. Es ist jedoch möglich, die Summe der Anzahl aller Instanzen von Aktionen eines bestimmten Server-Threads (Ereignis-Warteschlange) als Angabe über die maximale Anzahl an gleichzeitig wartenden Ereignissen in der Queue zu verwenden. Somit könnten die dafür notwendigen Ressourcen optimal durch eine mathematische Beziehung bestimmt werden.

Eine weitere Verbesserung der aktuellen Umsetzung wurde bereits in Abschnitt 5.1.6 beschrieben und betrifft die Divergenz zwischen Simulations-Verhalten und letztendlichem RT-Verhalten der Modellumsetzung. Zur Zeit können Unterschiede während der Co-Simulation sichtbar gemacht werden. Durch die Einführung eines weiteren Zwischenschrittes bei der Konkretisierung könnten ungünstige Auswirkungen von Implementierungsentscheidungen auf die Funktion des Designs bereits früher in der Entwicklung aufgezeigt werden. Die Bereitstellung einer sogenannten RTOS-Simulation könnte Phänomene wie Ereignis-Überholung oder Prioritätsvererbungen auch ohne direkte Abbildung auf die Zielhardware (wie in der derzeitigen Co-Simulation) simulieren und Auswirkungen auf das Verhalten wiedergeben.

Literatur

Roadmaps und Trends bei eingebetteten Systemen

[1] Ludwig D.J. Eggermont (ed.), *„Embedded Systems Roadmap 2002, Vision on technology for the future of PROGRESS“*,30 March 2002, www.stw.nl/progress/Esroadmap/ESRversion1.pdf

[2] H. Kopetz, *„Research issues in Dependable Embedded Systems”*, IFIP WG 10.4, Jan. 2002

[3] *“IT-Forschung 2006 Förderprogramm Informations- und Kommunikations-Technik“*, Bundesministerium für Bildung und Forschung (BMBF), Bonn, März 2002 (German National ICT Research Programm IT 2006)

[4] C. le Pair (ed.), *„Samen, strategische en sterker“*, Rapport, Task Force ICT-en-Kennis, Lange Voorhout 7, 2514 EA Den Haag, Jul 2001.

[5] M. Saksena and B. Selic. “Real-Time Software Design: State of the Art and Future Challenges”, *In IEEE Candian Review*. June 1999. Invited Paper.

[6] J. Sifakis, “Modelling Real-Time Systems – Challenges and Work Direktions”, *In Proceedings of the First International Workshop*, EMSOFT 2001, Tahoe City, CA, USA, October 8-10, 2001, LNCS 2211, pp. 373-389, ISBN 3-540-4267-6 Springer-Verlag

Regelungstechnik

[7] R. Isermann, *„Digitale Regelsysteme, Bd. 1: Grundlagen: Deterministische Regelungen.“*, ISBN 3-540-16596-7, Springer-Verlag 1986

[8] Univ.-Prof. Dr.-Ing. G. Schmidt, *„Regelungs- und Steuerungstechnik 1”*, Manukript zur Vorlesung Ausgabe WS 96/97, Lehrstuhl für Steuerungs- und Regelungstechnik Technische Universität München

Software-Engineering allgemein und für RT-Systeme

[9] Univ.-Prof. Dr.-Ing. G. Färber, *„Software Engineering”*, Manuskript zur Vorlesung, Stand SS 2003, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München

[10] B. Selic, G. Gullekson, P. T. Ward, *“Real-Time Object-Oriented Modelling”*, ISBN 0-471-59917-4, John Wiley & Sons, Inc., 1994

[11] P. A. Laplante, *“Real-time systems design and analysis: an engineer’s handbook”*, ISBN 0-7803-0402-0, IEEE Press, 1993

[12] A. Burns, A. Wellings, *“Real-Time Systems and Programming Languages”*, ISBN 0-201-40365-X, Addison Wesley Longman, 1996

[13] A. Burns and A. J. Wellings, “HRT-HOOD: A Structured Design Method for Hard Real-Time Systems”, *Journal of Real-Time Systems*, 6(1):73--114, 1994.

[14] D. J. Hatley, J. Derek, *“Strategien für die Echtzeit-Programmierung“*, ISBN 3-446-16288-7, Carl Hanser Verlag, 1993

[15] W. S. Heath, *„Real-time software techniques”*, ISBN 0-442-00305-6, Van Nostrand Reinhold, 1991

[16] P. T. Ward, *"Structured development of real-time systems"*, ISBN 0-13-854803-X, Prentice-Hall, Inc., 1986

[17] W. Leigh, *"Real time software for small systems"*, ISBN 0-905104-98-6, Sigma Press, 1988

[18] Schulz, *"Methoden des Software-Endwurfs und Strukturierte Programmierung"*, ISBN 3-11-007336-6, de Gruyter Lehrbuch, 1978

[19] J. P. Calvez, *„Embedded real-time systems: a specification and design methodology"*, ISBN 0-471-93563-8, John Wiley & Sons Ltd., 1993

[20] D. D. Gajski, F. Vahid, S. Narayan, J. Gong, *"Specification and Design of Embedded Systems"*, ISBN 0-13-150731-1, Prentice Hall, 1994

[21] S. Bennett, *"Real-time computer control: an introduction"*, ISBN 0-13-764176-1, Prentice Hall, 1994

[22] N. Halbwachs, *"Synchronous programming of reactive systems"*, ISBN 0-7923-9311-2, Kluwer Academic Publishers, 1993

[23] H. Kopetz, *„Real-time systems: design principles for distributed embedded applications"*, ISBN 0-7923-9894-7, Kluwer Academic Publisher, 1997

[24] Kopetz H.. "Event-Triggered versus Time-Triggered Real-Time Systems." *Lecture Notes in Computer Science*, vol. 563, Springer Verlag, Berlin, 1991.

[25] M. Andre, van Tilborg, M. K. Gary, *"Fondations of Real-Time Computing: Scheduling and Resource Management"*, ISBN 0-7923-9166-7, Kluwer Academic Publisher, 1991

[26] F. Thoen, F. Catthoor, *"Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems"*, ISBN 0-7923-7737-0, Kluwer Academic Publisher, 2000

[27] W. A. Halang and A. D. Stoyenko, *"Constructing Predictable Real Time Systems"*, ISBN 0-7923-9202-7, Kluwer Academic Publisher, 1991

[28] M. Schiebe and S. Pferrer, *"Real-Time Systems Engineering and Applications"*, ISBN 0-7923-9196-9, Kluwer Academic Publisher, 1992

[29] Alan C. Shaw, *"Real-time systems and software"*, John Wiley & Sons, Inc., ISBN 0-471-35490-2

[30] Stephen A. Edwards, "Design Languages for Embedded Systems", *In Proceedings of the Second Online Symposium for Electrical*, Engineers (OSEE2), March 2001.

[31] S. A. Edwards, L. Lavagno, Edward A. Lee, and Alberto, Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis", *Proceedings of the IEEE* 85(3):366-390, March, 1997

[32] Stephen A. Edwards, *"The Specification and Execution of Heterogeneous Synchronous Reactive Systems"*, PhD Thesis, University of California, Berkeley, March, 1997.

[33] G. Bucci, M. Campanai, P. Nesi, "Tools for specifying real-time system", *In: Real-Time Systems*, Vol. 8, No. 2-3, pages 117-172, Kluwer Academic Publisher, 1995.

Modellierungs-Sprachen

[34] J. Goguen, J. Meseguer, K. Futatsugi, P. Lincoln, and J. Jouannaud, *„Introducing OBJ"*, TR SRI-CSL-88-8, Computer Science Lab, SRI International, Menlo Park, CA, August 1988

[35] J. Spivey, *„The Z Notation: A Reference Manual"*, Prentice-Hall, Englewood Cliffs, NJ, 1989

[36] F. Jahanian and A. Mok, „Safety analysis of timing properties in real-time systems“, *IEEE Trans. on Software Engineering*, vol. SE-12, no. 9 (Sep. 1986), pp. 890-904

[37] F. Jahanian and A. Mok, „Modechart: A specification language for real-time systems“, *IEEE Trans. on Software Engineering*, vol. 20, no. 12 (Dec. 1994), pp.933-947

[38] C. Hoare, *„Communicating Sequential Processes“*, Prentice-Hall, Englewood Cliffs, NJ, 1984

[39] K. Cardelli and J. Misra, *„Parallel Program Design: A Foundation“*, Addison-Wesley, 1998

[40] R. Gerber and I. Lee, „A layered approach to automating the verification of real-time systems“, *IEEE Trans. on Software Engineering*, vol. 18, no. 9 (Sept. 1992), pp. 768-784

[41] G. Ghezzi, M. Jazayeri, and D. Mandrioli, *„Fundamentals of Software Engineering“*, Prentice-Hall, Englewood Cliffs, NJ, 1991

[42] G. Ghezzi, D. Mandrioli, S. Morasea, and M. Pezze, „A unified high-level Petri net formalism for time-critical systems“, *IEEE Trans. on Software Engineering*, vol. 17, no. 2 (Feb. 1991), pp. 160-172

[43] J. Ostroff and W. Wonham, „Modelling and verifying real-time embedded computer systems“, *Proc. IEEE Real-Time Systems Symp.*, Dec. 1987m pp. 124-132

[44] N. Lynch and M. Tuttle, *„An introduction to input-output automata“*, CWI-Quartely, vol. 2, 1989,; Tech Memo MIT/LCS/TM-373, Laboratory for Computer Science, MIT, Boston, MA, Nov. 1988

[45] A. Shaw, „Communicating real-time state machines“, *IEEE Trans. on Software Engineering*, vol. 18, no. 9 (Sept. 1992), pp. 805-816.

[46] F. Belina, D. Hogrefe and A. Sarma, *“SDL with applications from protocol specification”*, Prentice-Hall, Englewood Cliffs, NJ, 1991, ISBN 0-13-785890-6

[47] G. Booch, J. Rumbaugh, and I. Jacobson, *“The Unified Modelling Language User Guide”*, ISBN 0-201-57168-4, Addison Wesley object technology series, October 1998

CASE-Tools:

[48] K. Spurr and P. Layzell, *“CASE on Trial”*, ISBN 0-471-92893-3, John Wiley & Sons Ltd., 1990

[49] W. Dröschel, *“CASE Tools, Werkzeugunterstützung im Rahmen des V-Modells”*, ISBN 3-486-23239-8, Oldenbourg, 1995

[50] G. Müller-Ettrich, *“Objektorientierte Prozeßmodelle: UML einsetzen mit OOTC, V-Modell, Objectory“*, ISBN 3-8273-1364-3, Addison Wesley Longman, 1999

[51] D. M. Gee, B. P. Worrall and W.D. Henderson, “Real-time CASE Tools: A Review of Current Tools and Future Prospects”, *In CASE: Current Practice, Future Prospects*. Edited by Kathy Spurr and Paul Layzell, 1992 John Wiley & Sons Ltd, ISBN 0-471-93304-X, pp 67-84

[52] M. Beeby, J. S. Parkinson and V. Hamilton, “Integrated Support for Real-time Control Systems”, *In CASE: Current Practice, Future Prospects*. Edited by Kathy Spurr and Paul Layzell, 1992 John Wiley & Sons Ltd, ISBN 0-471-93304-X, pp107-120

[53] K. Lyytinen and V.-P. Tahvanainen, “Introduction: towards the next generation of Computer Aided Software Engineering (CASE)”, *In Next Gereration CASE Tools*, IOS Press, 1992, ISBN: 90-5199-076-6, pp 1-7

[54] Ivan Aaen, “CASE tool Bootstrapping – how little strokes fell great oaks”, *In Next Gereration CASE Tools*, IOS Press, 1992, ISBN: 90-5199-076-6, pp 8-17

[55] M. Broy, F. Huber und B. Schätz, *„AutoFocus - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme“*, Informatik Forschung und Entwicklung 14(3):121-134, 1999

[56] Edward A. Lee, *"Overview of the Ptolemy Project",* Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.

[57] S. A. Scheider, V. W. Chen, G. Pardo-Castellote, H. H. Wang, *“ControlShell: a software architecture for complex electromechanical systems”*, International Journal of Robotics Research, 4(17), 360-380, 1998.

Statecharts

[58] D. Harel, “Statecharts: A Visual Formalism For Complex Systems,” *Science of Computer Programming 8*, pp. 231-278, 1987.

[59] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. “On the Formal Semantics of Statecharts” *Proceedings Symposium on Logic in Computer Science*, pages 54--64, 1987.

[60] D. Harel, and A. Naamad, “The STATEMATE Semantics of Statecharts”, *ACM Trans. Soft. Eng. Method.*, 5:4, Oct. 1996.

[61] i-Logix, *“STATEMATE: The Semantics of Statecharts”*, user manual version 3.0, July 1990, i-Logix

[62] M. von der Beeck, “A comparison of Statecharts variants”, In Formal Techniques in Real-Time and Fault-Tolerant Systems: *Third International Symposium Proceedings*, volume 863 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[63] Peter Scholz, *„Design of Reactive Systems and their Distributed Implementation with Statecharts“*, PhD thesis, Technische Universität München, 1998.

Formale Semantik und Verifikation von Statecharts

[64] M. von der Beeck, "A concise compositional Statecharts semantics definition", *In FORTE/PSTV 2000*. Kluwer, 2000.

[65] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel, „On formal semantics of Statecharts as supported by statemate”, *In 2nd BCS-FACS Northern Formal Methods Workshop*, Springer-Verlag, July 97.

[66] J.-R. Beauvais, E. Rutten, T. Gautier, R. Houdelbine, P. le Guernic, and Y.-M Tang, “Modeling Statecharts and Activitycharts as Signal Equations”, *ACM Trans. Soft. Eng. Method.*, Vol. 10, No. 4, Oct. 2001, Pages 397-451.

[67] A. Tiwari and N. Shankar and J. Rushby, “Invisible formal methods for embedded control systems”, To be submitted to *Proceedings of the IEEE*, Special issue on Embedded Systems, 2002.

MATLAB, Simulink, Stateflow

[68] Lehrstuhl für Elektrische Antriebssysteme, *„Simulation mit Simulink / Matlab“*, Skriptum mit Übungsaufgaben, Januar 2001, www.eat.ei.tum.de

[69] MathWorks Automotive Advisory Board (MAAB), *“MAAB Controller Style Guidelines For Production Intent”*, Release V1.00, The MathWorks, Inc., Natick, MA, Released April 2001. http://www.mathworks.com/products/controldesign/maab.shtml

[70] The MathWorks, Inc., *“Stateflow, For State Diagram Modeling”*, User’s Guide Version 4, The MathWorks, Inc., Natick, MA, Sept. 2000.

[71] The MathWorks, Inc., *"Simulink, Dynamic System Simulation for Matlab"*, Using Simulink Version 4, The MathWorks, Inc., Natick, MA, Nov. 2000.

[72] The MathWorks, Inc., *"MATLAB, The Language of Technical Computing"*, Using MATLAB Version 6, The MathWorks, Inc., Natick, MA, Nov. 2000.

[73] The MathWorks, Inc., *"Target Language Compiler, For Use with Real-Time Workshop"*, Reference Guide Version 4, The MathWorks, Inc., Natick, MA, Sep. 2000.

Modell-basiertes Design

[74] Martin Orehek and Christian Robl, "Model-Based Design of an ECU with Data and Event-Driven Parts Using Auto Code Generation", In *Proc. IEEE Conf. on Robotics and Automation ICRA 2001*, Seoul, Korea, 2001.

[75] Martin Orehek and Christian Robl, "Model Based Design in the Development of Thermodynamic Systems and their Electronic Control Units", In *Proc. IEEE Conf. on Control Applications 2002*, Glasgow, Scottland, U.K. 2002.

[76] Martin Orehek, "Process Pattern: Model Based Real-Time Systems Development", In *Proceedings of the 1st Workshop on Software Development Patterns*, SDPP'02, Technische Universität München, Institut für Informatik, Dez. 2002, www.forsoft.de/zen/sdpp02/proceedings/TUM-I0213.pdf

[77] Andreas Hecker, *„Entwicklung einer Softwarearchitektur zur Messung von Ausführungszeiten in einer Simulationsumgebung"*, Diplomarbeit ausgeführt am Lehrstuhl für Realzeit-Computersysteme, Mai 2003

[78] A. Rau, "Potential and Challenges for Model-based Development in the Automotive Industry", in "Business Briefing: Global Automotive Manufacturing and Technology", World Market Research Center, pages 124-138, Oktober 2000.

[79] D. Buck and A. Rau, "On Modelling Guidelines: Flowchart Patterns for STATEFLOW", in *Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends*, 21(2), August 2001, ISSN 0720-8928, http://pi.informatik.uni-siegen.de/stt.

[80] A. Rau, *„Modellbasierte Entwicklung von eingebetteten Regelsystemen in der Automobilindustrie"*, Dissertation an der Fakultät für Informatik der Universität Tübingen, Juni 2002 Erschienen im Verlag www.dissertation.de, Mai 2003, ISBN 3-89825-599-9.

[81] K. Bender, M. Broy, I. Péter, A. Pretschner, T. Stauner, „Model based development of hybrid systems: specification, simulation, test case generation", In *Modelling, Analysis, and Design of Hybrid Systems*, Lecture Notes in Control and Information Sciences, Vol. 279, pp. 37-52, July 2002, Springer Verlag

[82] G. Hahn, J. Philipps, A. Pretschner, T. Stauner, "Tests for mixed discrete-continuous systems", *Technical Report TUM-I0301*, Institut für Informatik, TU München, January 2003

[83] B. Schätz, A. Pretschner, F. Huber, J. Philipps, "Model-Based Development of Embedded Systems", In *Advances in Object-Oriented Information Systems*, Lecture Notes in Computer Science, Vol. 2426, pp. 298-311, (Proc. Workshop Model-Driven Approaches to Software Development, Montpellier, September 2002), Springer Verlag

[84] S. Ranville and P. E. Black, "Automated Testing Requirements - Automotive Perspective", *The Second International Workshop on Automated Program Analysis, Testing and Verification*, Toronto, Canada (May 2001).

Regelungstechnik und Scheduling

[85] A. Cervin, D. Henriksson, B. Lincoln and K.-E. Årzén: “Jitterbug and TrueTime: Analysis Tools for Real-Time Control Systems." In *Proceedings of the 2nd Workshop on Real-Time Tools*, Copenhagen, Denmark, 2002

[86] D. Henriksson, A. Cervin and K.-E. Årzén: “TrueTime: Simulation of Control Loops Under Shared Computer Resources." In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, 2002

[87] J. Eker and A. Cervin: “A Matlab Toolbox for Real-Time and Control Systems Co-Design." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 320--327, Hong Kong, P.R. China, 1999

[88] A. Cervin: “Improved Scheduling of Control Tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 4--10, York, UK, 1999

[89] Anton Cervin: *“Integrated Control and Real-Time Scheduling“*. PhD thesis ISRN LUTFD2/TFRT--1065--SE, apr 2003. Department of Automatic Control, Lund Institute of Technology, Sweden

[90] D. Henriksson and A. Cervin: *“TrueTime 1.1---Reference Manual.“* Technical Report ISRN LUTFD2/TFRT--7605--SE, mar 2003 Department of Automatic Control, Lund Institute of Technology, Lund, Sweden

[91] A. Cervin and B. Lincoln: *“Jitterbug 1.1---Reference Manual.“* Technical Report ISRN LUTFD2/TFRT--7604--SE, jan 2003 Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[92] A. Cervin: *“Towards the Integration of Control and Real-Time Scheduling Design.“* Lic Tech thesis ISRN LUTFD2/TFRT--3226--SE, may 2000. Department of Automatic Control, Lund Institute of Technology, Sweden

[93] Anton Cervin: *“The Real-Time Control Systems Simulator---Reference Manual.“* Technical Report ISRN LUTFD2/TFRT--7592--SE, apr 2000 Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[94] T. A. Henzinger, B. Horowitz, C.M. Kirsch, „Giotto: A Time-Triggered Langauge for Embedded Programming“, in *Proc. 2nd Int. Workshop Embedded Software (EMSOFT),* LNCS 2491, Springer Verlag, 2002, pp. 76-92

[95] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido and W. Pree, „From Control Models to Real-Time Code Using Giotto“, In *IEEE Control Systems Magazine*, Feb. 2003

Realzeit-Analyse (Scheduling und *Rate-Monotonic-Analysis)*

[96] C. Liu, J. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real Time Environment", *Journal of the Association for Computing Machinery* 20(1):46-61, JACM, 1973.

[97] M. Joseph, and P. Pandya, “Finding Response Times in a Real-Time System”, *The Computer Journal*, Volume 29, No. 5, pp. 390-395, 1986

[98] L. Waszniowski, Z. Hanzalek, “Performance Analysis of Real-Time Systems Using RMA”, In: *International Summer School of Automation*, Maribor: University of Maribor, 2001, p. 103-108. ISBN 86-435-0411-4.

[99] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *11th Real-Time Systems Symposium*, pp. 201--209, Dec. 1990.

[100] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems", *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55--67, January 1994.

[101] K. W. Tindell, A. Burns, and A. Wellings, "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks," *YCS 189*, Department of Computer Science, University of York (1992)

[102] K. Tindell, “Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets”, *Dept of Computer Sceience, University of York* (Aufust 1992).

[103] N. Tracey, “True Real-time Embedded Systems Engineering, Building hard real-time systems using Deadline Monotonic Analysis”, *ESC Europe*, Maastricht, November 2000, Realogy, York, England, www.realogy.com

[104] L. Sha, R. Rajkumar, and J. Lehoczky. „Priority Inheritance Protocols: An Approach to RealTime Synchronization“. *IEEE Transactions on Computers*, 39(9):1175--1185, September 1990.

[105] N. C. Audsley, “Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times”, *YCS 164*, Dept. Computer Science, University of York (December 1991).

[106] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. “Hard real-time scheduling: the deadlinemonotonic approach”. In *Proc. of the Eighth RTOSS* , May 1991. A. K. Atlas and A. Bestavros. Statistical rate monotonic scheduling. Technical Report BUCS-TR-98-010, Boston University, 1998.

[107] Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A., "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal* (July 1993).

[108] M. Saksena, R. Gerber and A. Agrawala, “Scheduling with Relative Timing Constraints", citeseer.nj.nec.com/26545.html

[109] M. Saksena, P. Freedman, and P. Rodziewicz. “Guidelines for automated implementation of executable object oriented models for real-time embedded control systems”. In *Proceedings of the IEEE Real—Time, Systems Symposium (RTSS'97)*, San Francisco, California, December 2--5 1997. IEEE Computer Society Press.

[110] M. Saksena, A. Ptack, P. Freedman, P. Rodziewics. “Schedulability Analysis for Automated Implementations of RealTime Object-Oriented Models”. *Proceedings of the Real-Time Systems Symposium*, Madrid, Spain, December 1998. Computer Society Press.

[111] Y. Wang and M. Saksena. “Scheduling fixed priority tasks with preemption threshold”. In Proceedings, *IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.

[112] M. Saksena, P. Karvelas, and Y. Wang. “Automatic synthesis of multi-tasking implementations from real-time objectoriented models”. In *Proceedings, IEEE International Symposium on Object-Oriented Real-Time, Distributed Computing*, March 2000.

[113] M. Saksena and P. Karvelas. “Designing for Schedulability: Integrating Schedulability Analysis with Object-Oriented Design”. In *Proceedings, Euromicro Conference on Real-Time Systems*, June 2000.

[114] William Lamie, *“Preemption Threashold™”*, Express Logic, Inc., http://www.expresslogic.com/wppreemption.html

[115] W. Henderson, D. Kendall, A. Robson, “Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems Computing Minimal Response Times and Reducing Jitter”, *International Journal of Time-Critical Computing Systems*, 20, 5-25, 2001 Kluwer Academic Publishers

[116] K. Gresser, “An event model for deadline verification of hard real-time systems”, In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118-123, Oulu,Finland, 1993.

[117] K. Gresser, *“Echtzeitnachweis ereignisgesteuerter Realzeitsysteme”*, Dissertation, Lehrstuhl für Prozeßrechner, Technische Universität München. Fortschrittsberichte VDI, Reihe 10, Nr. 268, VDI Verlag, Düsseldorf, 1993.

Ausführungszeit Messung

[118] S. Petters, *“Worst Case Execution Time Estimation for Advanced Processor Architectures”*, Dissertation, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2002.

[119] A. Shaw, "Reasoning about time in higher-level language software," *IEEE Transactions on Software Engineering*, vol. 15, July 1989.

[120] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gubstafsson and Hans Hansson. “Worst-case execution-time analysis for embedded real-time systems”. *Journal of Software Tools for Technology Transfer*, 2001.

[121] R. Kirner, R. Lang and P. Puschner, “WCET Analysis for Systems Modelled in Matlab/Simulink”, *22nd IEEE Real-Time Systems Symposium - Work in Progress Proceedings*, December 2001

[122] A. Colin and I. Puaut, “Worst-Case Timing Analysis Of The RTEMS Real-Time Operating System“, *Publication Interne Nr. 1277*, November 1999, IRISA

POSIX 4

[123] Bill Gallmeister, *„POSIX 4, Programming for the Real World”*, ISBN 1-56592-074-0, O’Reilly, 1st Edition September 1994

[124] IEEE Computer Society, *“1003.1b-1993 POSIX - Part 1: API C Language - Real-Time Extensions (ANSI/IEEE)”*, 1993. ISBN 1-55937-375-X.

RTEMS

[125] OAR Corporation, *“RTEMS C User's Guide”*, Edition 1, for RTEMS 4.5.0, On-Line Applications Research Corporation, 6 September 2000

[126] OAR Corporation, *“RTEMS POSIX API User's Guide”*, Edition 1, for RTEMS 4.5.0, On-Line Applications Research Corporation, 6 September 2000

[127] OAR Corporation, *“RTEMS POSIX 1003.1 Compliance Guide”*, Edition 1, for RTEMS 4.5.0, On-Line Applications Research Corporation, 6 September 2000

[128] OAR Corporation, *“BSP and Device Driver Development Guide”*, Edition 1, for RTEMS 4.5.0, On-Line Applications Research Corporation, 6 September 2000

[129] OAR Corporation, *“RTEMS Porting Guide”*, Edition 1, for RTEMS 4.5.0, On-Line Applications Research Corporation, 6 September 2000

A Anhang: Erweiterung der Block-Bibliothek mit m- und tlc-Files

A.1 Erweiterungen zur Modellierung der SW-Architektur

Nachfolgend sind die graphischen Blöcke, die zur Modellierung der neuen Konzepte in Simulink entwickelt wurden, dargestellt. Es handelt sich um den *Message Generator Block* und den *Message Receive Block*.

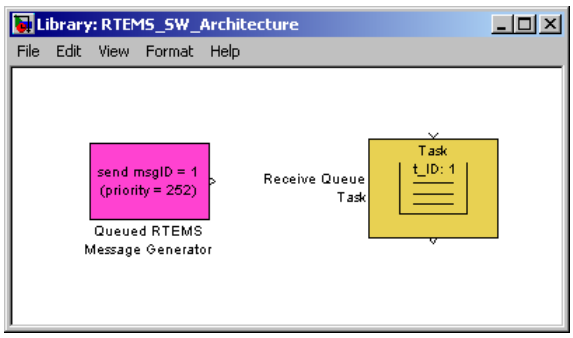

Bild 1: Blöcke zur Modellierung der neuen Konzepte in Simulink

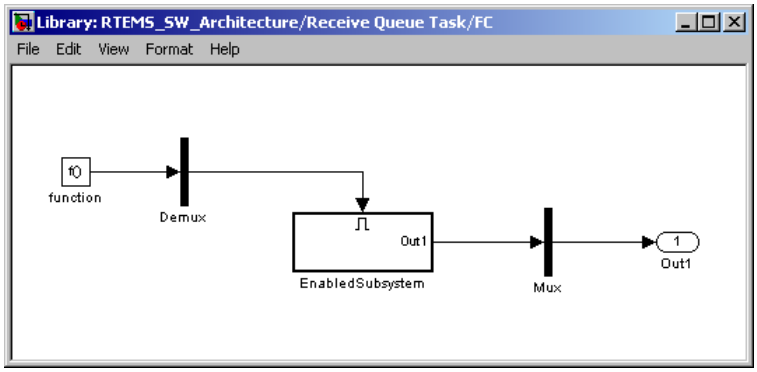

Bild 2: Innenleben des *Receive Queue Task Blocks*, abhängig von der Anzahl an empfangenen Message-Typen wird automatisch die Anzahl an *EnabledSubsystems* verändert.

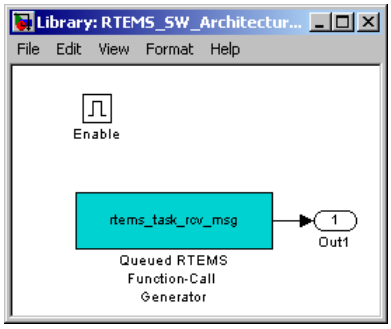

Bild 3: Innenleben des *EnabledSubsystems* im *Receive Queue Task Block*, bei der Codegenerierung bewerkstelligt dieser Block, dass aus dem angebundenen Functioncall-SubSystem ein Server-Thread wird.

A.2 rtems_task_rcv_msg_build.m

Dieses Skript passt die Anzahl an *EnabledSubsystems* des *Receive Queue Task Block*, abhängig von der Anzahl unterschiedlicher Message-Typen, an.

```
function rtems_task_rcv_msg_build(number, queueLength)
% Funktion zum Aufbau der Queue SubSysteme
% dienen zur Simulation von Systemen
% by Martin Orehek (RCS) 2002

% hole aktuellen Pfad des Systems
system = gcb;

% finde alle Demux
demux = find_system(system, 'LookUnderMasks','on','FollowLinks','on', ...
   'BlockType', 'Demux');

% finde alle Mux
mux = find_system(system, 'LookUnderMasks','on','FollowLinks','on', ...
   'BlockType', 'Mux');

% finde alle SubSystems (aktuelle, FunctionCall und alle EnabledSubSys.)
subsystems=find_system(system,'LookUnderMasks','on','FollowLinks','on', ...
                    'BlockType', 'SubSystem');
% aktuelles System
system = subsystems(1);

% Triggered Subsystem
trsubsys = subsystems(2);

% alle darunterliegenden Enabled Subsysteme (variabel!)
ensubsystems = subsystems(3:length(subsystems));

% haupt En.Subsystem (von dem werden Kopien gemacht!)
ensubsys = ensubsystems(1);
numensubsystems = length(ensubsystems);

% setze Linkoption 'inactive', sonst Fehlermeldung bei
% Veränderung!
if ( strcmp(get_param(char(system), 'LinkStatus'), 'resolved') )
  set_param(char(system), 'LinkStatus', 'inactive');
end
if ( strcmp(get_param(char(trsubsys), 'LinkStatus'), 'resolved') )
  set_param(char(trsubsys), 'LinkStatus', 'inactive');
end
if ( strcmp(get_param(char(trsubsys), 'LinkStatus'), 'resolved') )
  set_param(char(trsubsys), 'LinkStatus', 'inactive');
end

set_param(char(trsubsys), 'Tag',  num2str(queueLength));

% bestimme ob was dazu oder weg kommen muss!
if number > numensubsystems % add other enabled subsystems

  set_param(char(demux), 'Outputs', num2str(number));
  set_param(char(mux), 'Inputs', num2str(number));

  for i = (numensubsystems + 1) : number,
   position = get_param(char(ensubsys), 'Position');
   add_block(char(ensubsys), ...
            [char(ensubsys),int2str(i)], ...
            'Position', (position + [0 60 0 60] * (i-numensubsystems)));
   add_line(char(trsubsys), ['Demux/', int2str(i)], ...
           ['EnabledSubsystem',int2str(i),'/Enable']);
   add_line(char(trsubsys), ['EnabledSubsystem',int2str(i),'/1'], ...
           ['Mux/', int2str(i)]);
  end
```

```
elseif number < numensubsystems % remove some enabled subsystems
  % setze Linkoption 'inactive', sonst Fehlermeldung bei
  % Veränderung!
  if ( strcmp(get_param(char(trsubsys), 'LinkStatus'), 'resolved') )
    % set_param(char(system), 'LinkStatus', 'inactive');
    set_param(char(trsubsys), 'LinkStatus', 'inactive');
  end

  for i = (numensubsystems) : (-1) : (number + 1),
    delete_line(char(trsubsys), ['EnabledSubsystem',int2str(i),'/1'], ...
             ['Mux/', int2str(i)]);
    delete_line(char(trsubsys), ['Demux/', int2str(i)], ...
             ['EnabledSubsystem',int2str(i),'/Enable']);

    delete_block( [char(ensubsys),int2str(i)]);

  end

  set_param(char(demux), 'Outputs', num2str(number));
  set_param(char(mux), 'Inputs', num2str(number));
end
```

A.3 rtems_send_msg.tlc

Als Beispiel eines *Target-Language-Compiler* Files, das zur Codegenerierung der Blöcke aus Simulink verwendet wird, ist hier die Implementierung des *Message Generator Block* wiedergegeben.

```
%% $RCSfile: rtems_send_msg.tlc,v $
%% Abstract:
%%   Message Generator Block target file.
%implements rtems_send_msg "C"
%include "rtems_swarch_lib.tlc"

%function BlockInstanceSetup(block, system) void
%endfunction

%function Start(block, system) Output
  %foreach fcnCallIdx = NumSFcnSysOutputCalls
    %% call the downstream system
    %with SFcnSystemOutputCall[fcnCallIdx]
      %% skip unconnected function call outputs
      %if LibIsEqual(BlockToCall, "unconnected")
        %continue
      %endif

      %assign sysIdx = BlockToCall[0]
      %assign blkIdx = BlockToCall[1]
      %assign ssBlock = System[sysIdx].Block[blkIdx]
      %assign block = block + ssBlock
      %assign sysToCall = System[ssBlock.ParamSettings.SystemIdx]
      %%assign taskName = ssBlock.Identifier
      %% Support message queue
      %%
      %% The queue name is
      %%   o taskBlockIdentifier_semaphore (task name not specified)
      %%
      %assign taskQueueName = "%<sysToCall.Identifier>_queue"

      %% open the existing message queue (if not implicitly create one)
      /* Block: %<Name> */
      {
       %if (debug_rtems == 1)
       #define DBGputs(a)   puts(a)
       %else
       #define DBGputs(a)
       %endif
```

```
      /* open existing message queue for fc-task: %<ssBlock.Name> */
      /* if not existing create it implicitly */
      struct mq_attr      MQattr;
      %% erhalte die eingestellte MessageLength über das
      %% Tag des FC-Blockes im TaskQueue Block!
      %assign maxmsg = CAST("Number", ssBlock.Tag)
      MQattr.mq_maxmsg  = (int) %<maxmsg>;
      MQattr.mq_msgsize = sizeof(struct mq_message);
      %<LibBlockPWork(QueueID,"","",0)> = (void *) mq_open( \
                                          "%<taskQueueName>", \
                                         O_CREAT | O_RDWR, \
                                         0x777, &MQattr);
      if ( %<LibBlockPWork(QueueID,"","",0)> == (void *) (-1) ) {
        ssSetErrorStatus(%<tSimStruct>,\
       "mq_open call of existing block %<ssBlock.Name>.\n");
        assert(0);
      }
      DBGputs("open mq: %<taskQueueName> in block:  %<Name>");
      #undef DBGputs
     }
    %endwith
  %endforeach
%endfunction

%function Outputs(block, system) Output
  %foreach fcnCallIdx = NumSFcnSysOutputCalls
    %% call the downstream system
    %with SFcnSystemOutputCall[fcnCallIdx]
      %% skip unconnected function call outputs
      %if LibIsEqual(BlockToCall, "unconnected")
        %continue
      %endif
      %assign sysIdx = BlockToCall[0]
      %assign blkIdx = BlockToCall[1]
      %assign ssBlock = System[sysIdx].Block[blkIdx]
      %assign sysToCall = System[ssBlock.ParamSettings.SystemIdx]
      %assign msgID = CAST("Number", SFcnParamSettings.MsgID)
      %assign msgPriotity = CAST("Number", SFcnParamSettings.MsgPriority)
      /* Block: %<Name> */
      {
      %if (debug_rtems == 1)
       #define DBGputs(a)   puts(a)
       %else
       #define DBGputs(a)
       %endif

      /* send message to receiving fc-system */
       int status;
       struct mq_attr      MQattr;
       struct mq_message   message;
       message.msgID = %<msgID>;
       message.FcnPortElement = %<SFcnSystemOutputCall[0].FcnPortElement>;
                               /* send FcnPortElement value */

       status = mq_getattr((mqd_t) \
                           %<LibBlockPWork(QueueID,"","",0)>, &MQattr);
       assert(!status);
       if (MQattr.mq_curmsgs != MQattr.mq_maxmsg) {
          status = mq_send((mqd_t) %<LibBlockPWork(QueueID,"","",0)>, \
                           (char *) &message, sizeof(struct mq_message),\
                           (int)   %<msgPriotity>);
          assert(!status);
          DBGputs("snd_msg:  %<Name>");
       }
       else {
         puts("Error sending MSG, Queue is full");
         assert(0);
       }
      #undef DBGputs
    }
```

```
    %endwith
  %endforeach
%endfunction

%function Terminate(block, system) Output
  /* Block: %<Name> */
  /* do nothing */
%endfunction

%% [EOF] rtems_send_msg.tlc
```

A.4 Block-Bibliothek zur Realisierung der Co-Simulation

Die hier dargestellten Blöcke wurden im Rahmen einer Diplomarbeit [77] entwickelt.

Der *CoSimulationHostInterface* Block wird im PC-Modell zur Kommunikation mit dem Target eingebaut. Die Eingänge (links) sind entweder Daten-Signale (Input_X) oder Ereignis-Signale (EV_X). Die Daten-Signale besitzen einen im Modell eindeutigen Identifier (Host: X, Target: SigID), über dem die Zuordnung des augenblicklich übermittelten Wertes vom PC-Modell zum Target-Modell durchgeführt wird. Im Target-Modell sorgt der *RT-SimOutput* Block für die Weiterverarbeitung des übermittelten Signals im implementierten Design. Für die Übermittlung der auf dem Target berechneten Ergebnisse wird der *RT-SimInput* Block im Target-Modell eingesetzt. Die übermittelten Signale werden dem PC-Modell über die Ausgänge des *CoSimulationHostInterface* Blocks (rechts) ausgegeben (Zuordnung wieder über IDs). Die Ereignis-Eingänge dienen zur gezielten Aktivierung von Reaktionen in der Software-Architektur auf dem Target während der Co-Simulation. So wird z.B. bei einer reinen periodischen Ausführung zu jedem Abtastzeitpunkt der PIT-IRQ über ein geeignetes Ereignis übermittelt. Werden hingegen auch externe, aperiodische Interrupts simuliert (z.B. Hochregallager), werden entsprechend zusätzliche EV-Signale in die Modelle eingeführt. Auf Target-Seite wird der Empfang eines solchen Ereignisses durch die Blöcke: *Co-Simulation Ext Interrupt Control* und *Co-Simulation IMB Interrupt Control* emuliert, die im RT-Modell die richtige Anbindung an die Target-Interrupt-Hardware realisieren.

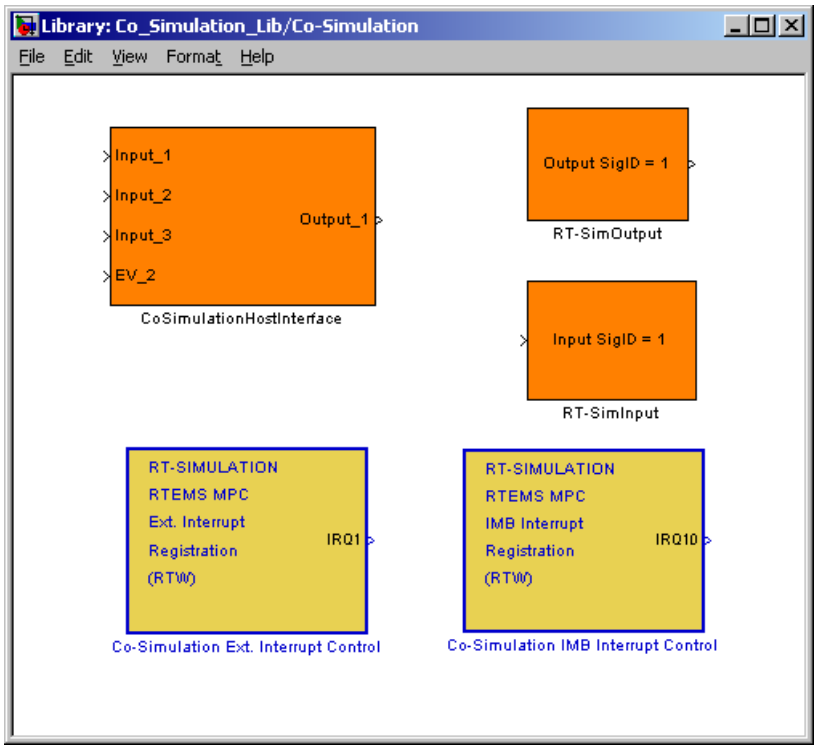

Bild 4: Blockbibliothek zur Realisierung der Co-Simulation

Bei einer Co-Simulation wird auf dem Target-System neben den eigentlichen Tasks zur Umsetzung der graphischen Spezifikation, zusätzlich eine Manager-Task erzeugt. Diese Manager-Task stellt die Kommunikation über einen seriellen Link zum PC-Modell her. Sie besitzt vor jedem Ausführungsschritt immer die höchste Priorität im System. Nachdem alle nowendigen Daten-Signale empfangen und an die verschiedenen Blöcke verteilt wurden, werden die für den aktuellen Co-Simulationsschritt notwendigen Ereignisse empfangen. Je nach Ereignis ruft die Manager-Task dann alle entsprechenden Interrupt-Händler auf und verteilt dadurch alle aktuell im System geforderten Berechnungswünsche (Semaphoren für zyklische Bearbeitung, Messages für Ereignis-getriebene Bearbeitung). Dabei wird die Manager-Task nicht unterbrochen, da sie die höchste Priorität im System besitzt. Am Ende setzt sie ihre Priorität unter der tiefsten Modell-Task-Priorität und startet damit die Ausführung des aktuellen Co-Simulationsschrittes. Sind alle Aktionen beendet spricht die Manager-Task wieder an, setzt die eigene Priorität höher und übermittelt die berechneten Ergebnisse.

Während eines solchen Simulationsablaufes wird nicht nur die Funktion des Modells verifiziert, sondern auch bei jedem Schritt die in Anspruch genommenen Berechnungsdauern aller einzelnen Aktionen gemessen. Diese realistischen Messungen werden in einer Text-Datei gespeichert, um off-line die Realzeitanalyse mit realistischen Ausführungszeiten zu ermöglichen.

Weitere Informationen zur Umsetzung der Co-Simulation sind in der Diplomarbeit: „Entwicklung einer Softwarearchitektur zur Messung von Ausführungszeiten in einer Simulationsumgebung“ [77] zu finden.

B Anhang: Automatisiertes Parsen des graphischen Modells

B.1 parse_for_APG.m

```
function [action_chain_gl,action_gl,Ts,file_name]=parse_for_APG(file_name)
%[action_chain_gl, action_gl, Ts, file_name] = ...
%                                       parse_for_APG('file_name')
%
% in this function the model '<file_name>.mdl' is parsed and the
% different actions gained by the parsing process are filled with
% right data from:
% -> the Co-Simulation measurement
%    '.\RT_SimMeasure\<file_name>_exeTime.m'
% -> and the parse information within:
%    '.\RT_SimMeasure\<file_name>_ParseInfo.m'
%
% input:
%   file_name ......... file name to open and parse the right model
%
% outputs:
%   action_chain_gl ... vector of different actions of a transaction
%   action_gl ......... structure array of all actions within the
%                       sw-arch.
%   Ts ................ vector of the sample-rates of cyclic model part
%   file_name ......... file name for postprocessing
%
% (c) 2003 Martin Orehek (RCS) (Martin.Orehek@rcs.ei.tum.de)
%
% clear global, to avoid problems with existing variables with the
% same name!
clear global

% globals to exchange data between local functions
global action_gl;
global recursion_level;
global action_chain_gl;
global file_name_gl;
global fid_gl
global answers_gl
global line_index_gl;
global max_ev_exe_times;
global C_ISR_overhead;
global C_tBase_overhead;
global C_tRateX_overhead;
global C_first_EV_action_overhead;
global C_other_EV_action_overhead;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Runtime Overhead due to RTOS and measurements %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% they results from a reference measurement of the runtime
% implementation
TB_resolution = 0.0004; %msec TimeBase resolution of the Hardware
% see file for details: RTOS_Overhead_RTEMS.xls
% measured once within the SW-frame
C_schedule_tBase            = 0.0096 + TB_resolution; % msec
C_schedule_tRateX           = 0.0380 + TB_resolution; % msec
% IRQ Entry RTOS + IRQ-MPC555 ExtException Handling Entry
% ++ IRQ-MPC555 ExtExcep. Exit + IRQ Exit RTOS
```

```
C_ISR_overhead              = 0.0275 + TB_resolution; % msec
% TaskSwitch + sem_wait ++ sem_wait + TaskSwitch
C_tBase_overhead            = 0.1192 + TB_resolution; % msec
% sem_wait ++ sem_wait + TaskSwitch
C_tRateX_overhead           = 0.0744 + TB_resolution; % msec
% TaskSwitch + mq_receive ++ mq_receive + TaskSwitch
C_first_EV_action_overhead  = 0.1312 + TB_resolution; % msec
% mq_receive ++ mq_receive + TaskSwitch
C_other_EV_action_overhead  = 0.0864 + TB_resolution; % msec

file_name_gl = file_name;

% open Model
open_system(file_name);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1. PARSE model gattering informations for processing %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prepare model information needed for graph %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Find all MsgGen blocks in the model
all_send_msg_p = find_system(file_name, 'FunctionName', ...
                         'rtems_send_msg');
all_send_msg_h = get_param(all_send_msg_p, 'Handle');

% check for corect msgIDs: (1..X) no doubles!
ids_cell = get_param(all_send_msg_p, 'msgID');
ids = [];
for i = 1:length(ids_cell)
    ids = [ids, evalin('base', ids_cell{i})];
end

ids = sort(ids);
for i = 1:length(ids)-1
    if ( (ids(i+1) - ids(i)) ~= 1 )
        error(['Message ID: ', num2str(ids(i)),[' in the model is' ...
                 ' NOT O.K.! (double or one ID is missed between)']);
    end
end

%Find all EXTERNAL msgGen Blocks in the model
ext_send_msg_p = find_system(file_name, 'FunctionName', ...
                         'rtems_send_msg', 'extEV', 'on' );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prepare IRQ information needed for graph %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Find all External Interrupt Blocks in the model (must be only one!)
%ToDo: Handling also IMB-IRQ-Block here!
all_ext_IRQ_p = find_system(file_name,'RegExp','on', 'MaskType', ...
                        '\<Interrupt Block\>'); %must be only ONE!
if length(all_ext_IRQ_p) > 1
    error('To many "RTEMS Interrupt Blocks" in the model!');
end

if length(all_ext_IRQ_p) ~= 0 %there is a IRQ-Block in the model
    ext_IRQ_numbers = str2num(cell2mat(...
       get_param(all_ext_IRQ_p, 'irqNumbers'))); %get vector numbers
    % look for ISR SubSystems
    ext_IRQ_h = get_param(all_ext_IRQ_p{1}, 'Handle');
    ext_IRQ_port_h = get_param(ext_IRQ_h, 'PortHandles');
    for i = 1:length(ext_IRQ_port_h.Outport)
        irq_connection_line_h = ...
          get_param(ext_IRQ_port_h.Outport(i), 'Line');
        ISR_system_h(i) = ...
          get_param(irq_connection_line_h, 'DstBlockHandle');
          %contains handle to ISR SubSystems
    end
end %if
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2. import measured data and sample-rate informations %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get all execution times necessary for the graph and the
% action spezification
[max_irq_exe_times, irq_time_data, max_cyc_exe_times, ...
        cyclic_time_data, cyclic_schedule_time_data, ...
        max_ev_exe_times, ev_time_data, Ts] = ...
    get_CompTimes_all(file_name);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% check if there is a path to build or not! %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ( isempty(ext_send_msg_p) & isempty(Ts) & isempty(all_ext_IRQ_p))
    error(['ERROR, no external MsgGenerators, nor sample rates,',...
          ' nor IRQs in the Model: ', file_name]);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%
%% initialize actions %%
%%%%%%%%%%%%%%%%%%%%%%%%%%
%Number = (all actions due to MessageBlocks) + ...
%         (actions for ISR/FunctionCallBlock Overhead)
action_gl_prototype = struct('id', [], 'priority', [], ...
                         'sucessor', [], 'comp_time', [], ...
                         'thread', [], 'msg_name', []);
action_gl = action_gl_prototype;

% initialize "event actions", regular actions, internal actions,
% Number = Number_of_SendEventBlocks
for i = 1:length(all_send_msg_h)
    action_gl(i) = action_gl_prototype;
end

% initialize "external actions" (ISRs or trigger form cyclic part!)
% Number = Number_of_External_SendEventBlocks
% IMPORTANT: here the mapping of HW-Interrupt-Levels onto
%            action-priorities must be done!
irq_priority_offset = 500; %to analyze ISRs together with other actions
irq_thread_offset = irq_priority_offset; %to analyze ISRs together
                                         %with other actions

% for all additional actions a datastructure must be initialized
% additional actions are: ISRs and ext_send_msg Blocks
% it can happen:
% 0. that one ext_send_msg Block is in a ISR system,
%    => one action is required!
% 1. that an ISR is without an ext_send_msg Block,
%    => an action is required!
% 2. that a ext_send_msg Block is in a conditionally triggered
%    sampled modelpart, => an action is required
% 3. two or more ext_send_msg Blocks are in one ISR system,
%    => only one action is required
% 4. two or more ext_send_msg Blocks are in a conditionally
%    triggered sampled modelpart, => only one action is required
% at the moment only case 0. is considered !!!

for i = (length(all_send_msg_h) + 1) : (length(all_send_msg_h)+ ...
                                 length(ext_send_msg_p))
    %ATTENTION: for each External MsgSendBlock an additional action is
    %           included, to allow analysis of ISR overhead!
    action_gl(i) = action_gl_prototype;
    action_gl(i).id = i; % ID is starting from last MsgID!
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% fill event-actions with right data %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trace answers for later recall!
cr = sprintf('\n');
```

```
if exist([file_name_gl,'_log.m'])
    resp = input(['??? do you want to overwrite existing ',...
              'trace file (yes, no) or use it? [u, y, n]: '],'s');
    switch(resp)
    case {'n'}
        %do nothing
        fid_gl = [];
    case {'y'}
        %overwrite existing file
        fid_gl = fopen([file_name_gl,'_log.m'],'w');
        fprintf(fid_gl, ['function answers = ',file_name_gl,'_log()', cr, cr]);
        fprintf(fid_gl, ['answers = struct(''response'', '''','',...
                         '''message'', '''' );', cr]);
        line_index_gl = 1;
    case {'', 'u'}
        %use available file!
        disp([cr,'### use decisions traced in file: ',file_name_gl,...
             '_log.m',cr]);
        answers_gl = eval([file_name_gl,'_log']);
    otherwise
        error('ERROR, unknown choice!');
    end
else
    fid_gl = fopen([file_name_gl,'_log.m'],'w');
    fprintf(fid_gl, ['function answers = ',file_name_gl,'_log()', cr, cr]);
    fprintf(fid_gl, ['answers = struct(''response'', '''','',...
                     '''message'', '''' );', cr]);
    line_index_gl = 1;
end

%% fill  actions with right data, row data for the transaction graph
for i =1:length(ext_send_msg_p)
  src_h = get_param(ext_send_msg_p{i}, 'Handle');
  %find ISR-Handle of external EventSource
  % ToDo: handling of extenal event sources, starting
  %       from a cyclic part!
  src_parent_p = get_param(src_h, 'Parent'); %upper parent of
                                             %ExtMsgSend Block
  src_parent_parent_p = src_parent_p;  %one level higher
                                       %=> to recognize 'block_diagram'
  while( ~strcmp(get_param(src_parent_parent_p, 'Type'), ...
              'block_diagram') )
      % move up the hirarchie, getting to the ISR-SubSystem block!
      src_parent_p = src_parent_parent_p;
      src_parent_parent_p = get_param(src_parent_p, 'Parent');
  end

  clear this_IRQ;
  for j = 1:length(ISR_system_h)
      if(get_param(src_parent_p, 'Handle') == ISR_system_h(j))
          this_IRQ = ext_IRQ_numbers(j);
          continue;
      end
  end

  %create Root of Transaction Graph
  msgID = evalin('base',get_param(src_h, 'msgID'));

  action_gl(length(all_send_msg_h) + i).sucessor = msgID;

  if ( ~exist('this_IRQ') )
      warning(['ATTENTION: Block ',get_param(src_parent_p, 'Name'), ...
            ' is not an ISR SubSystem!']);
      action_gl(length(all_send_msg_h) + i).priority = 200;
       %ToDo: here should come the priority of the cyclic
               %      triggering part!!
      action_gl(length(all_send_msg_h) + i).msg_name = ...
         ['ExtEvent_',get_param(src_parent_p, 'Name')];
      action_gl(length(all_send_msg_h) + i).comp_time = 0;
      %ToDo: fill in here the right C (hard to define!!)
```

```
  else
      action_gl(length(all_send_msg_h) + i).priority = ...
        irq_priority_offset - map_Ext_IRQ(this_IRQ);
      action_gl(length(all_send_msg_h) + i).msg_name = ...
        ['IRQ_',num2str(this_IRQ)];
      ISR_comp_time = max_irq_exe_times(map_Ext_IRQ(this_IRQ)) + ...
        C_ISR_overhead;
      action_gl(length(all_send_msg_h) + i).comp_time = ISR_comp_time;
  end
  add_new_activity(src_h); %recursive function!!
end %for

if (~isempty(fid_gl)) & (fid_gl ~= -1)
    fclose(fid_gl);
    disp(['closed file: ',file_name_gl,'_log.m !']);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% fill ISR-actions with thread IDs %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get highes Thread-Number
irq_thread_offset = 0;
for i = 1:length(action_gl)
    if (irq_thread_offset < action_gl(i).thread)
        irq_thread_offset = action_gl(i).thread;
    end
end
for i =1:length(ext_send_msg_p)
    action_gl(length(all_send_msg_h) + i).thread = ...
      irq_thread_offset + i; %all ISRs are preemptible otherwise
                             %use same pseudo Thread
end

% initialize action_chain_gl, used for further real-time analysis
for i = 1:length(ext_send_msg_p)
    action_chain_gl{i} = {[]};
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  EV driven part
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
recursion_level = 0;
ext_ev_action_vector = [];
for i = 1:length(ext_send_msg_p)
    build_action_chain(action_gl(length(all_send_msg_h) + i), i);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  TT Cyclic part
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ( length(Ts) ~= 0 ) % is cyclic part in model?
    PIT_priority = 6; % HW IRQ priority level of the
                      % PeriodicInterruptTimer (PIT)
    tBase_priority = 200;
    % get highes Thread-Number
    PIT_thread = 0;
    for i = 1:length(action_gl)
        if (PIT_thread < action_gl(i).thread)
            PIT_thread = action_gl(i).thread;
        end
    end
    PIT_thread = PIT_thread + 1;
    tBase_thread = PIT_thread + 1; % so it is a continuous number
                                   % and max(action_gl.thread)
                                   % gives the total number of threads
    %extend and initialize action_chain_gl for cyclic part
    action_chain_gl_start_index_TT = length(action_chain_gl) + 1;
    action_chain_gl_stop_index_TT  = length(action_chain_gl) + ...
                                     length(Ts);
    for i = action_chain_gl_start_index_TT : action_chain_gl_stop_index_TT
```

```
        action_chain_gl{i} = {[]};
    end
    action_gl_PIT_index = length(all_send_msg_h) + ...
                          length(ext_send_msg_p) + 1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% build the additional actions for the cyclic part  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % include PIT action to consider PIT_ISR
    action_gl(action_gl_PIT_index) = action_gl(end);
    action_gl(action_gl_PIT_index).id = action_gl_PIT_index;
    action_gl(action_gl_PIT_index).priority = ...
      irq_priority_offset - map_IMB_IRQ(PIT_priority);
    action_gl(action_gl_PIT_index).sucessor = action_gl_PIT_index + 1;
    C_PIT_ISR = max_irq_exe_times(map_IMB_IRQ(PIT_priority)) + ...
              C_ISR_overhead; % msec, measured exe_time of PIT_ISR
    action_gl(action_gl_PIT_index).comp_time = C_PIT_ISR;
    action_gl(action_gl_PIT_index).thread = PIT_thread;
    action_gl(action_gl_PIT_index).msg_name = ...
      ['PIT_IRQ_LvL_',num2str(PIT_priority)];
    action_gl_start_index_TT = length(action_gl) +1;
    action_gl_stop_index_TT  = length(action_gl) + 2*length(Ts);
    j = 0;
    % build automatically all actions of the cyclic part!!
    if length(Ts) == 1 %single tasking mode
        % A_tBase
        C_A_tBase           = max_cyc_exe_times(j+1);
        action_gl(2) = struct('id', [2], ...
                    'priority', [tBase_priority], ...
                    'sucessor', [], ...
            'comp_time', [C_A_tBase],...
                 'thread', [tBase_thread], ...
            'msg_name', ['Ts_',num2str(j),...
                              ' = ',num2str(Ts(j+1)),' sec']);
        j = j + 1;
    else
      for i = action_gl_start_index_TT : 2 : action_gl_stop_index_TT
        if( i == action_gl_start_index_TT )
            % A_schedule_tBase, A_tBase
            C_A_schedule_tBase  = C_schedule_tBase + C_tBase_overhead;
            C_A_tBase           = max_cyc_exe_times(j+1);
            action_gl(i) = struct('id', [i], ...
                            'priority', [tBase_priority], ...
                            'sucessor', [i+1], ...
                 'comp_time', [C_A_schedule_tBase],...
                            'thread', [tBase_thread], ...
                 'msg_name', ['Ts_',num2str(j),...
                                    ' = ',num2str(Ts(j+1)),' sec']);
            action_gl(i+1) = struct('id', [i+1], ...
                              'priority', [tBase_priority], ...
                              'sucessor', [], ...
                              'comp_time', [C_A_tBase], ...
                              'thread', [tBase_thread], ...
                              'msg_name', ['Schedule_tRate',...
                                           num2str(j)]);
            j = j + 1;
        else
            % A_schedule_tRateX, A_tRateN
            C_A_schedule_tRateX = C_schedule_tRateX;
            C_A_tRateX          = max_cyc_exe_times(j+1) + ...
                                  C_tRateX_overhead;
            action_gl(i) = struct('id', [i], ...
                            'priority', [tBase_priority], ...
                            'sucessor', [i+1], ...
                            'comp_time',[C_A_schedule_tRateX],...
                            'thread', [tBase_thread], ...
                            'msg_name', ['Ts_',num2str(j),...
                                    ' = ',num2str(Ts(j+1)),' sec']);
            action_gl(i+1) = struct('id', [i+1], ...
                              'priority', [tBase_priority-j],...
```

```
                                'sucessor', [], ...
                                'comp_time', [C_A_tRateX], ...
                                'thread', [tBase_thread+j], ...
                                'msg_name', ['Schedule_tRate',...
                                              num2str(j)]);
            j = j + 1;
          end % if
        end
      end %if length(Ts) == 1 %single tasking mode

      j = action_chain_gl_start_index_TT; %length(action_gl) + 1;

      % add complet path for this starting ext_msg
      build_action_chain(action_gl(action_gl_PIT_index), j);
      j = j + 1;
      % add other TT cyclic Transactions
      for i = (action_gl_start_index_TT + 2) : 2 : action_gl_stop_index_TT
          % add complet path for this starting ext_msg
          build_action_chain(action_gl(i), j);
          j = j + 1;
      end
  end %if ( length(Ts) ~= 0 ) % is cyclic part in model?
  return;

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  function action_id = add_new_activity(src_h)
  % function adds new activity to the transaction graph which is GLOBAL!
  % will be called RECURSIVELY
  % input:  'src_h'  ... source handle (MsgGen)
  global action_gl;
  global file_name_gl;
  global fid_gl;
  global answers_gl;
  global line_index_gl;

  [activity, internal_src_h, srv_task_h] = ...
      get_activity_possible_internal_EVs(src_h);

  action_id = activity.id; % id of elaborated action!

  if(isempty(action_gl(action_id).id))
      % activity is NOT initialized
      % => questions about application are necessary!
      action_gl(action_id) = activity;
      for k = 1:length(internal_src_h)
        cr = sprintf('\n');
        if isempty(answers_gl)
          % ask designer for sent messages
          ButtonName = input(['Can message: ',...
                          get_param(src_h, 'Name'), ...
                          ' trigger message: ', ...
                          get_param(internal_src_h(k), 'Name'),...
                          ' ?', '[y, n, c]: '],'s');
          switch ButtonName,
              case {'', 'y', 'yes'},
                  message = ['YES, message: ',get_param(src_h, 'Name'), ...
                      ' can trigger message: ', ...
                      get_param(internal_src_h(k), 'Name'),' !'];
                  disp([message,cr]);
                  if (~isempty(fid_gl)) & (fid_gl ~= -1)
                    %write response into the trace file
                    fprintf(fid_gl, ['answers(',num2str(line_index_gl),...
                          ') = struct(''response'', ''YES'',',...
                          '''message'', ''',message,''');' , cr]);
                    line_index_gl = line_index_gl +1;
                  end
                  %recursive call! AND extend sucessor vector if
                  %there is one, return is always 'local_index'
                  action_gl(action_id).sucessor = ...
                              [action_gl(action_id).sucessor, ...
```

```
                            add_new_activity(internal_src_h(k))];
            case {'n','no'},
                message = ['NO, message: ',...
                         get_param(src_h, 'Name'), ...
                         ' can not trigger message: ', ...
                         get_param(internal_src_h(k), 'Name'),' !'];
                disp([message, cr]);
                if (~isempty(fid_gl)) & (fid_gl ~= -1)
                  %write response into the trace file
                  fprintf(fid_gl, ['answers(',num2str(line_index_gl),...
                        ') = struct(''response'', ''NO'',',...
                        '''message'', ''',message,''');' , cr]);
                  line_index_gl = line_index_gl +1;
                end
            case {'c', 'cancel'},
                disp(['CANCEL, parsing interrupted by user ',...
                         'processing question:', cr,cr, ...
                         'Can message: ',get_param(src_h, 'Name'), ...
                         'trigger message: ', ...
                         get_param(internal_src_h(k), 'Name'),' ?', cr]);
                error('parsing interrupted by user');
            otherwise
                error(['ERROR: unknown choice!']);
        end %switch
     else %if answers_gl => use already traced answers!
        if strcmp(answers_gl(1).response, 'YES')
            answers_gl = answers_gl(2:end); %move on in structure!
            message = ['YES, message: ',get_param(src_h, 'Name'), ...
                       ' can trigger message: ', ...
                       get_param(internal_src_h(k), 'Name'),' !'];
            disp([message,cr]);
            %recursive call! AND extend sucessor vector if
            %there is one, return is always 'local_index'
            action_gl(action_id).sucessor = ...
                    [action_gl(action_id).sucessor, ...
                    add_new_activity(internal_src_h(k))];
        else
            answers_gl = answers_gl(2:end); %move on in structure!
            message = ['NO, message: ',...
                         get_param(src_h, 'Name'), ...
                          ' can not trigger message: ', ...
                          get_param(internal_src_h(k), 'Name'),' !'];
            disp([message, cr]);
        end
     end %if
   end %for
end %if
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [activity, internal_src_h, srv_task_h] = ...
      get_activity_possible_internal_EVs(src_h)
% function follows line to the handle of the RcvQueueTask Block,
% and checks for POSSIBLE new internal events (EVs) within ServerTask
% input:  'src_h'  ... source handle (MsgGen)
% return: 'activity' ... structure of activity (only partially filled!
%                        missing User interaction!)
%         'internal_src_h' ... handles of POTENTIALLY new internal EVs

%Find all the line handles for this model
global file_name_gl
global max_ev_exe_times
global C_first_EV_action_overhead;
global C_other_EV_action_overhead;

all_lines_h = find_system(file_name_gl,'FindAll','On','type', 'line');

%Find all EXTERNAL msgGen Blocks in the model
ext_send_msg_p = find_system(file_name_gl, 'FunctionName', ...
                        'rtems_send_msg', 'extEV', 'on' );
```

```
ext_send_msg_h = get_param(ext_send_msg_p, 'Handle');

  % handle of RTEMS TaskQueue Block
  msg_queue_h = get_dst_RcvQueueTask(src_h);

  % get fct-call Subsystem implemented as Server-Task!
  msg_queue_PortHandles = get_param(msg_queue_h, 'PortHandles');
  msg_queue_port_h = msg_queue_PortHandles.Outport( 1 ); %TaskQueue
                                                  %Block has only ONE output
  connection_line = find_system(all_lines_h, 'SrcBlockHandle', ...
                          msg_queue_h); %TaskQueue Block
                                           %has only ONE output

  srv_task_h = get_param(connection_line, 'DstBlockHandle');

  % src_h == source of message
  % srv_task_h == server task for activity and source of src_h
  msgID = evalin('base', get_param(src_h, 'msgID'));
  msgName = get_param(src_h, 'Name');
  priority = evalin('base', get_param(src_h, 'msgPriority'));
  tID = str2num( get_param(msg_queue_h, 'taskID') );

  if ( any(find(cell2mat(ext_send_msg_h) == src_h)) )
      % first action of a transaction graph
      this_exe_time = max_ev_exe_times(msgID) + ...
                    C_first_EV_action_overhead;
  else
      % other action of a transaction graph
      this_exe_time = max_ev_exe_times(msgID) + ...
                    C_other_EV_action_overhead;
  end

  activity = struct('id', msgID, 'priority', priority, ...
                 'sucessor', [], 'comp_time', this_exe_time, ...
                 'thread', tID, 'msg_name', msgName);

  internal_src_h = find_system(srv_task_h, 'FunctionName', ...
                           'rtems_send_msg');
  return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function dst_h = get_dst_RcvQueueTask(src_h)
% function follows line to the handle of the RcvQueueTask Block!
% input:  'src_h'  ... source handle
% return: 'dst_h' ... handle of the RcvQueueTask Block
  global file_name_gl

  %Find all the line handles for this model
  all_lines_h = find_system(file_name_gl,'FindAll','On',...
                       'type', 'line');
  %Find all 'Goto' and 'From' blocks
  all_from_p = find_system(file_name_gl, 'BlockType', 'From');

  help_h = src_h;

  while( ~strcmp( get_param(help_h, 'MaskType'), ...
                'RTEMS queued Task Block') )
     if( strcmp( get_param(help_h, 'BlockType'), 'S-Function') )
         % message generator
         connection_line = find_system(all_lines_h, ...
                                  'SrcBlockHandle', help_h);
                          % has only ONE output
         help_h = get_param(connection_line, 'DstBlockHandle');
         help_port_h = get_param(connection_line, 'DstPortHandle');
     elseif( strcmp( get_param(help_h, 'BlockType'), 'Outport') )
          % get next handle for block
          port_number = str2num( get_param(help_h, 'Port') );
                      % portnumber of signal
          parent_p = get_param(help_h, 'Parent');
          help_PortHandles = get_param(parent_p, 'PortHandles');
```

```
          help_h = get_param(parent_p, 'Handle');
                 % new handle of SubSystem
          help_port_h = help_PortHandles.Outport( port_number );
     elseif( strcmp( get_param(help_h, 'BlockType'), 'SubSystem') )
          connection_line = get_param(help_port_h, 'Line');
          help_h = get_param(connection_line, 'DstBlockHandle');
          help_port_h = get_param(connection_line, 'DstPortHandle');
     elseif( strcmp( get_param(help_h, 'BlockType'), 'Mux') )
          connection_line = find_system(all_lines_h, ...
                                'SrcBlockHandle', help_h);
                           %Mux has only ONE output
          help_h = get_param(connection_line, 'DstBlockHandle');
          help_port_h = get_param(connection_line, 'DstPortHandle');
     elseif( strcmp( get_param(help_h, 'BlockType'), 'Goto') )
          for j = 1:length(all_from_p)
              if( strcmp( get_param(help_h, 'GotoTag'), ...
                      get_param(all_from_p{j}, 'GotoTag')) )
                  help_h = get_param(all_from_p{j}, 'Handle');
                        %got right exit of signals!
                  break;
              end
          end
     elseif( strcmp( get_param(help_h, 'BlockType'), 'From') )
          connection_line = find_system(all_lines_h, ...
                                'SrcBlockHandle', help_h);
                           %'From' has only ONE output
          help_h = get_param(connection_line, 'DstBlockHandle');
          help_port_h = get_param(connection_line, 'DstPortHandle');
     elseif( strcmp( get_param(help_h, 'BlockType'), 'Terminator') )
          error('ERROR: BlockType: Terminator are NOT allowed !\n');
     else % unknown BlockType
          hilite_system(help_h);
          error('ERROR: unknown BlockType!\n');
     end
  end %While

    dst_h = help_h;
  return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  newNode = build_action_chain(action, transaction_id)
% function returns the format string for a new node!
% input:  'action'  ... action with information for Node
% return: 'newNode' ... string for Node within DaVinci Tool
%  activity = struct('id',    msgID,  'priority',  priority,   ...
%        'sucessor', [],  'comp_time',    NaN,     'thread',   tID);
global action_gl;
global recursion_level
global transaction_graph
global action_chain_gl

recursion_level = recursion_level + 1;

if ( iscell(action_chain_gl{transaction_id}) )
    chain = cell2mat( action_chain_gl{transaction_id});
else
    chain = action_chain_gl{transaction_id};
end

if ( isempty(chain) | isempty( find(chain == action.id ) ) )
    % following action is NOT part of the chain already!
    % => can be added
    chain = [chain, action.id];
    action_chain_gl{transaction_id} = chain;
    for i = 1:length(action.sucessor)
        build_action_chain(action_gl(action.sucessor(i)), ...
                    transaction_id);
    end
else
    warndlg(['PARSE WARNING: loop in transaction ',...
```

```
            num2str(transaction_id),' with action ',num2str(action.id)], ...
           'activity loop');
end

recursion_level = recursion_level - 1;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function internal_IRQ_level = map_Ext_IRQ(Ext_IRQ_Level)
%mapping of External IRQ Level [1..7] onto internal IRQ Level [0..39]

internal_IRQ_level = Ext_IRQ_Level * 2;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function internal_IRQ_level = map_IMB_IRQ(IMB_IRQ_Level);
%mapping of IMB IRQ Level [0..31] onto internal IRQ Level [0..39]
if (IMB_IRQ_Level < 0) & (IMB_IRQ_Level > 31)
    error('IMB IRQ Level out of allowed range [0..31]!',...
         ' (checked in: map_IMB_IRQ()) ');
end

if (IMB_IRQ_Level <= 7)
    internal_IRQ_level = IMB_IRQ_Level * 2 + 1;
elseif (IMB_IRQ_Level > 7)
    internal_IRQ_level = IMB_IRQ_Level + 8;
end
return;
```

B.2 build_APG.m

```
function build_APG(action_chain_gl_in, action_gl_in, Ts_in, file_name)
% build_APG(action_chain_gl, action_gl, Ts, file_name)
%
% inputs:
%   action_chain_gl ... action chains reflecting different transactions
%   action_gl ......... all single actions within the system
%   Ts ................ sample-rates of the model
%   file_name ......... file name of the model
%
% function converts given input information into a graphical
% representation, the external tool: DaVinci is started to
% visualize a graph-structure
%
% (c) 2003 Martin Orehek (RCS) (Martin.Orehek@rcs.ei.tum.de)
%
% clean global, to avoid problems with existing variables
% with the same name!
clear global
% globals to exchange data between local functions
global action_gl;
global recursion_level;
global action_chain_gl;

% initialize action_chain_gl, used for detection of LOOPS
% in action_chain_gl_in!!
for i = 1:length(action_chain_gl_in)
    action_chain_gl{i} = {[]};
end

action_gl = action_gl_in;

% constant values for graph text
cr = sprintf('\n');
tab_t = sprintf('\t');
nodeType = ['T2'];
edgeType = ['T1'];
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  EV driven part
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% build the transaction graph for the EV driven part using data     %%
%% from the activities                                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ( (length(action_chain_gl_in) - length(Ts_in)) ~=0 ) %is ev-driven
                                                    %part in model?
    recursion_level = 0;
    graph_text = [''];
    nodeTypeAttr_EV = ['a("_GO","rhombus"), a("COLOR","#ffbebe")'];
    for i = 1: (length(action_chain_gl_in) - length(Ts_in))
        unic_nodeName = ['ExtEV_',num2str(action_chain_gl_in{i}(1))];
        ExtMsgText = [action_gl(action_chain_gl_in{i}(1)).msg_name,...
                  '\\n','transaction ',num2str(i)];
        %%add starting external EV rhombus
        %EV rhombus!! OPEN
        graph_text = [graph_text,'l("', unic_nodeName,...
                  '",n("',nodeType,'",[a("OBJECT","',ExtMsgText,...
                  '"), ', nodeTypeAttr_EV,'],[',cr]; %open node
        %edges for sucessors:
        edgeText = [action_gl(action_chain_gl_in{i}(1)).msg_name];
        %edge OPEN
        graph_text = [graph_text,'l("',unic_nodeName,...
                  '->start",e("',edgeType,...
                  '",[a("OBJECT","''")],',cr];
        % add complet path for this starting ext_msg
        graph_text = [graph_text, buildPath(action_gl_in(...
                                    action_chain_gl_in{i}(1)), i)];
        %edge CLOSE
        graph_text = [graph_text,'))',cr];
        %EV rhombus!! CLOSE
        graph_text = [graph_text,'])),',cr]; %close node
    end
end %if ( (length(action_chain_gl_in) - length(Ts_in)) ~=0 )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  TT Cyclic part
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% build the transaction graph for the cyclic part                   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if ( length(Ts_in) ~= 0 ) % is cyclic part in model?
    recursion_level = 0;
    graph_text_cyclic = [''];
    nodeTypeAttr_TT = ['a("_GO","rhombus"), a("COLOR","#6fc0ff")'];
    for i = (length(action_chain_gl_in) - ....
           length(Ts_in) + 1): length(action_chain_gl_in)
        unic_nodeName = ['ExtEV_',num2str(action_chain_gl_in{i}(1))];
        CyclicEVText =[action_gl(action_chain_gl_in{i}(1)).msg_name,...
                   '\\n','transaction ',num2str(i)];
        %%add starting external EV rhombus
        %EV rhombus!! OPEN
        graph_text_cyclic = [graph_text_cyclic, 'l("', ...
                 unic_nodeName,'",n("',nodeType,...
                 '",[a("OBJECT","',CyclicEVText,'"), ', ...
                 nodeTypeAttr_TT,'],[',cr]; %open node
        %edges for sucessors:
        edgeText = [action_gl(action_chain_gl_in{i}(1)).msg_name];
        %edge OPEN
        graph_text_cyclic = [graph_text_cyclic,'l("',...
                 unic_nodeName,'->start",e("',edgeType,...
                 '",[a("OBJECT","''")],',cr];
        % add complet path for this starting ext_msg
        graph_text_cyclic = [graph_text_cyclic, ...
                 buildPath(action_gl_in(...
                          action_chain_gl_in{i}(1)), i)];
        %edge CLOSE
```

```
        graph_text_cyclic = [graph_text_cyclic,'))',cr];
        %EV rhombus!! CLOSE
        graph_text_cyclic = [graph_text_cyclic,'])),',cr]; %close node
    end
end %if ( length(Ts_in) ~= 0 ) % is cyclic part in model?

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Write into File!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%open file for writing the graph!
fid = fopen([file_name,'.status'],'w');
fprintf(fid,'[');

if ( (length(action_chain_gl_in) - length(Ts_in)) ~=0 ) %is ev-driven
                                                        %part in model?
    fprintf(fid,graph_text);
end %if ( (length(action_chain_gl_in) - length(Ts_in)) ~=0 )

if ( length(Ts_in) ~= 0 ) % is cyclic part in model?
    fprintf(fid,graph_text_cyclic);
end %if ( length(Ts) ~= 0 ) % is cyclic part in model?

fprintf(fid,']');
fclose(fid);

%start the DaVinci Tool!
eval(['!start ',file_name,'.status']);
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  newNode = buildPath(action, transaction_id)
% function returns the format string for a new node!
% input:  'action'  ... action with information for Node
% return: 'newNode' ... string for Node within DaVinci Tool
%  activity = struct('id',    msgID,  'priority',  priority, ...
%           'sucessor', [],  'comp_time',    NaN,     'thread',   tID);
global action_gl;
global recursion_level
global action_chain_gl

recursion_level = recursion_level + 1;

cr = sprintf('\n');
tab_t = sprintf('\t');
nodeName = ['A',num2str(action.id)];
unic_nodeName = [nodeName,'_', num2str(transaction_id)];
nodeType = ['T2'];

tab = tab_t;
for i = 1:recursion_level
    tab = [tab, tab_t];
end

if ( iscell(action_chain_gl{transaction_id}) )
    chain = cell2mat( action_chain_gl{transaction_id});
else
    chain = action_chain_gl{transaction_id};
end

if ( isempty(chain) | isempty( find(chain == action.id ) ) )
    % following action is NOT part of the chain already!
    % => can be added
    chain = [chain, action.id];
    action_chain_gl{transaction_id} = chain;
    nodeText = [nodeName,'\\n', 'priority = ',...
             num2str(action.priority),'\\n','C = ', ...
             num2str(action.comp_time),' msec\\n',...
             't_ID = ',num2str(action.thread)];
    nodeTypeAttr = ['a("_GO","circle")'];
    nodeEVTypeAttr = ['a("_GO","box")'];
```

```
    edgeType = ['T2'];
    newNode = [tab_t, 'l("', unic_nodeName,'",n("',nodeType,...
              '",[a("OBJECT","',nodeText,'"), ', nodeTypeAttr,...
              '],[',cr,tab]; %open node
    for i = 1:length(action.sucessor)
        sucessorName = ['A',num2str(action.sucessor(i))];
        unic_sucessorName = [sucessorName,'_',num2str(transaction_id)];
        %add edges for sucessors: "edge_1 -> BoxNode (EV) -> edge_2"
        edgeText = [action_gl(action.sucessor(i)).msg_name];
        %edge_1 OPEN
        newNode = [newNode,'l("',unic_nodeName,'_1->',...
                unic_sucessorName,'",e("',edgeType,...
                '",[a("OBJECT","''")],',cr,tab];
        %BoxNode (EV)!! OPEN
        newNode = [newNode, 'l("', unic_nodeName,'->',...
                unic_sucessorName,'",n("',nodeType,...
                '",[a("OBJECT","',edgeText,'"), ', ...
                nodeEVTypeAttr,'],[',cr,tab]; %open node
        %edge_2 OPEN
        newNode = [newNode,'l("',unic_nodeName,'_2->',...
                unic_sucessorName,'",e("',edgeType,...
                '",[a("OBJECT","''")],',cr,tab];
        newNode = [newNode, buildPath(action_gl(action.sucessor(i)),...
                                transaction_id)];
        %edge_2 CLOSE
        newNode = [newNode,tab,')),',cr, tab];
        %BoxNode (EV)!! CLOSE
        newNode = [newNode,']))',cr]; %close node
        %edge_1 CLOSE
        newNode = [newNode,tab,')),',cr, tab];
    end

    if (recursion_level == 1)
        %newNode = [newNode,']))),',cr]; %close node
        newNode = [newNode,']))',cr]; %close node
    else
        newNode = [newNode,']))',cr]; %close node
    end

else
    % following action IS already part of the chain !
    % => DO NOT ADD it, to avoid looping!
    newNode = [tab_t, 'r("', unic_nodeName,'")',cr,tab]; %reference to
                                                         %existing node
    warndlg(['PARSE WARNING: loop in transaction ',...
            num2str(transaction_id),' with action ',...
            nodeName,cr], 'activity loop');
end

recursion_level = recursion_level - 1;
return;
```

C Anhang: Realzeit-Analyse Skripten

Nachfolgend sind die zwei Skripten zur Berechnung der wors-case Reaktionszeiten einer konkreten Implementierung wiedergegeben. Die Unterschiede zwischen den zwei Skripten liegen in der angenommenen Prioritätsvererbungs-Strategie zwischen Nachrichten-Prioritäten und Server-Thread-Prioritäten (*BPI, Basic Priority Inheritence* und *PT, Preemption Threshold*).

C.1 responsetime_BPI.m

```
function transaction = responsetime_BPI(ES_in, ...
                                        Ts, ...
                                        action_chain_gl, ...
                                        action_in, ...
                                        n_T, ...
                                        n_A)
% transactions = ...
%        responsetime_BPI(ES, Ts, action_chain_gl, action_gl, n_T, n_A)
%
% inputs:
%   ES ................. cell with all EventStream definitions of
%                        the external action_chain_gl
%   Ts ................. sample-rates of the model
%   action_chain_gl ... action chains reflecting different transactions
%   action_gl ......... all single actions within the system
%   n_T ................ observed transaction
%   n_A ................ action of transaction n_T for which the
%                        worst-case-responsetime should be calculated
%
% output:
%   transactions ...... structure array of all transactions of
%                       the system (ev- and tt-part)
%
% function calculates the worst-case response time of:
%   action(n_A)
% beeing part of the transaction:
%   transaction(n_T)
% the underlying implementation is supposed to be a multi-threaded
% software-architecture, with a 'run-to-completion' semantic and
% a 'basic-priority-inheritence' policy (BPI) between
% message-priorities and  server-thread-priorities
% (=> patch 1 of the POSIX API of the RTOS: RTEMS-4.5.0!)
%
% (c) 2003 Martin Orehek (RCS) (Martin.Orehek@rcs.ei.tum.de)
%

% clean global, to avoid problems with existing variables
% with the same name!
clear global

global action
global transaction

cr = sprintf('\n');

action = action_in(1);
for i = 1:length(action_in);
    action(i) = action_in(i);
    action(i).comp_time = action(i).comp_time/1000; % [msec] -> [sec]
end
```

```
if ( isempty(find(action_chain_gl{n_T} == n_A)) )
    disp(['*** ERROR: action A_',num2str(n_A),...
         ' is NOT part of transaction T_', num2str(n_T), '!!', cr]);
    return;
end

% [ExtMsg] transactions!
transaction    = struct('id', [], 'ES', struct('z', [], 'a', []), ...
                    'action_chain', []);
for i = 1:(length(action_chain_gl) - length(Ts))
   transaction(i) = struct('id', i, 'ES', ES_in{i}, 'action_chain', ...
                       action_chain_gl(i));
%   EventStreamFigure(ES_in{i});
%   title(['Transaktion: TG_',num2str(i),', Ereignisfolge']);
end

% [..,cyclic] transaction!
j = 1;
for i = (length(action_chain_gl)-length(Ts)+1):length(action_chain_gl)
    % action_chain: for tBaseRate == A_base
    %               for tSubRateX == A_schedule, A_rateX
    %
    %ES is implicitely given with the different Sample-Rates of
    %the model
    transaction(i) = struct('id', i, ...
                        'ES', struct('z', Ts(j), 'a', 0), ...
                        'action_chain', action_chain_gl(i));
    j = j+1;
end

disp([cr, cr,'*** Basic Priority Inheritence Policy adopted!!']);
disp(['*** calculate worst case responsetime of action A_',...
      num2str(n_A),' in transaction T_', num2str(n_T), cr]);
level_i = action(n_A).priority;
T_A_i = n_T; %get_transaction(action(n));

B_i = get_blocking_time_BPI(level_i);
disp(['max blocking time of level-',...
      num2str(get_priority(action(n_A))),...
      ' is B(',num2str(n_A),'): ',num2str(B_i), cr])

WF = []; %wait to start action(n) to calculate finishing time
stopFlag = 0;
k = 1; % => instance of: action(n) in transaction: T_A_i
while( ~stopFlag )
    % calculate recursivelly the value for the worst case
    % start point of the k-th instance of action(n)
    WF_old = -1;
    if (k == 1)
        WF(k) = 0;   % ATTENTION: value must be less than
                  %            final calculated value!!
    else
        WF(k) = F(k-1); % start value is at least finishing
                     % time point of previous instance!
    end

    while((WF(k) ~= WF_old) & isfinite(WF(k)))
       WF_old = WF(k);
       WF(k) = B_i + ...
               I_other_transactions(WF(k), T_A_i, action(n_A)) + ...
               (k-1) * comp_time_transaction(transaction(T_A_i), ...
                                     action(n_A)) + ...
               (EventStream(transaction(T_A_i).ES, WF(k)) - k + 1) *...
             comp_time_without_sucessor(T_A_i, action(n_A));
    end

    if isinf(WF(k))
        disp([cr,'### ERROR: level-',num2str(n_A),...
             ' busy period is infinit!', cr])
        return
```

```
    end

    % finishing time for Multi-Thread implementation:
    % calculate recursivelly the value for the finishing time point of
    % the q-th instance of action(n)
    F_old = -1;
    F(k) = WF(k) ; % first approximation of finishing time
    while((F(k) ~= F_old) & isfinite(F(k)))
        F_old = F(k);
        F(k) = WF(k) + ...
          get_comp_time(action(n_A)) + ...
          I_other_transactions_BPI(F(k), WF(k), ...
                           T_A_i, action(n_A)) + ...
         I_own_transaction_BPI(F(k), WF(k), T_A_i, action(n_A));
    end % while

    if ( EventStream(transaction(T_A_i).ES, F(k)) > k )
        % new instance arrived in the level-i busy window
        k = k + 1;
        stopFlag = 0;
    else
        stopFlag = 1;
    end
end % while

% compute Arrival time of k instances of external event
% which triggers the analysed action
Arr = EventStreamArrivalTimes(transaction(T_A_i).ES, F(end));

%number of instances of action A_i within the busy window!!
q = length(Arr);

% calculate response time:
R = F - Arr;

disp(['there are',sprintf('\t'),'q_max = ',num2str(q),...
      sprintf('\t'),' iteration of action A_',num2str(n_A),cr]);
disp(['level-',num2str(n_A),' busy period is: ',num2str(F(end)),cr])
disp(['q-th instance of action:     ', sprintf('%7d\t\t',(1:q))])
disp(['arrival time are:            ', sprintf('%2.5f\t\t',Arr)])
disp(['starting time are:           ', sprintf('%2.5f\t\t',WF)])
disp(['finishing time :             ', sprintf('%2.5f\t\t',F)])
disp(['response time are:           ', sprintf('%2.5f\t\t',R)])
disp(['worst case response time is: ', sprintf('%2.5f\t\t',max(R))])

%draw figure of computational requests to the system
CompRequestsFigure(transaction, F(end));

return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function string = sprintValue(A_i)
num2str(WF)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_blocking_time_BPI(A_i)
% returns the blocking time of action Ai for the
% multi thread implementation with Basic Priority Inheritency Policy!
%
% input can be eighter: 'struct action(i)' or 'priority of level-i'
%
% Die maximale Blockierung durch Aktionen mit niederer Priorität
% hängt von der gesamten Konfiguration ab. Je nachdem welche
% Aktionen gemeinsam eine Message-Queue teilen können unterschiedliche
% Prioritätsvererbungen zu sehr ungünstigen Abhängigkeiten führen
% Bei jedem Thread kann eine Blockierung auftreten
% => max. Blockierung ist eine Summe (so viel Elemente wie Threads)
% aus den jeweiligen längsten Blockierungszeiten
% Hieraus ergibt sich der Vorteil durch 'preemption threshold'!!
%
global action %transaction
```

```
if isstruct(A_i)
    level_i = A_i.priority;
else
    level_i = A_i;
end

% get highes priority of action (message) of a certain thread.
% length of thread = number of threads
% and Thread(1) contains the highest priority of action in
% thread number 1
Thread = [];
for i = 1:length(action)
    if length(Thread) >= action(i).thread
        if Thread(action(i).thread) < get_priority(action(i))
            Thread(action(i).thread) = get_priority(action(i));
        end
    else
        Thread(action(i).thread) = get_priority(action(i));
    end
end
disp(['highest priority in different threads are: ', num2str(Thread)])

% get max. blocking time for all the different threads (message-queues)
% in the sw-architecture
c = 0;
for i = 1:length(action)
    if(Thread(action(i).thread) >= level_i) & ...
         (level_i > get_priority(action(i)))
        if (length(c) >= action(i).thread)
            if ( c(action(i).thread) < get_comp_time(action(i)))
                c(action(i).thread) = get_comp_time(action(i));
            end
        else
            c(action(i).thread) = get_comp_time(action(i));
        end
    end
end
disp(['blocking time for each thread in sw-architecture: ', ...
      num2str(c)])

y = sum(c); %here with BPI the sum of all possible blocking times!
disp(['blocking time is: ', num2str(y)])
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = I_other_transactions(W, T_of_A_i, A_i)
% interference other transactions
% returns the time consumed by other actions of other
% transactions (not T_of_A_i) with higer or same priority of Ai in the
% time window W
global action transaction

y = 0;
for i = 1:length(transaction)
    C = 0;
    if i ~= T_of_A_i % all transaction but the transaction with A_i
        for j = 1:length(transaction(i).action_chain)
            % all action with priority higher or equal
            if get_priority(...
             action(transaction(i).action_chain(j))) >= ...
               get_priority(A_i)
                C = C + ...
                 get_comp_time(...
                    action(transaction(i).action_chain(j)));
            end
        end
        y = y + EventStream(transaction(i).ES, W) * C;
    end
end
```

```
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = comp_time_without_sucessor(T_of_A_i, A_i)
% with check: is sucessor?!
% returns the time used for the computation of
% actions A_i from the actual transaction T_of_A_i
global action transaction

C = 0;
for j = 1:length(transaction(T_of_A_i).action_chain)
    % all action wich are NOT sucessors and have a higher
    % or equal priority contribute to the execution time
    if(~isSucessor(...
       action(transaction(T_of_A_i).action_chain(j)),A_i...
       ) & (get_priority( ...
           action( transaction(T_of_A_i).action_chain(j) ) ...
           ) >= get_priority(A_i) ) )

        C = C + ...
           get_comp_time(...
              action(transaction(T_of_A_i).action_chain(j)));
    end
end
y = C;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = comp_time_transaction(T, A_i)
% returns the time used for the computation of actions of the
% transactionchain with priority equal of higher than A_i
global action transaction

C = 0;
for j = 1:length(T.action_chain)
    % all action with higher or equal priority
    % contribute to the execution time
    if ((get_priority( action( T.action_chain(j) ) ) >= ...
        get_priority(A_i) ) )

        C = C + get_comp_time(action(T.action_chain(j)));
    end
end
y = C;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_priority(A_i)
% returns the priority of action A_i

y = A_i.priority;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_comp_time(A_i)
% returns the computation time of action A_i
%global action transaction

y = A_i.comp_time;
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_transaction(A_i)
% get transaction of A_i
global action transaction

y = [];
j = 1;
for i = 1:length(transaction)
    if ~isempty( find(transaction(i).action_chain == A_i.id ))
```

```
        y(j) = i;
        j = j+1;
    end
end
return;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = isSucessor(A_j, A_i, T_of_A_i)
% returns 1 if A_j is sucessor of A_i in transaction T_of_A_i
global action transaction

% if A_i and A_j are the same, than we suppose that we check a new
% iteration of q and therefore it is a sucessor (FIFO!)
if A_i.id == A_j.id
    y = 1;
    return
end

% check whether A_j is sucessor of A_i
% it is done recursively "down the tree"
for i = 1:length(A_i.sucessor)
    if A_i.sucessor(i) == A_j.id
        y = 1;
        return
    elseif isSucessor(A_j, action(A_i.sucessor(i))) % recursiv call!!
        y = 1;
        return
    end
end

y = 0;
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = I_other_transactions_BPI(F, W, T_of_A_i, A_i)
% interference other transactions
% returns the time consumed by other actions of other
% transactions with higer or same priority of Ai in the
% time window W and multitiasking ...
global action transaction

y = 0;
for i = 1:length(transaction)
    if i ~= T_of_A_i % all transaction but the transaction with A_i
        y = y + ...
            ( EventStreamMinus(transaction(i).ES, F) - ...
             EventStream(transaction(i).ES, W) ) * ...
            Lk_BPI(A_i, action(transaction(i).action_chain(1)));
          % Lk_BPI() is a recursive function!
    end
end
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = I_own_transaction_BPI(F, W, T_of_A_i, A_i)
% interference own transaction
% returns the time consumed by own actions of own
% transactions during the
% time window W and multitiasking ...
global action transaction

% get transaction of A_i
%T_of_A_i = get_transaction(A_i);

y = ( EventStreamMinus(transaction(T_of_A_i).ES, F) - ...
      EventStream(transaction(T_of_A_i).ES, W) ) * ...
    Lk_BPI(A_i, action(transaction(T_of_A_i).action_chain(1)));
    % Lk_BPI() is a recursive function!
return
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = Lk_BPI(A_i, A_j)
% recursive function to calculate the interference from higher prior
% actions for a multithread implementation
%
%
global action transaction

% in the worst-case action A_i runs during the whole execution with
% it's own (lowest!) priority, due to the fact that no higher message
% enters the queue of the thread from A_i!!
% => here the priority of A_i is the "threshold" for other actions!!
if (A_j.thread == A_i.thread) | (get_priority(A_j) < get_priority(A_i))
    C = 0;
else
    C = get_comp_time(A_j);
    if ~isempty(A_j.sucessor)
        for i = 1:length(A_j.sucessor)
            C = C + Lk_BPI(A_i, action(A_j.sucessor(i)));
        end
    end
end
y = C;
return
```

C.2 responsetime_PT.m

Da sich die Skripten zur Berechnung der worst-case Reaktions-Zeit nur in bestimmten Punkten bei der Realisierung unterschieden, sind nachfolgend lediglich die divergierenden Teile wiedergegeben.

```
function transaction = responsetime_PT(ES_in, ...
                                       Ts, ...
                                       action_chain_gl, ...
                                       action_in, ...
                                       n_T, ...
                                       n_A)
% transaction = ...
%        responsetime_PT(ES, Ts, action_chain_gl, action_gl, n_T, n_A)
%
% inputs:
%   ES ................. cell with all EventStream definitions of
%                        the external action_chain_gl
%   Ts ................. sample-rates of the model
%   action_chain_gl ... action chains reflecting different transactions
%   action_gl ......... all single actions within the system
%   n_T ................ observed transaction
%   n_A ................ action of transaction n_T for which the
%                        worst-case-responsetime should be calculated
%
% output:
%   transactions ...... structure array of all transactions of
%                        the system (ev- and tt-part)
%
% function calculates the worst-case response time of:
%   action(n_A)
% beeing part of the transaction:
%   transaction(n_T)
% the underlying implementation is supposed to be a multi-threaded
% software-architecture, with a 'run-to-completion' semantic and
% a 'preemption-threshold' policy (PT) between message-priorities and
% server-thread-priorities
% (=> patch 2 of the POSIX API of the RTOS: RTEMS-4.5.0!)
%
% (c) 2003 Martin Orehek (RCS) (Martin.Orehek@rcs.ei.tum.de)
%
```

```
{...}

disp([cr, cr,'*** Preemption Threshold Policy adopted!!']);
disp(['*** calculate worst case responsetime of action A_',...
      num2str(n_A),' in transaction T_', num2str(n_T), cr]);

{...}

B_i = get_blocking_time_PT(level_i);

{...}

        WF(k) = B_i + ...
                I_other_transactions(WF(k), T_A_i, action(n_A)) + ...
                (k-1) * comp_time_transaction(transaction(T_A_i), ...
                                    action(n_A)) + ...
                (EventStream(transaction(T_A_i).ES, WF(k)) - k + 1)*...
                  comp_time_without_sucessor(T_A_i, action(n_A));

{...}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_blocking_time_PT(A_i)
% returns the blocking time of action Ai for the
% multi thread implementation with Preemption Threshold Policy!
%
% input can be eighter: 'struct action(i)' or 'priority of level-i'
%
% Die maximale Blockierung durch Aktionen mit niederer
% Priorität hängt von der gesamten Konfiguration ab. Je
% nachdem welche Aktionen gemeinsam eine Message-Queue
% teilen kann jeweils eine max. Blockierzeit für jede
% Queue berechnet werden. Durch die Preemption Threshold
% Policy kann nur EIN Thread eine Blockierung  hervorrufen
% =>  max. Blockierung ist das MAXIMUM aller möglichen
% jeweiligen längsten Blockierungszeiten
%
global action %transaction

if isstruct(A_i)
    level_i = A_i.priority;
else
    level_i = A_i;
end

% get highes priority of action (message) of a certain thread == gamma
% length of thread = number of threads
% and Thread(1) contains the highest priority of action in thread
% number 1
Thread = [];
for i = 1:length(action)
    if length(Thread) >= action(i).thread
        if Thread(action(i).thread) < get_priority(action(i))
            Thread(action(i).thread) = get_priority(action(i));
        end
    else
        Thread(action(i).thread) = get_priority(action(i));
    end
end
disp(['highest priority in different threads are: ', num2str(Thread)])

% get max. blocking time for all the different threads (message-queues)
% in the sw-architecture
c = 0;
for i = 1:length(action)
    if(Thread(action(i).thread) >= level_i) & (level_i > ...
                                        get_priority(action(i)))
        if (length(c) >= action(i).thread)
            if ( c(action(i).thread) < get_comp_time(action(i)))
                c(action(i).thread) = get_comp_time(action(i));
```

```
            end
        else
            c(action(i).thread) = get_comp_time(action(i));
        end
    end
end
disp(['blocking time for each thread in sw-architecture: ', ...
      num2str(c)])

y = max(c); %here with PT only the maximum of all possible
            %blocking times!
disp(['blocking time is: ', num2str(y)])
return

{...}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = I_other_transactions_PT(F, W, T_of_A_i, A_i)
% interference form other transactions
% returns the time consumed by other actions of other
% transactions with higer or same priority of Ai in the
% time window W and multitiasking ...
global action transaction

y = 0;
for i = 1:length(transaction)
    if i ~= T_of_A_i % all transaction but the transaction with A_i
        y = y + ...
            ( EventStreamMinus(transaction(i).ES, F) - ...
             EventStream(transaction(i).ES, W) ) * ...
            Lk_PT(A_i, action(transaction(i).action_chain(1)));
           % Lk_PT() is a recursive function!
    end
end
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = I_own_transaction_PT(F, W, T_of_A_i, A_i)
% interference from own transaction
% returns the time consumed by own actions of own
% transactions during the
% time window W and multitiasking ...
global action transaction

% get transaction of A_i
%T_of_A_i = get_transaction(A_i);

y = ( EventStreamMinus(transaction(T_of_A_i).ES, F) - ...
      EventStream(transaction(T_of_A_i).ES, W) ) * ...
    Lk_PT(A_i, action(transaction(T_of_A_i).action_chain(1)));
    % Lk_PT() is a recursive function!
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = Lk_PT(A_i, A_j)
% recursive function to calculate the interference from higher prior
% actions for a multithread implementation
%
%
global action transaction

% due to PT during runntime the priority of the analysed action
% is rised to the GAMMA value by the run-time-system!!
% => an advantage compared to the BPI!
if (A_j.thread == A_i.thread) | (get_priority(A_j) < get_gamma(A_i))
    C = 0;
else
    C = get_comp_time(A_j);
    if ~isempty(A_j.sucessor)
        for i = 1:length(A_j.sucessor)
```

```
            C = C + Lk_PT(A_i, action(A_j.sucessor(i)));
        end
    end
end
y = C;
return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y = get_gamma(A_i)
% return the highest priority of messages arriving at the
% Queue of A_i
%
global action transaction

Thread = get_priority(A_i);
for i = 1:length(action)
    if A_i.thread == action(i).thread
        if Thread < get_priority(action(i))
            Thread = get_priority(action(i));
        end
    end
end
y = Thread;
return;
```

D Anhang: Realzeitbetriebsystem RTEMS-4.5.0

D.1 Erweiterungen des RTOS für Prioritätsvererbungs-Strategien

Kernel Funktionen

Bevor die einzelnen Änderungen des Laufzeitsystems vorgestellt werden, sind in Tabelle D.1 und Tabelle D.2 alle betroffenen Teile und die dazugehörigen Dateien mit Bemerkungen zusammengefasst.

Tabelle D.1: Geänderte Datenstrukturen und Funktionen des RTOS (RTEMS-4.5.0) zur Implementierung des *BPI-Protokolls* zwischen Nachrichten-Prioritäten und Server-Thread-Prioritäten.

Datenstruktur	`_CORE_message_queue_Control`
Datei	`<rtems-4.5.0>/c/src/exec/score/include/rtems/score/coremsg.h`
Bemerkungen	Erweiterung der Message-Queue Datenstruktur um Queue-Holder Felder.

Funktion	`mq_initialize_holder()`
Dateien	`<rtems-4.5.0>/c/src/exec/posix/include/mqueue.h` `<rtems-4.5.0>/c/src/exec/posix/src/mqueueopen.c`
Bemerkungen	Funktion zur Initialisierung der neuen Daten-Felder der Queue-Struktur. Wird *einmal* im Server-Thread während der Initialisierung aufgerufen.

Funktion	`_CORE_message_queue_Initialize_holder()`
Dateien	`<rtems-4.5.0> /c/src/exec/score/include/rtems/score/coremsg.h` `<rtems-4.5.0>/c/src/exec/score/src/coremsg.c`
Bemerkungen	Implementiert die Kernel-Funktion, die aus `mq_initialize_holder()` für die Initialisierung der neuen Daten-Felder aufgerufen wird.

Funktion	`_POSIX_Message_queue_Send_support()`
Datei	`<rtems-4.5.0>/c/src/exec/posix/src/mqueuesendsupp.c`
Bemerkungen	Erweiterung um den Nachrichten-Prioritäts-Check, zur Überprüfung einer möglichen Abbildung von Message-Prioritäten auf die Thread-Prioritäten.

Funktion	`_CORE_message_queue_Submit()`
Datei	`<rtems-4.5.0>/c/src/exec/score/src/coremsgsubmit.c`
Bemerkungen	Implementiert das Senden von Nachrichten auf Kernel-Ebene.

Funktion	`_CORE_message_queue_Insert_message()`
Datei	`<rtems-4.5.0>/c/src/exec/score/src/coremsginsert.c`
Bemerkungen	Implementiert das Einfügen von Nachricht in Warteschlangen auf Kernel-Ebene.

Funktion	`_CORE_message_queue_Seize()`
Datei	`<rtems-4.5.0>/c/src/exec/score/src/coremsgseize.c`
Bemerkungen	Implementiert das Entnehmen einer Nachricht aus einer Warteschlange auf Kernel-Ebene.

Tabelle D.2: Geänderte Datenstrukturen und Funktionen des RTOS (RTEMS-4.5.0) zur Implementierung des PT-Protokolls zwischen Nachrichten-Prioritäten und Server-Thread-Prioritäten

Datenstruktur	`_CORE_message_queue_Control`
Datei	`<rtems-4.5.0>/c/src/exec/score/include/rtems/score/coremsg.h`
Bemerkungen	Erweiterung der Message-Queue Dataenstruktur um die Threshold-Priorität

Funktion	`mq_initialize_threshold_priority()`
Dateien	`<rtems-4.5.0>/c/src/exec/posix/include/mqueue.h` `<rtems-4.5.0>/c/src/exec/posix/src/mqueueopen.c`
Bemerkungen	Funktion zur Initialisierung der neuen Daten-Felder der Queue-Struktur

Funktion	_CORE_message_queue_Initialize_threshold_prio()
Dateien <rtems-4.5.0> /c/src/exec/score/include/rtems/score/coremsg.h <rtems-4.5.0>/c/src/exec/score/src/coremsg.c	
Bemerkungen Implementiert die Kernel-Funktion zur Initialisierung der neuen Daten-Felder	

Funktion	_POSIX_Message_queue_Receive_support()
Datei <rtems-4.5.0>/c/src/exec/posix/src/mqueuerecvsupp.c	
Bemerkungen Erst beim Starten des Server-Threads durch den Scheduler wird in dieser Funktion die Priorität auf die der Queue zugeordnete Threshold-Priorität gesetzt.	

coremsg.h

```
{...}
/*
 *  The following defines the control block used to manage each
 *  counting message_queue.
 */
typedef struct {
  {...}
  /* extention for message priority inheritency (by Martin Orehek (RCS)) */
  Thread_Control                    *holder;
  Objects_Id                         holder_id;
  /* extention for message preemption-threshold (by Martin Orehek (RCS)) */
  Priority_Control                  threshold_priority;
}   CORE_message_queue_Control;

{...}

/*
 *  _CORE_message_queue_Initialize_holder
 *
 *  This routine sets the holder of a message queue.
 *
 *  Input parameters:
 *    the_message_queue            - the message queue to initialize
 *
 *  by Martin Orehek (RCS 2003)
 */
boolean _CORE_message_queue_Initialize_holder(
  CORE_message_queue_Control    *the_message_queue
);

/*
 *  _CORE_message_queue_Initialize_threashold_prio
 *
 *  This routine sets the holder of a message queue.
 *
 *  Input parameters:
 *    the_message_queue            - the message queue to initialize
 *
 *  by Martin Orehek (RCS 2003)
 */
boolean _CORE_message_queue_Initialize_threashold_prio(
      CORE_message_queue_Control    *the_message_queue,
      int                           psx_threshold_priority
```

```
);

{...}
```

coremsg.c

```
/*PAGE
 *
 *  _CORE_message_queue_Initialize
 *
 *  This routine initializes a newly created message queue based on the
 *  specified data.
 *
 *  Input parameters:
 *    the_message_queue            - the message queue to initialize
 *    the_class                    - the API specific object class
 *    the_message_queue_attributes - the message queue's attributes
 *    maximum_pending_messages     - maximum message and reserved buffer count
 *    maximum_message_size         - maximum size of each message
 *    proxy_extract_callout        - remote extract support
 *
 *  Output parameters:
 *    TRUE   - if the message queue is initialized
 *    FALSE  - if the message queue is NOT initialized
 */

boolean _CORE_message_queue_Initialize(
  CORE_message_queue_Control    *the_message_queue,
  Objects_Classes                the_class,
  CORE_message_queue_Attributes *the_message_queue_attributes,
  unsigned32                     maximum_pending_messages,
  unsigned32                     maximum_message_size,
  Thread_queue_Extract_callout   proxy_extract_callout
)
{
  unsigned32 message_buffering_required;
  unsigned32 allocated_message_size;

  {...}
  /* extention for message priority inheritency (by Martin Orehek (RCS)) */
  the_message_queue->holder                    = NULL;
  the_message_queue->holder_id                 = 0;
  /* extention for message preemption-threshold (by Martin Orehek (RCS)) */
  the_message_queue->threshold_priority        = -1;
  {...}
}

/*PAGE
 *
 *  _CORE_message_queue_Initialize_holder
 *
 *  This routine sets the holder of a message queue.
 *
 *  Input parameters:
 *    the_message_queue            - the message queue to initialize
 *
 *  by Martin Orehek (RCS 2003)
 */
boolean _CORE_message_queue_Initialize_holder(
  CORE_message_queue_Control    *the_message_queue
)
{
  Thread_Control                    *executing;

  if( the_message_queue == NULL)
    /*ERROR uninitialized Message Queue*/
    return FALSE;

  /* extention for message priority inheritency (by orehek) */
  executing = _Thread_Executing;
  the_message_queue->holder     = executing;
```

```
  the_message_queue->holder_id  = executing->Object.id;

  return TRUE;
}

/*PAGE
 *
 *  _CORE_message_queue_Initialize_threashold_prio
 *
 *  This routine sets the holder of a message queue.
 *
 *  Input parameters:
 *    the_message_queue            - the message queue to initialize
 *
 *  by Martin Orehek (RCS 2003)
 */
boolean _CORE_message_queue_Initialize_threashold_prio(
      CORE_message_queue_Control    *the_message_queue,
      int               psx_threshold_priority
)
{
  if( the_message_queue == NULL)
    /*ERROR uninitialized Message Queue*/
    return FALSE;

  if (psx_threshold_priority == -1)
    return TRUE; /* leave uninitialized => PT deactivated! */

  /* extention for preemption threashold (by orehek) */
  /* use RTEMS native priorities! */
  the_message_queue->threshold_priority = (Priority_Control)\
                      (255 - psx_threshold_priority);

  return TRUE;
}
```

coremsgsubmit.c

```
/*PAGE
 *
 *  _CORE_message_queue_Submit
 *
 *  This routine implements the send and urgent message functions. It
 *  processes a message that is to be submitted to the designated
 *  message queue.  The message will either be processed as a
 *  send message which it will be inserted at the rear of the queue
 *  or it will be processed as an urgent message which will be inserted
 *  at the front of the queue.
 *
 *  Input parameters:
 *    the_message_queue            - message is submitted to this message queue
 *    buffer                       - pointer to message buffer
 *    size                         - size in bytes of message to send
 *    id                           - id of message queue
 *    api_message_queue_mp_support - api specific mp support callout
 *    submit_type                  - send or urgent message
 *
 *  Output parameters:
 *    CORE_MESSAGE_QUEUE_SUCCESSFUL - if successful
 *    error code                     - if unsuccessful
 */

void _CORE_message_queue_Submit(
  CORE_message_queue_Control               *the_message_queue,
  void                                     *buffer,
  unsigned32                                size,
  Objects_Id                                id,
  CORE_message_queue_API_mp_support_callout  api_message_queue_mp_support,
  CORE_message_queue_Submit_types           submit_type,
  boolean                                   wait,
```

```
  Watchdog_Interval                        timeout
)
{
  ISR_Level                             level;
  CORE_message_queue_Buffer_control    *the_message;
  Thread_Control                       *the_thread;
  Thread_Control                       *executing;

  _Thread_Executing->Wait.return_code = CORE_MESSAGE_QUEUE_STATUS_SUCCESSFUL;

  if ( size > the_message_queue->maximum_message_size ) {
    _Thread_Executing->Wait.return_code =
       CORE_MESSAGE_QUEUE_STATUS_INVALID_SIZE;
    return;
  }

  /*
   *  Is there a thread currently waiting on this message queue?
   */

  if ( the_message_queue->number_of_pending_messages == 0 ) {
    the_thread = _Thread_queue_Dequeue( &the_message_queue->Wait_queue );
    if ( the_thread ) {
      _CORE_message_queue_Copy_buffer(
        buffer,
        the_thread->Wait.return_argument,
        size
      );
      *(unsigned32 *)the_thread->Wait.return_argument_1 = size;
      the_thread->Wait.count = submit_type;

#if (PRIORITY_INHERITY == 1)
      /* extention for message priority inheritency (by Martin Orehek 2002 (RCS))
*/
    /* set priority of holder thread equal message priority                    *
     * because thread is blocked and waiting
     * for priority calculation see:                                           *
     *   mqsendsupport.c => _POSIX_Message_queue_Priority_to_core( msg_prio )  *
     *   mqueue.inl                                                            *
     *     AND                                                                 *
     *   pthreadcreate.c =>                                                    *
     *     core_priority = _POSIX_Priority_To_core( schedparam.sched_priority ); *
     *   priority.inl                                                          */

      /* holder is equal to waiting thread AND
        sent message is NOT a default sent message (nativ API) */
      if ( (the_message_queue->holder == the_thread) && \
          (submit_type != CORE_MESSAGE_QUEUE_SEND_REQUEST) ) {
       /* NOT a default sent message (nativ API) */

       unsigned int posix_msg_priority = ((unsigned int) \
                                    abs((CORE_message_queue_Submit_types) sub-
mit_type));
       /* rtems priorities! */
       Priority_Control  core_msg_priority = (Priority_Control)(255 - po-
six_msg_priority);

       /* change priority always! */
       the_message_queue->holder->real_priority = core_msg_priority;
       _Thread_Change_priority( the_message_queue->holder, core_msg_priority,
FALSE);
       }
#endif

#if defined(RTEMS_MULTIPROCESSING)
      if ( !_Objects_Is_local_id( the_thread->Object.id ) )
        (*api_message_queue_mp_support) ( the_thread, id );
#endif
      return;
    }
```

```
  }

  /*
   *  No one waiting on the message queue at this time, so attempt to
   *  queue the message up for a future receive.
   */

  if ( the_message_queue->number_of_pending_messages <
       the_message_queue->maximum_pending_messages ) {

    the_message =
        _CORE_message_queue_Allocate_message_buffer( the_message_queue );

    /*
     *  NOTE: If the system is consistent, this error should never occur.
     */
    if ( !the_message ) {
      _Thread_Executing->Wait.return_code =
          CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED;
      return;
    }

    _CORE_message_queue_Copy_buffer(
      buffer,
      the_message->Contents.buffer,
      size
    );
    the_message->Contents.size = size;
    the_message->priority  = submit_type;

    _CORE_message_queue_Insert_message(
       the_message_queue,
       the_message,
       submit_type
    );
    return;
  }

  /*
   *  No message buffers were available so we may need to return an
   *  overflow error or block the sender until the message is placed
   *  on the queue.
   */

  if ( !wait ) {
    _Thread_Executing->Wait.return_code = CORE_MESSAGE_QUEUE_STATUS_TOO_MANY;
    return;
  }

  executing = _Thread_Executing;

  _ISR_Disable( level );
  _Thread_queue_Enter_critical_section( &the_message_queue->Wait_queue );
  executing->Wait.queue              = &the_message_queue->Wait_queue;
  executing->Wait.id                 = id;
  executing->Wait.return_argument    = (void *)buffer;
  executing->Wait.return_argument_1  = (void *)size;
  executing->Wait.count              = submit_type;
  _ISR_Enable( level );

  _Thread_queue_Enqueue( &the_message_queue->Wait_queue, timeout );
}
```

coremsginsert.c

```
/*PAGE
 *
 *  _CORE_message_queue_Insert_message
 *
```

```
 *  This kernel routine inserts the specified message into the
 *  message queue.  It is assumed that the message has been filled
 *  in before this routine is called.
 *
 *  Input parameters:
 *    the_message_queue - pointer to message queue
 *    the_message       - message to insert
 *    priority          - insert indication
 *
 *  Output parameters:  NONE
 *
 *  INTERRUPT LATENCY:
 *    insert
 */

void _CORE_message_queue_Insert_message(
  CORE_message_queue_Control        *the_message_queue,
  CORE_message_queue_Buffer_control *the_message,
  CORE_message_queue_Submit_types    submit_type
)
{
  int position = 0;

  the_message_queue->number_of_pending_messages += 1;

  the_message->priority = submit_type;

  switch ( submit_type ) {
    case CORE_MESSAGE_QUEUE_SEND_REQUEST:
      _CORE_message_queue_Append( the_message_queue, the_message );
      break;
    case CORE_MESSAGE_QUEUE_URGENT_REQUEST:
      _CORE_message_queue_Prepend( the_message_queue, the_message );
      break;
    default:
      /* XXX interrupt critical section needs to be addressed */
      {
        CORE_message_queue_Buffer_control *this_message;
        Chain_Node                        *the_node;
        Chain_Control                     *the_header;

        the_header = &the_message_queue->Pending_messages;
        the_node = the_header->first;
        while ( !_Chain_Is_tail( the_header, the_node ) ) {

          this_message = (CORE_message_queue_Buffer_control *) the_node;

          if ( this_message->priority <= the_message->priority ) {
            the_node = the_node->next;
           position++;
            continue;
          }

          break;
        }
        _Chain_Insert( the_node->previous, &the_message->Node );
      }

#if (PRIORITY_INHERITY == 1)
     /* extention for message priority inheritency (by Martin Orehek 2002 (RCS)) *
      * if the message inserted in the queue is on the first position than       *
      * set priority of holder thread as high as the message priority            *
      * thread has to have at least the priority  of the highest waiting         *
      * message in the queue => prevend priority inversion!                      *
      * for priority calculation see:                                            *
      *   mqsendsupport.c => _POSIX_Message_queue_Priority_to_core( msg_prio )   *
      *   mqueue.inl                                                             *
      *     AND                                                                  *
      *   pthreadcreate.c =>                                                     *
      *     core_priority = _POSIX_Priority_To_core( schedparam.sched_priority );*
```

```
      *   priority.inl                                                          */
      if (position == 0) { /* new message was entered in the first position */
       /* holder is equal to waiting thread AND
        * sent message is NOT a default sent message (nativ API) */
       if ( (the_message_queue->holder != NULL) && \
            (the_message_queue->priority != CORE_MESSAGE_QUEUE_SEND_REQUEST) )
            /* NOT a default sent message (nativ API) */
         {
           unsigned int posix_msg_priority = ((unsigned int) \
                        abs((CORE_message_queue_Submit_types) the_message-
>priority));

           /* rtems priorities! */
           Priority_Control  core_msg_priority = (Priority_Control)(255 - po-
six_msg_priority);
           if ( core_msg_priority < the_message_queue->holder->current_priority ) {
             /* for mutex termIOS problem: */
             the_message_queue->holder->real_priority = core_msg_priority;
             _Thread_Change_priority( the_message_queue->holder, core_msg_priority,
FALSE);
           }
         }
      }
#endif

      break;
  }

  /*
   *  According to POSIX, does this happen before or after the message
   *  is actually enqueued.  It is logical to think afterwards, because
   *  the message is actually in the queue at this point.
   */

  if ( the_message_queue->number_of_pending_messages == 1 &&
       the_message_queue->notify_handler )
    (*the_message_queue->notify_handler)( the_message_queue->notify_argument );
}
```

coremsgseize.c

```
/*PAGE
 *
 *  _CORE_message_queue_Seize
 *
 *  This kernel routine dequeues a message, copies the message buffer to
 *  a given destination buffer, and frees the message buffer to the
 *  inactive message pool.  The thread will be blocked if wait is TRUE,
 *  otherwise an error will be given to the thread if no messages are available.
 *
 *  Input parameters:
 *    the_message_queue - pointer to message queue
 *    id                - id of object we are waitig on
 *    buffer            - pointer to message buffer to be filled
 *    size              - pointer to the size of buffer to be filled
 *    wait              - TRUE if wait is allowed, FALSE otherwise
 *    timeout           - time to wait for a message
 *
 *  Output parameters:  NONE
 *
 *  NOTE: Dependent on BUFFER_LENGTH
 *
 *  INTERRUPT LATENCY:
 *    available
 *    wait
 */

void _CORE_message_queue_Seize(
  CORE_message_queue_Control      *the_message_queue,
  Objects_Id                       id,
```

```
  void                              *buffer,
  unsigned32                        *size,
  boolean                            wait,
  Watchdog_Interval                  timeout
)
{
  ISR_Level                           level;
  CORE_message_queue_Buffer_control *the_message;
  Thread_Control                    *executing;
  Thread_Control                    *the_thread;

  executing = _Thread_Executing;
  executing->Wait.return_code = CORE_MESSAGE_QUEUE_STATUS_SUCCESSFUL;
  _ISR_Disable( level );
  if ( the_message_queue->number_of_pending_messages != 0 ) {
    the_message_queue->number_of_pending_messages -= 1;

    the_message = _CORE_message_queue_Get_pending_message( the_message_queue );
    _ISR_Enable( level );

    *size = the_message->Contents.size;
    _Thread_Executing->Wait.count = the_message->priority;
    _CORE_message_queue_Copy_buffer(the_message->Contents.buffer,buffer,*size);

#if (PRIORITY_INHERITY == 1)
    /* extention for message priority inheritency (by Martin Orehek 2002 (RCS)) *
     * if current thread priority is higher than the highes message *
     * priority, set priority lower  *
     * for priority calculation see:                                              *
     *   mqsendsupport.c => _POSIX_Message_queue_Priority_to_core( msg_prio )     *
     *   mqueue.inl                                                               *
     *     AND                                                                    *
     *   pthreadcreate.c =>                                                       *
     *     core_priority = _POSIX_Priority_To_core( schedparam.sched_priority ); *
     *   priority.inl                                                             */

    /* holder is equal to waiting thread AND
     * sent message is NOT a default sent message (nativ API) */
    if ( (the_message_queue->holder != NULL) && \
        (the_message->priority != CORE_MESSAGE_QUEUE_SEND_REQUEST) )
      /* NOT a default sent message (nativ API) */
      {
      unsigned int posix_msg_priority = ((unsigned int) \
                  abs((CORE_message_queue_Submit_types) the_message->priority));

      /* rtems priorities! */
      Priority_Control  core_msg_priority = (Priority_Control)(255 - po-
six_msg_priority);

      if (the_message_queue->holder != executing) {
        /* ERROR, for debugging, block system here, *
         * need to be handled in the final version! */
        while (1)
          ;
      }

      /* change always priority! */
      the_message_queue->holder->real_priority = core_msg_priority;
      _Thread_Change_priority( the_message_queue->holder, core_msg_priority, TRUE);

      }
#endif
    /*
     *  There could be a thread waiting to send a message.  If there
     *  is not, then we can go ahead and free the buffer.
     *
     *  NOTE: If we note that the queue was not full before this receive,
     *  then we can avoid this dequeue.
     */
```

```
    the_thread = _Thread_queue_Dequeue( &the_message_queue->Wait_queue );
    if ( !the_thread ) {
      _CORE_message_queue_Free_message_buffer( the_message_queue, the_message );
      return;
    }

    /*
     *  There was a thread waiting to send a message.  This code
     *  puts the messages in the message queue on behalf of the
     *  waiting task.
     */

    the_message->priority  = the_thread->Wait.count;
    the_message->Contents.size = (unsigned32)the_thread->Wait.return_argument_1;
    _CORE_message_queue_Copy_buffer(
      the_thread->Wait.return_argument,
      the_message->Contents.buffer,
      the_message->Contents.size
    );

    _CORE_message_queue_Insert_message(
       the_message_queue,
       the_message,
       the_message->priority
    );
    return;
  }

  if ( !wait ) {
    _ISR_Enable( level );
    executing->Wait.return_code = CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED_NOWAIT;
    return;
  }

  _Thread_queue_Enter_critical_section( &the_message_queue->Wait_queue );
  executing->Wait.queue              = &the_message_queue->Wait_queue;
  executing->Wait.id                 = id;
  executing->Wait.return_argument    = (void *)buffer;
  executing->Wait.return_argument_1  = (void *)size;
  /* Wait.count will be filled in with the message priority */
  _ISR_Enable( level );

  _Thread_queue_Enqueue( &the_message_queue->Wait_queue, timeout );
}
```

mqueuerecvsupp.c

```
/*PAGE
 *
 *  _POSIX_Message_queue_Receive_support
 *
 *  NOTE: XXX Document how size, priority, length, and the buffer go
 *        through the layers.
 */

ssize_t _POSIX_Message_queue_Receive_support(
  mqd_t               mqdes,
  char               *msg_ptr,
  size_t              msg_len,
  unsigned int       *msg_prio,
  Watchdog_Interval   timeout
)
{
  register POSIX_Message_queue_Control *the_mq;
  Objects_Locations                     location;
  unsigned32                            length_out;

  the_mq = _POSIX_Message_queue_Get( mqdes, &location );
  switch ( location ) {
    case OBJECTS_ERROR:
```

```
      set_errno_and_return_minus_one( EBADF );
    case OBJECTS_REMOTE:
      _Thread_Dispatch();
      return POSIX_MP_NOT_IMPLEMENTED();
      set_errno_and_return_minus_one( EINVAL );
    case OBJECTS_LOCAL:
      if ( (the_mq->oflag & O_ACCMODE) == O_WRONLY ) {
        _Thread_Enable_dispatch();
        set_errno_and_return_minus_one( EBADF );
      }

      if ( msg_len < the_mq->Message_queue.maximum_message_size ) {
        _Thread_Enable_dispatch();
        set_errno_and_return_minus_one( EMSGSIZE );
      }

      /*
       *  Now if something goes wrong, we return a "length" of -1
       *  to indicate an error.
       */

      length_out = -1;

      _CORE_message_queue_Seize(
        &the_mq->Message_queue,
        mqdes,
        msg_ptr,
        &length_out,
        (the_mq->oflag & O_NONBLOCK) ? FALSE : TRUE,
        timeout
      );

      _Thread_Enable_dispatch();

#if (PREEMPTION_THRESHOLD == 1)
      {
       Thread_Control                    *executing;
       CORE_message_queue_Control      *the_message_queue = &the_mq->Message_queue;

       _Thread_Disable_dispatch();
       executing = _Thread_Executing;

       if ( (the_message_queue->holder != NULL) && \
            (the_message_queue->threshold_priority != -1) )
         {
           if (the_message_queue->holder != executing) {
             /* ERROR, for debugging, block system here, *
              * need to be handled in the final version! */
             while (1)
              ;
           }

           /* change always priority to threshold priority! */
           the_message_queue->holder->real_priority = the_message_queue-
>threshold_priority;
           _Thread_Change_priority( the_message_queue->holder, \
                                    the_message_queue->threshold_priority, \
                                    TRUE);
         }

       _Thread_Enable_dispatch();
      }
#endif

      *msg_prio =
        _POSIX_Message_queue_Priority_from_core(_Thread_Executing->Wait.count);

      if ( !_Thread_Executing->Wait.return_code )
        return length_out;
```

```
      set_errno_and_return_minus_one(
        _POSIX_Message_queue_Translate_core_message_queue_return_code(
          _Thread_Executing->Wait.return_code
        )
      );
  }
  return POSIX_BOTTOM_REACHED();
}
```

POSIX-API Funktionen

mqueue.h

```
{...}
/*
 *  Initialize Holder of a Message Queue, for Basic Priority Inheritence
 *  MessagePrirority -> ThreadPriority
 *  extension by Martin Orehek (RCS 2003)
 *
 *  Usage: should be called only ONCE per Messagequeue in the
 *         corresponding Server Thread!!
 *         If Not Used => holder remains NULL => Basic Priority inheritence
 *         is DisActivated!! (message priority does NOT affect thread priority)
 */
boolean mq_initialize_holder(mqd_t mq_id);

/*
 *  Initialize Threshold Priority of a Message Queue,
 *  for Preemtion Threshold Policy
 *  extension by Martin Orehek (RCS 2003)
 *
 *  Usage: should be called only ONCE per Messagequeue in the
 *         corresponding Server Thread!!
 *         If Not Used => Threshold Priority will be -1
 *         PT is DisActivated!! (priority does NOT affect thread priority)
 */
boolean mq_initialize_threshold_priority(mqd_t mq_id, int threshold_priority);
{...}
```

mqueueopen.c

```
{...}

boolean mq_initialize_holder(mqd_t mq_id)
{
  POSIX_Message_queue_Control *the_mq;
  Objects_Locations                    location;

  _Thread_Disable_dispatch();

  the_mq = _POSIX_Message_queue_Get( mq_id, &location );

    switch ( location ) {
    case OBJECTS_ERROR:
      /* ERORR: wrong Object */
      return FALSE;

    case OBJECTS_REMOTE:
      /* ERORR: no MP support */
      return FALSE;

    case OBJECTS_LOCAL:
       if ( !_CORE_message_queue_Initialize_holder(
           &the_mq->Message_queue
         ) ) {
        /* ERORR: initialization not possible*/
        return FALSE;
       }
       /* first for:  _Thread_Disable_dispatch(); */
```

```
       /* second for: _Objects_Get behind
                     _POSIX_Message_queue_Get( mq_id, &location );*/
       _Thread_Enable_dispatch();
       _Thread_Enable_dispatch();
       return TRUE;
    }

    return POSIX_BOTTOM_REACHED();
}

boolean mq_initialize_threshold_priority(mqd_t mq_id, \
               int threshold_priority)
{
  POSIX_Message_queue_Control *the_mq;
  Objects_Locations                    location;

  _Thread_Disable_dispatch();

  the_mq = _POSIX_Message_queue_Get( mq_id, &location );

    switch ( location ) {
    case OBJECTS_ERROR:
      /* ERORR: wrong Object */
      return FALSE;

    case OBJECTS_REMOTE:
      /* ERORR: no MP support */
      return FALSE;

    case OBJECTS_LOCAL:
       if ( !_CORE_message_queue_Initialize_threashold_prio( \
                    &the_mq->Message_queue,
                    threshold_priority) ) {
        /* ERORR: initialization not possible*/
        return FALSE;
       }
       /* first for:  _Thread_Disable_dispatch(); */
       /* second for: _Objects_Get behind
                     _POSIX_Message_queue_Get( mq_id, &location );*/
       _Thread_Enable_dispatch();
       _Thread_Enable_dispatch();
       return TRUE;
    }

    return POSIX_BOTTOM_REACHED();
}

{...}
```

mqsendsupp.c

```
/*PAGE
 *
 *  _POSIX_Message_queue_Send_support
 */

int _POSIX_Message_queue_Send_support(
  mqd_t               mqdes,
  const char         *msg_ptr,
  unsigned32          msg_len,
  unsigned32          msg_prio,
  Watchdog_Interval   timeout
)
{
  register POSIX_Message_queue_Control *the_mq;
  Objects_Locations                    location;

{...}

#if (PRIORITY_INHERITY == 1)
```

```
  /* extension for the priority inheritence of message priorities to threads! *
   * make sure that message priority is valid POSIX thread priority!          *
   * _POSIX_Priority_Is_valid(msg_prio) [1..254]                              */
  if (msg_prio < 1 || msg_prio > 254)
    set_errno_and_return_minus_one( EINVAL );
#endif
{...}
```

D.2 RTOS-Overhead (RTEMS-4.5.0 auf MPC555)

Nachfolgend ist eine Zusammenfassung der gemessenen Ausführungszeiten für Realzeitbetriebssystem-Aufrufe von RTEMS-4.5.0 auf dem Mikrocontroller MPC555 wiedergegeben. Diese Zeiten werden im Realzeitnachweis als Konstanten gezielt hinzuaddiert, um die bei der Co-Simulation nicht mitgemessenen RTOS-Aufrufe in der Analyse zu berücksichtigen.

Die Messungen wurden entweder mittels digitalen Ausgangssignalen und einem Oszilloskop durch die Zeitmessfunktion des verwendeten Debuggers (Trace32 von Lauterbach Datentechnik GmbH) oder durch die Timing-Test-Programme (tm01 bis tm27) von RTEMS (Anpassung auf Floating-Point Task durchgeführt) ermittelt. Sie spielgeln lediglich einen realistischen Wert wieder, nicht aber den worst-case Fall, der nur durch relativ hohem Aufwand bestimmt werden kann. Genauere Daten können mit anderen Techniken, wie z.B. in [122] beschrieben, ermittelt werden.

Die verwendeten Einstellungen des Mikrocontrollers:

- IMB-Busfrequenz: 40 MHz
- Speicherzugriffe mit 0 Wait-States
- Seriallisierung: off

Interrupt-Verarbeitungs-Overhead

Beschreibung	**Zeit [µsec]**	**Herkunft**
interrupt exit overhead: returns to interrupted task	6,8	tm27
interrupt exit overhead: returns to nested interrupt	6	tm27
IRQ Entry RTOS + IRQ-MPC555 ExtException Handling ... 1.Line ISR_Code (für IMB + 2,4 µsec)	14,6	psx_phycore_timing_test
IRQ Exit MPC555 ExtException Handling!	6,1	Lauterbach: Ende PIT->nested,return!

Der Interrupt Verarbeitungs-Overhead ergibt sich somit zu:
C_ISR_overhead = 14,6 + 6,1 + max(6,8; 6) = 27,5 µsec

Task-Switch-Overhead

Beschreibung	Zeit [µsec]	Herkunft
rtems_semaphore_obtain: available	14,8	tm01
rtems_semaphore_obtain: not available -- NO_WAIT	14,8	tm01
fp context switch: restore 1st FP task	34,4	tm26
fp context switch: save idle, restore initialized	31,2	tm26
fp context switch: save idle, restore idle	44,8	tm26
fp context switch: save initialized, restore initialized	31,2	tm26
rtems_message_queue_receive: available	25,2	tm09
rtems_message_queue_receive: not available -- NO_WAIT	16,4	tm10

Der Overhead für tBase ergibt sich somit zu:

C_tBase_overhead = TaskSwitch + sem_wait ++ sem_wait + TaskSwitch

C_tBase_overhead = max(34,4; 31,2; 44,8; 31,2) + 14,8 + 14,8 + max(34,4; 31,2; 44,8; 31,2)

C_tBase_overhead = 119,2 µsec

Der Overhead für tRateX ergibt sich somit zu:

C_tRateX_overhead = sem_wait ++ sem_wait + TaskSwitch

C_tRateX_overhead = 14,8 + 14,8 + max(34,4; 31,2; 44,8; 31,2)

C_tRateX_overhead = 74,4 µsec

Der Overhead für die erste Aktion eines APG ergibt sich somit zu:

C_first_EV_action_overhead = TaskSwitch + mq_receive ++ mq_receive + TaskSwitch

C_first_EV_action_overhead = max(34,4; 31,2; 44,8; 31,2) + 25,2 + 16,4 + max(34,4; 31,2; 44,8; 31,2)

C_first_EV_action_overhead = 131,2 µsec

Der Overhead für die anderen Aktion eines APG ergibt sich somit zu:

C_other_EV_action_overhead = mq_receive ++ mq_receive + TaskSwitch

C_ other _EV_action_overhead = 25,2 + 16,4 + max(34,4; 31,2; 44,8; 31,2)

C_ other _EV_action_overhead = 86,4 µsec