DIPLOMARBEIT

Formale Beschreibung einer IEC 61499-Laufzeitumgebung unter Berücksichtigung von Echtzeitanforderungen und dem unterlagerten Betriebssystem

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs

unter der Leitung von

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Markus Vincze und
Dipl.-Ing. Christoph Sünder

E376
Institut für Automatisierungs- und Regelungstechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektro- und Informationstechnik

von

Ivo Gosetti
9925488
St. Georgstr. 10
39011 Lana (BZ)
Italien

Wien, im *September 2007* _____

Kurzfassung

Der Einsatz von Verifikation wird in der heutigen Zeit in industriellen Unternehmen ein immer wichtiger werdender Punkt für die Erkennung und Eliminierung von Fehlern vor Beginn der Produktion von Produkten. Anhand von Verifikationsmethoden können diese Fehler vermieden werden, was zu einer Senkung der Entwurfskosten führt.

Mit dieser Arbeit wird eine formale Beschreibung einer IEC 61499 Laufzeitumgebung und einem darunter liegenden Betriebssystem durchgeführt, wobei die Eigenschaft der Echtzeitfähigkeit einfließt. Für die formale Verifikation wird die Methode des Model-Checking angewandt. Mit dem Model-Checking werden formale Modelle auf dessen Verhalten und Spezifikation überprüft.

Nach einem theoretischen Einblick in die IEC 61499 μCrons Laufzeitumgebung des Instituts für Automatisierungs- und Regelungstechnik an der Technischen Universität Wien und in das Echtzeitbetriebssystem eCos wird ein Konzept für die Erstellung der jeweiligen formalen Modelle erarbeitet. Dabei wird der Modellierungsansatz der Net Condition Event Systems verwendet, da diese im Zusammenhang mit dem Standard IEC 61499 die größte Verbreitung haben. Bei der Modellierung wurde auf einen sequentiellen Ereignisfluss geachtet, um mittels Verzögerungen die reale Abarbeitungszeit erfassen zu können.

Unter Verwendung von Aktivitätsdiagrammen werden anfangs die erstellten formalen Modelle präsentiert. Es wird auf dessen Konzepterstellung und Verhalten eingegangen. Neben der Realisierung und Beschreibung der formalen Modelle für die Laufzeitumgebung und das Betriebssystem, wird im besonderen auch auf deren Zusammenspiel eingegangen. Die Arbeit wird mit der Modellierung eines Anwendungsbeispiels abgeschlossen, wo ein Funktionsblocknetzwerk über zeitgesteuerte und extern getriggerte Ereignisse angesteuert wird. Unter Durchführung von unterschiedlichen Abfragen wird die Anwendung auf deren Korrektheit und Verhalten überprüft.

Abstract

The use of verification nowadays becomes more and more important in industrial enterprises. This is because it helps to identify and eliminate the errors before the start of the production. These errors can be avoided by means of the verification methods, which at the same time can lead to decreasing development costs.

This thesis realises a formal description of an IEC 61499 runtime environment and an underlying operation system, whereas the attribute of real-time capabilities is incorporated. The method of model-checking is used for the formal verification. The formal models are checked by use of this method for their behaviour and specification.

Primarily a theoretical overview of the IEC 61499 μCrons runtime environment established by the automation and control institute at Technical University of Vienna and the real-time operation system eCos is given. Afterwards a concept for the realisation of respective formal models is elaborated. For the realisation of the formal models a modelling approach of Net Condition Event Systems is used. The reason for this is that they are often used in connection with the standard IEC 61499. During the modelling it was important to pay attention to the sequential event flow, because of the registration of the real execution time by the means of delays.

The development of the formal models is presented using activity diagrams. The elaboration of concept and behaviour are shown more detailed. Behind the realisation and the description of the formal models for the runtime environment and the operation system, attention is turned especially to their interaction. The thesis is concluded with the modelling of an example for an application, where a function block network is driven through time triggered and externally triggered events. This application is checked for its correctness and behaviour using different enquiries.

Sintesi

L'uso della procedura di verificazione nell'odierna impresa industriale diventa un punto sempre più importante per riconoscere ed eliminare errori che insorgono all'inizio del processo produttivo. Questi errori possono essere evitati attraverso l'utilizzo di diversi metodi di verificazione che conducono a una diminuzione di costi di progettazione.

Con questa tesi viene eseguita una formale descrizione di una funzione temporale basata sullo standard IEC 61499 ed un sistema operativo in cui è rilevante l'influsso della capacità del tempo reale. Per la verificazione formale viene usato il metodo del model-checking, funzionale anche alla verifica del comportamento e delle specificità dei modelli formali.

La prima fase della tesi è consistita in uno studio teorico della funzione temporale μCrons basata sullo standard IEC 61499, che è stata realizzata all'istituto d'automazione e controllo presso la Technische Universität di Vienna, e del sistema operativo eCos. Dopodiché è stato elaborato un concetto per la realizzazione dei rispettivi modelli formali per la cui costruzione viene utilizzato il metodo della modellazione dei Net Condition Event Systems. Questa modellazione è stata scelta per la grande diffusione in coerenza con lo standard IEC 61499. Durante il processo di modellazione è stato rispettato un flusso d'evento sequenziale per comprendere attraverso ritardi temporali il tempo reale di un'elaborazione.

In primo luogo, attraverso l'utilizzo di diagrammi d'attività, vengono presentati i modelli formali cui fa seguito una dettagliata spiegazione del loro concetto e contenuto. Oltre la realizzazione e descrizione dei modelli formali della funzione temporale e del sistema operativo, viene altrettanto approfondita la loro interazione. La tesi si conclude con la modellazione di un esempio d'applicazione. Questa applicazione contiene altrettanto una rete di blocchi funzionali che viene eseguita tramite eventi temporali e esterni. Con un'esecuzione di diverse prove l'applicazione viene verificata nella sua correttezza e comportamento.

Danksagung

Für die Erstellung dieser Arbeit möchte ich mich an dieser Stelle bei allen bedanken, die mir im Technischen und Persönlichen beigestanden sind.

Allen voran möchte ich mich bei Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Markus Vincze für die Begutachtung der Arbeit bedanken. Bei meinem Betreuer Dipl.-Ing. Christoph Sünder möchte ich mich für die technischen Ratschläge und für das ständige Engagement bei anfälligen Problemen bedanken. Dem gesamten Personal am Institut für Automatisierungs- und Regelungstechnik möchte ich meinen Dank für das angenehme Arbeitsklima aussprechen.

Bei Dr.Eng.Dr.Sci. Valeriy Vyatkin möchte ich mich für die Verwendung seines Programmes ViVe (Visual Verifier) und bei auftretenden Fehleren im Programm für seine schnellen Fehlerbehebungen bedanken.

Ein besonderes Dankeschön möchte ich meiner Familie aussprechen.
Meiner Mutter Karin und meinem Vater Gilberto danke ich für die finanzielle und moralische Unterstützung während meiner Ausbildung.
Meiner Freundin Mag. Yuliya Lissitsina danke ich für das Korrekturlesen der Arbeit und dafür, dass sie immer an mich geglaubt hat.
Ganz besonders möchte ich mich noch bei meiner Großmutter Gertrude bedanken, für die guten Ratschläge, die mich auf meinem Lebensweg begleiten werden und auch für die finanzielle Zuwendung.

Widmung

Diese Diplomarbeit widme ich meiner Familie.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In der heutigen Zeit fordern die Globalisierung der Märkte und der Konkurrenzdruck aus Niedriglohnländern industrielle Unternhemen auf, deren Produktion so flexibel wie möglich zu gestalten. Für die Herstellung von Produkten sind die Kosten erheblich höher als jene von ostasiatischen Mitbewerbern. So kommt es zu einem massiven Produktimport aus Asien, wodurch Produkte in Europa bald nicht mehr konkurrenzfähig sein werden. Konkurrenzfähig kann etwa ein Wettbewerbspartner in Europa sein, indem er schneller als ihre Wettbewerber bestimmte Innovationen auf den Markt bringen kann. Die Schnelligkeit reicht jedoch nicht aus, denn es spielen auch die Faktoren Kosten und „Time to Market“ eine wichtige Rolle gegenüber hoher Qualität der Produkte und Flexibilität im Unternehmen. Somit gilt es innovative Produkte zu vertretbaren Preisen herzustellen, wobei ein Augenmerk auf die totalen Produktionskosten gelegt werden müssen [FBZ05].
Ist man gewillt, ein innovatives Produkt auf den Markt zu bringen, so kann es durchaus vorkommen, dass es bei dessen Produktion in der Anfangsphase zu Fehlern kommt. Fehler, die am häufigsten am Anfang der Produktion auftreten sind Programmierfehler. Dabei gilt es solche Fehler vor Beginn der Produktion zu erkennen und zu eliminieren. Um solche Fehler zu beheben, können Verifikationsmethoden angewandt werden, die zu einer Senkung der Entwurfskosten führen, obwohl der Einsatz der Verifikation zunächst einen zusätzlichen Entwurfsschritt darstellt. Trotzdem ist es erstrebenswert, Verifikation anzuwenden [Kro05].
Als Motivation für diese Arbeit galt es, eine Verifikation von einer IEC[1] 61499 [Lew01] Laufzeitumgebung durchzuführen, wobei ein unterlagertes Echtzeitbetriebssystem mitberücksichtigt wurde.

1.1 Aufgabenstellung

Die Aufgabe dieser Arbeit ist die Realisierung einer formalen Beschreibung einer Laufzeitumgebung[2] für den Standard IEC 61499. Als Laufzeitumgebung

[1] International Electrotechnical Commission
[2] engl.: Runtime

wird die μCrons Laufzeitumgebung verwendet, die im Laufe des μCrons Projekt [MYC] entwickelt wurde. Weiters soll noch ein unterlagertes Echtzeitbetriebssystem[3] miteinbezogen werden. Unter zu Hilfenahme von Net Condition Event Systems (NCES) sollen die einzelnen Elemente von der Laufzeitumgebung und dem Echtzeitbetriebssystem modelliert werden.
Das Ziel dieser Diplomarbeit ist ein Modell für die μCrons Laufzeitumgebung und ein Modell für das unterlagerte Echtzeitbetriebssystem zu erstellen. Dabei sollen die einzelnen Teile der Laufzeitumgebung und des Echtzeitbetriebssystems eingearbeitet und mit NCES modelliert werden. Die Modellierung wird mit einem Editor Namens TNCES (Timed Net Condition Event System) realisiert und mit dem Model-Checker ViVe (Visual Verifier) [Vyaa] im Anschluss auf dessen Funktionalität überprüft. Anhand dieser Modelle hat man die Möglichkeit eine Appliaktion die im Zusammenhang mit der μCrons Laufzeitumgebung und dem unterlagerten Echtzeitbetriebssystem liegt, auf ihre Korrektheit zu überprüfen. Mit der Verwendung dieser Modelle wird es möglich sein, eine beliebige Anwendung zu realisieren und anschließend auf dessen Verhalten zu verifizieren. Damit kann auf die Besonderheiten der Abarbeitungsmethodik der Laufzeitumgebung bei der Verifikation eingegangen werden.

1.2 Aufgabenbereiche

Die Arbeit wurde in die folgenden Teilaufgaben unterteilt und entsprechend abgearbeitet.

- Verständnis und Einarbeitung der Modellierung mit NCES
- Literaturrecherche der unterschiedlichen Modellierungsarten, Laufzeitumgebungen und Echtzeitbetriebssysteme
- Kennenlernen der Laufzeitumgebung und Erstellung des Modells
- Implementierung der Zeitabhängigkeiten der Laufzeitumgebung in das Modell
- Kennenlernen der Eigenschaften und Verhalten des Echtzeitbetriebssystem eCos
- Erstellung der einzelnen Modelle des Echtzeitbetriebssystems eCos

[3] engl.: RTOS (Real Time Operating System)

- Erstellung eines Applikationsmodells im Zusammenhang mit dem Modell der Laufzeitumgebung und Echtzeitbetriebssystem
- Testen des Applikationsmodells mittels formaler Beschreibung

1.3 Leitfaden

Dieser Abschnitt soll ein Überblick über die folgenden Kapitel geben. Im Kapitel 2 wird über den Stand der Technik berichtet. In diesem wird auf die Verifikation und speziell auf das Model-Checking eingegangen. Dabei werden die unterschiedlichen Modellierungsmethoden kurz vorgestellt und auf die Modellierungsmethode Net Condition Event Systems wird näher eingegangen. Weiters wird der Standard IEC 61499 mit dessen Eigenschaften und Laufzeitumgebungen angeführt. Ein Überblick über verschiedene Echtzeitbetriebssysteme wird zum Ende noch angegeben.
Im Kapitel 3 findet man eine nähere Beschreibung des Echtzeitbetriebssystems eCos und der μCrons Laufzeitumgebung. Hauptsächlich sind in diesem Kapitel deren Eigenschaften und Funktionsweise beschrieben. Die einzelnen Teile des Echtzeitbetriebssystems eCos und der μCrons Laufzeitumgebung werden im Detail angeführt und es wird auf die dazugehörigen formalen Modelle eingegangen.
Das Kapitel 4 beinhaltet den Anwendungsbereich der erstellten Modelle und deren Nutzen. Weiters werden die Programme vorgestellt, mit denen die NCES Modelle erstellt wurden und auf diese im Detail eingegangen. Abschließend wird eine Applikation auf dessen Verhalten verifiziert.
Kapitel 5 gibt eine Zusammenfassung der gesamten Arbeit wieder und Kapitel 6 vervollständigt die Arbeit mit einem Ausblick über weitere Forschungsarbeiten.

2 Aktueller Stand der Technik

In diesem Kapitel soll ein Überblick über die Verifikation in Systemen, das Model-Checking sowie über den Standard IEC 61499 gegeben werden. Auf das Model-Checking wird dabei im Detail eingegangen.

2.1 Verifikation

Die Aufgabe der Verifikation ist es, nach einer Spezifikationserstellung, Fehler zu erkennen. Beinhaltet jedoch die Spezifikation schon Fehler, so werden diese mit einer Verifikation nicht nachgewiesen. Bevor man nun näher in die Thematik der Verifikation eingeht, sollten zu allererst einige Grundbegriffe kurz erläutert werden. Man geht dabei vom Systembegriff aus.

Ein System besteht aus einer Menge von Elementen, die miteinander kommunizieren und sich auch gegenseitig beeinflussen können. Des Weiteren ist ein System von einer Umgebung zu einer Gesamtheit zusammengefasst. So kann ein System auch mit seiner Umgebung kommunizieren und das Systemverhalten kann durch Größen aus der Umgebung beeinflusst werden. Ein solches System wird ein offenes System genannt. Ein solches System ist in Abbildung 2.1 dargestellt. Hingegen spricht man von einem geschlossenen System, wenn die Umgebung nicht mit dem System kommuniziert. Um solche Systeme überhaupt darzustellen, verwendet man Modelle. [Bau96]

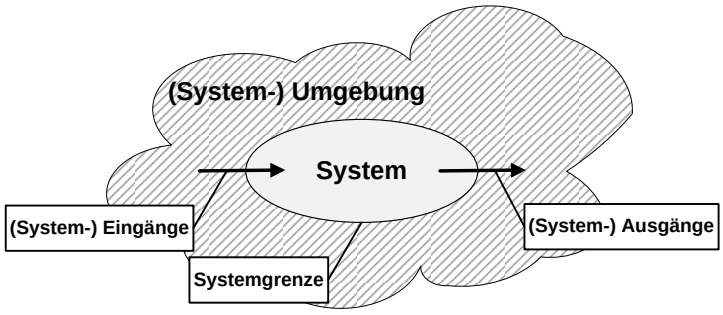


Abbildung 2.1: Allgemeiner Systembegriff

> *So erlaubt ein (System-) Modell das relevante Verhalten eines Systems formal nach einem bestimmten Schema darzustellen. Die nicht relevanten Eigenschaften werden jedoch vernachlässigt [Bey02].*

Wenn nun das Modell grafische Darstellungselemente enthält, wie es bei Petri- Netzen (siehe Kapitel 2.2.2) der Fall ist, dann kann es dadurch veranschaulicht werden. Wenn die grafische Darstellung eine eindeutige mathematisch formale Semantik besitzt, wie es ebenfalls bei Petri- Netzen der Fall ist, dann entspricht sie einer formalen Beschreibung.

> *So beschreibt eine formale Beschreibung für ein System die erlaubten, nicht erlaubten und geforderten Eigenschaften, Verhaltensweisen und Beobachtungen [Bey02].*

Eine formale Beschreibung kann, sofern es der Formalismus erlaubt, simuliert, analysiert und transformiert werden. Durch eine Simulation lässt sich das Verhalten des Systems schrittweise nachvollziehen und testen. Mit Hilfe von Analysewerkzeugen lassen sich bestimmte allgemeine Eigenschaften (z. B. Deadlockfreiheit[1]) erkennen. Durch den Einsatz von Transformations- und Synthesewerkzeugen lässt sich z. B. ein ausführbarer Code für einen Mikroprozessor erzeugen.

Möchte man nun eine Software verifizieren, so bedient man sich unterschiedlicher Techniken. So kann man eine Verifikation mit einer formalen oder nichtformalen Technik ausführen. Unter der nicht-formalen Verifikation versteht man das dynamische und statische Testen. Auch hier wird versucht, Fehler zu finden, die während des Entwicklungsprozesses entstanden sind. Hingegen spricht man von einer formalen Verifikation, wenn die mathematische Logik als Basis verwendet wird. Es soll dabei die Korrektheit der Eigenschaften einer Spezifikation gegenüber einem Modell überprüft werden. So werden in den meisten Fällen bei einer Modellierung formale Modelle verwendet. Dabei werden die Modelle mittels mathematischer Formalismen definiert und deren Eigenschaften werden mathematisch berechnet. Hat man nun eine Spezifikation in formaler Form vorliegen, z. B. in Prädikatenlogik erster Ordnung, in Aussagenlogik oder in Form eines endlichen Automaten, so ergeben sich mehrere Methoden zur formalen Verifikation.

Die am häufigsten angewandten Methoden zur formalen Verifikation laut [CWea] sind Theorem Proving und Model-Checking. Beim Theorem Proving liegt eine präzise Spezifikation des Problems in Form einer bestimmten Logik vor, z.B. Aussagenlogik oder Prädikatenlogik. So können bei der Verifikation

[1] deut.: Verklemmungsfreiheit; deadlock: gegenseitiges warten von zwei Prozessen

das Vorhandensein bestimmter Eigenschaften, sowie das Nichtvorhandensein bestimmter Fehler nachgewiesen werden [Pel01]. Beim Model-Checking erwartet man als Eingabe ein Programm und eine Spezifikation in Form endlicher Automaten. Somit müssen das Modell des Systems, als auch die Spezifikation in ihrer definierten Sprache übereinstimmen, d. h. dass ein Systemmodell A die Spezifikation B erfüllen muss. Beide Methoden weisen Vor- und Nachteile auf. Ein Vorteil des Theorem Proving ist, dass Fehler durch Überprüfung jedes Beweisschrittes auf dessen Richtigkeit vermieden werden können. Ein Nachteil ergibt sich aus dem Kosten/Nutzen- Verhältnis, da sich der Einsatz erst bei der Entwicklung sicherheitskritischer Anwendungen lohnt. Ein weiterer Nachteil beim Theorem Proving ist jener, dass durch den Benutzer Fehler einfließen können und somit die Beweise nur eine eingeschränkte oder überhaupt keine Aussagekraft besitzen. Der Vorteil von Model-Checking ist jener, dass dieser zu einem großen Teil automatisch abläuft. Das Erstellen der Modelle selbst bleibt jedoch dem Benutzer überlassen. Ein Nachteil ist die Komplexität der zu einstellenden Parameter im Model-Checker selbst. Weiters unterstützten Model-Checker in der Regel eigene Modellierungssprachen. So ist es oft nicht möglich, Modelle in diesen Sprachen automatisch generieren zu lassen.

2.2 Model-Checking

Wie schon im Kapitel 2.1 erläutert, ist das Model-Checking eine formale Technik zur Verifikation von Software. Erste Beschreibungen der Technik des Model-Checking wurden durch E. M. Clarke und E. A. Emerson in [CE81] und J. P. Quielle und J. Sifakis in [QS82] durchgeführt. Die Definition von „Model-Checking“ lautet nach [CS01]:

> *„Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems“.*

2.2.1 Allgemein

Mit dem Model-Checking hat man die Möglichkeit die Modellierung eines Systems daraufhin zu überprüfen, ob die Eigenschaften aus einer bestimmten Spezifikation erfüllt sind. Als sehr interessant erweist sich die Verwendung von Model-Checking bei sicherheitskritischen reaktiven Systemen. So lassen sich mit dem Model-Checking Fehler ausschließen, die zu einem sicherheitskritischen Zustand führen könnten. Weiters kann man Model-Checking für die Verifikation von nebenläufigen Systemen einsetzen. Dabei hat man die Mög-

lichkeit Deadlocks und Livelocks[2] zu finden. Heutzutage werden vorwiegend zwei verschiedene Ansätze für das Model-Checking angewandt. Zum einen gibt es das symbolische Model-Checking, wo der Zustandsübergangsgraph als eine Formel in einem Binary Decision Diagramm beschrieben wird, und zum anderen das on-the-fly Model-Checking. Beim on-the-fly Model-Checking wird der komplette Zustandsgraph mit allen Übergängen aufgebaut und jeder Zustand untersucht. Für dessen Überprüfung müssen alle relevanten Zustände eines Systems gespeichert werden.

Für die Untersuchung eines Systems mit Hilfe von formalen Verifikationstechniken, wie auch beim Model-Checking, wird diese in den Phasen *Modellierung des Systems*, *Spezifikation* und *Verifikation* eingeteilt.

Phase 1: Modellierung des Systems

In der Phase 1 muss das zu modellierende System in diejenige Form gebracht werden, wie der vom Model-Checking verwendete Algorithmus es fordert. Geht man von einem reaktiven System aus, so steht der Kontrollfluss in diesem System im Vordergrund. Man spricht dann von einem reaktiven System, wenn das System mit seiner Umgebung ständig interagiert und in der Regel nie terminiert. Weiters sind beim Kontrollfluss noch boolsche Variablen beteiligt, die einen Zustand repräsentieren. Wird nun der Wert einer Variablen geändert, so folgt darauf ein Zustandswechsel. Somit entstehen in reaktiven Systemen Transitionsfolgen, die mittels der Kripke- Struktur (siehe Abbildung 2.2) formal gut beschrieben werden können [CGP99].

Geht man davon aus, dass die Kripke-Struktur einen gerichteten Graphen darstellt, dann werden Zustände in diesem mit Eigenschaften markiert.

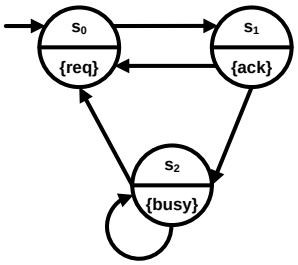


Abbildung 2.2: Kripke-Struktur

[2]Ist eine Art Deadlock, wobei Prozesse nicht in einem Zustand verharren sondern sie wechseln ständig zwischen mehreren Zuständen

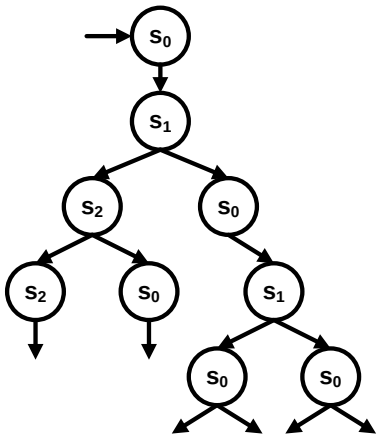


Abbildung 2.3: Berechnungsbaum

Das Beispiel in Abbildung 2.2 dient zur Veranschaulichung einer solchen Kripke-Struktur. Diese besitzt drei Zustände s_0, s_1 und s_2, welche mit einer Menge von Aussagen ausgestattet sind. Es handelt sich um eine Teilmenge von {req, ack, busy}. Mit Pfeilen wird die Transitionsrelation zwischen den Zuständen abgebildet. Das Strukturbeispiel stellt hier einen einfachen Kontrollmechanismus dar, der Zugriffe auf eine Resource gewährleistet. So wird nach einem Request (req) der Zugriff auf Zustand s_1 (ack) erlaubt. Von diesem Zustand gelangt man entweder wieder zurück nach s_0 oder es wird der Zustand s_2 aufgerufen. Dabei kann in Zustand s_2 die Aussage *busy* beliebig lang gültig sein. Um nun eine Berechnung auf einer Kripke-Struktur durchzuführen, werden Pfade angewandt, wobei ein Pfad eine unendliche Folge von Zuständen ist. Abbildung 2.3 verdeutlicht eine solche Menge von Pfaden, die einen Berechnungsbaum[3] bilden.

Phase 2: Spezifikation

Nach einer erfolgreichen Modellierung des Systems, müssen die Systemeigenschaften angegeben werden und man befindet sich in der Phase 2 des Model-Checking Prozesses. Hier wird die Überprüfung einer Spezifikation über den Zustandsübergangsgraphen des Systems betrachtet. Als Basis für die Spezifikation wird eine temporale Logik- bzw. Endliche Automatenspezifikation verwendet. Diese Logik erweitert dabei die klassische Aussagenlogik um weitere Operatoren, mit denen man zeitliche Zusammenhänge spezifiziert. Die tempo-

[3] engl.: computation tree

rale Logik wurde im Jahre 1977 erstmals von Amir Pnueli vorgestellt. Die Spezifikation eines Modells kann in verschiedenen temporalen Logiken beschrieben werden. Die am häufigsten verwendeten Logiken sind CTL*, CTL und LTL. Auf Basis der Kripke- Struktur können nun die unterschiedlichen Logiken beschrieben werden [Kro97].

CTL*

Wie schon zuvor erwähnt, spricht man bei Computation Tree Logic (CTL) von einer Temporallogik, wobei sich die Formeln dieser Logik auf die Gültigkeit von Aussagen über temporale Strukturen beziehen. Dabei verwendet die CTL* Logik ein Zeitmodell von diskreten Zeitpunkten und verzweigender Zeit. Das bedeutet, dass von einem Zustand aus mehrere Zustände erreicht werden können. Somit wird als Konzept für diese Logik die Beschreibung der Eigenschaften von Systemzuständen entlang eines Berechnungsbaumes verwendet (siehe Abbildung 2.3). Möchte man nun eine CTL*- Formel definieren, muss man zwischen zwei Formeln unterscheiden: Zustandsformel und Pfadformel. Dabei wird definiert, dass Pfadformeln ein Teil von Zustandsformeln sind und die Menge der CTL*- Formeln als die Menge der Zustandsformeln dargestellt wird.

Ausgehend von der Synatx von CTL* ergeben sich zwei Kategorien von Operatoren: Pfadquantoren und temporale Operatoren. Die Pfadquantoren stellen dabei die erste Kategorie der temporalen Operatoren dar. Mit ihnen kann man die Gültigkeit der Eigenschaft an einem Knoten überprüfen, der von einer Berechnung ausgegangen ist. Man hat somit die Pfadquantoren A (Eigenschaften treffen für alle Berechnungen zu) und E (Eigenschaft trifft mindestens in einer Berechnung zu) zur Verfügung.

Die temporalen Operatoren beschreiben hingegen die Eigenschaften von Systemzuständen entlang einer Berechnung im Berechnungsbaum. Mit diesen Charakteren der Operatoren kann man nun das zukünftige Verhalten des Systems beschreiben. Die wichtigsten Operatoren sind dabei X (Eigenschaft tritt in nächster Zeit auf), F (Eigenschaft tritt in Zukunft auf), G (Eigenschaft tritt immer auf) und U (erste Eigenschaft trifft zu bis zweite Eigenschaft zutrifft).

CTL

Bei der CTL Logik handelt es sich um eine Untermenge von der CTL* Logik. Bei dieser Logik müssen Aussagen über Pfade von Berechnungsbäumen getroffen werden. Das Zeitmodell ist dabei wieder dasselbe wie das bei der CTL* Logik. Für die Syntax von CTL ergeben sich zwei Konsequenzen. Eine Konsequenz betrifft den temporalen Operatoren X, U, F und G, wobei diesen

Operatoren stets ein Pfadquantor vorausgehen muss. Die zweite Konsequenz ist, dass den Pfadquantoren A und E ein temporaler Operator folgen muss. Anhand dieser Konsequenzen existieren exakt acht Basisoperatoren (EX und AX, EU und AU, EF und AF, EG und EG) für CTL.

LTL

Die Linear Temporal Logik (LTL) kann als echte Teillogik von CTL* verstanden werden. Die LTL Logik ist so definiert, dass jeder Zustand genau einen Folgezustand hat. Da die Menge der Zustände endlich ist, gilt ferner, dass es in der Struktur genau einen Zyklus gibt (oder keine Zustände). Also gibt es zu einer solchen linearen Struktur genau einen unendlich langen Pfad, wobei für jeden Startzustand ein Element von einer endlichen Menge von Zuständen gibt. Für die Syntax von LTL ergibt sich die Konsequenz, dass für den Pfadquantor A eine beliebige Anzahl an temporalen Operatoren X, U, F und G folgen kann. Ein Beispiel für eine LTL Formel wäre: $AXXUFGX$.

Phase 3: Verifikation

Zum Schluß, in der Phase 3, erfolgt eine Verifikation, welche durch die Ausführung eines Model-Checking Algorithmus überprüft, ob das System mit den definierten Spezifikationen übereinstimmt. Anhand eines Markierungsalgorithmus werden für das gegebene Modell und eine Formel die Menge aller Zustände berechnet. Voraussetzung ist dabei, dass die gegebene Formel erfüllt wird. Weiters durchläuft der Algorithmus einen Syntaxbaum der gegebenen Formel, wobei er für jede Teilformel alle Zustände markiert, die die Formel vollständig erfüllt.

2.2.2 Übersicht der Modellierungsarten

In diesem Unterkapitel soll ein kurzer Überblick über die verschiedenen Modellierungsarten für Systeme gegeben werden. Es wird eine Einsicht in die Petri-Netze, Bedingung/Ereignis- Netze, Timed Automata und Net Condition Event Systems geben. Auf Net Condition Event Systems, die eine Grundlage der Diplomarbeit bilden, wird im Kapitel 2.2.3 genauer eingegangen.

Petri- Netze

Petri- Netze sind formale Konstrukte, die sehr gut zur Modellierung und Analyse von Systemen und Prozessen geeignet sind. Das Konzept dieser Petri-Netze basiert auf der Dissertation („Kommunikation mit Automaten“) von Carl Adam Petri im Jahre 1962, dem Erfinder der Petri- Netze. Besonders

gute Verwendung finden Petri- Netze in der Modellierung von diskreten Systemen, verteilten Systemen sowie auch in der Modellierung von Nebenläufigkeit und Parallelität. Petri- Netze werden graphisch mit fünf Bestandteilen dargestellt: Stellen, Transitionen, Eingangs- Funktionen, Ausgangs- Funktionen und Marken. Die Struktur eines Petri- Netzes besteht aus zwei verschiedenen Arten von Knoten, den Stellen und den Transitionen. Eine Stelle entspricht einer Zwischenablage für Daten bzw. Informationen und wird durch einen Kreis symbolisiert. Eine Transition hingegen beschreibt die Verarbeitung von Daten bzw. Informationen und wird durch ein Rechteck oder einen Balken symbolisiert. Ferner existieren Kanten, die jeweils nur von einer Knotensorte zur anderen führen dürfen.

Heutzutage gibt es schon eine Vielzahl an Petri- Netzklassen, wobei unter anderem die Coloured Petri- Netze (CPN) und die Steuerungstechnisch interpretierte Petri- Netze (SPIN) dazugehören [Abe90]. Die Colored Petri- Netze sind eine der bekanntesten Weiterentwicklungen, welche die genannten Erweiterungen (Colored) besitzen. Jede Stelle besitzt eine Typvorgabe (Color set), welche die Art der Marke festlegt. Weiters besitzt jede Marke einen „Wert“ (Color), der dem Typ des Zustandes entsprechen muß. Bei der Petri- Netzklasse SPIN werden den Netzelementen problemspezifische Bedeutungen unterlegt. So dienen Stellen in der Regel zur Darstellung von Systemkomponenten mit stationärem Verhalten und mit den Transitionen werden die Veränderungen und ihre auslösenden Ereignisse beschrieben.

Bedingungs/Ereignis- Netze

Bei den Bedingung/Ereignis- Netzen (B/E- Netze) ist man im Grunde interessiert, ob eine Bedingung zutrifft oder nicht zutrifft, wenn die Bedingung durch Ereignisse eintreten oder beendet werden. Bei den B/E- Netzen repräsentieren Stellen Bedingungen und Transitionen Ereignisse. Stellen können mit einer Marke versehen sein (Bedingung gilt) oder sie besitzen keine Markierung (Bedingung gilt nicht). Somit wird bei den Stellen generell eine Kapazität mit eins gefordert, denn eine Bedingung trifft entweder zu oder sie trifft nicht zu. Weiters gilt noch, dass Bedingungen und Ereignisse sich gegenseitig definieren. So gibt es keine zwei Bedingungen, die aus demselben Ereignis eintreten oder beendet werden können, noch können zwei Ereignisse dieselbe Bedingung herbeiführen oder beenden [Bau96].

Die Schaltregel für B/E- Systeme ist wie folgt definiert:

- Eine Transition t kann nur dann schalten, wenn alle Eingänge der Transition eine Marke besitzen und alle Ausgangsstellen leer sind.

- Wenn nun die Transition t schaltet, werden alle Marken von den Eingangsstellen entfernt und jeder Ausgangsstelle wird eine Marke hinzugefügt.

Laut Definition aus [Abe98] ist ein B- Signal eine abschnittsweise konstante Funktion, die die Zeit in eine Menge von Bedingungen abbildet. Hingegen sind E- Signale Zeitfunktionen. Sie nehmen dabei nur zu diskreten Zeitpunkten Werte aus einer Menge von Ereignissen an. Zur Veranschaulichung solcher B/E- Systeme dient Abbildung 2.4. Es ist deutlich zu erkennen, dass jedes der beiden Signalarten Ein- und Ausgänge besitzen kann. Man hat somit B- und E- Signalflusslinien. Weiters ist das Übertragungsverhalten über einige forma-

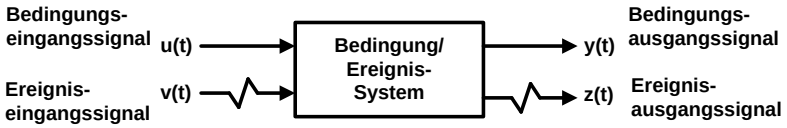


Abbildung 2.4: Blocksymbol für ein B/E-System [Abe98]

le Anforderungen definiert und die interne Beschreibung eines solchen B/E-Systems ist auf keiner Weise festgelegt. So kann im Grund jede Modellform verwendet werden, um die Internas von B/E- Modulen zu beschreiben.

Timed Automata

Timed Automata basiert auf dem Konzept der endlichen Automaten. Ein solcher endlicher Automat besitzt Zustände (Stellen), Übergänge (Transitionen) und üblicherweise Aktionen. Dabei dienen die Aktionen als Beschriftung für die Transitionen (Übergänge). Wird nun eine Erweiterung am endlichen Automaten mit einem Uhrenmodell durchgeführt, folgt daraus ein Timed Automata. Somit werden an den Transitionen zusätzlich so genannten Clock constraints angeführt. Es gibt laut [CGP99] zwei Arten von Clock constraints:

- Clock constraints an Transitionen nennt man *Guards*
- Clock constraints innerhalb von Zuständen nennt man *Invarianten*

Somit hat ein Clock constraint die Besonderheit, dass er nicht nur auf die eigentliche Aktion achten muss, sondern auch, ob die Invarianten eines aktuellen Zustandes und eines zukünftigen, sowie auch der Guard der Transition erfüllt sind. Sind nun all diese Begrenzungen erfüllt, kann ein Zustandsübergang erfolgen.

Bei den Transitionen in einem Timed Automata unterscheidet man zwei Arten von Übergängen:

- *Events* (Diskrete Übergänge)
- *Delay* (ΔT Übergänge)

Voraussetzung für das Schalten der beiden Transtionsarten ist, dass die Invarianten des aktuellen und des zukünftigen Zustandes erfüllt sein müssen. Der Übergang selbst verbraucht dabei keine Zeit. Bei einem Event Übergang erfolgt der Wechsel umgehend, sofern die Aktion erfüllt ist. Man kann somit sagen, dass die Zeit gleich bleibt und der Zustandsknoten geändert wird. Hingegen ist das Verhalten bei einem Delay Übergang genau umgekehrt. Hier wird Zeit verbraucht, der Zustand bleibt jedoch der gleiche. Deshalb werden alle Uhren um einen reellen positiven Wert erhöht. Die Timed Automata stellen ein gutes Werkzeug dar, um reaktive Systeme zu modellieren und zu analysieren.

2.2.3 Net Condition Event System

Bei diesem Konzept der Net Condition Event System (NCES) wurde von den Petri- Netzen ausgegangen, wobei diese in Modulen gekapselt werden. So kann man mit NCES eine Verbindung von mehreren Petri- Netze mit deren Ein- und Ausgangssignalen zu einem Gesamtmodell, mittels internen Modulen, erstellen. Auf dieses Konzept soll nun im folgenden Kapitel näher eingegangen werden.

Aus der Definition nach [SR02] ergeben sich für NCES zwei unterschiedliche Verbindungen zwischen Zustand und Trasitionen. Dabei ergibt sich ein Bedingungsbogen, ausgehend von einem Zustand zu einer Transition und ein Ereignisbogen von einer Transition zur anderen Transition (siehe Abbildung 2.5 und 2.6).

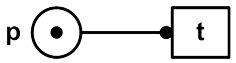


Abbildung 2.5: Bedingungsbogen (Condition Arc) [p, t] [SR02]

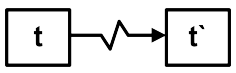


Abbildung 2.6: Ereignisbogen (Event Arc) [t, t'] [SR02]

Weiters wird für eine Menge von Transitionen t der Modus $M(t)$ definiert, der die Verarbeitung von einkommenden Ereignissignalen übernimmt. Ist nun $M(t) = ODER$, dann benötigt man zum Auslösen eines Übergangs t zumindest einen Ereignisbogen $[t_1, t]$, der ein Ereignissignal auslöst. Hingegen müssen für $M(t) = UND$ alle mit t über Ereignisbogen verbundene Übergänge ausgeführt werden (und damit Ereignisse an t senden) damit t initiiert wird.

Besitzt ein Übergang t keine eingehende Ereignisbögen, dann wird der Übergang als unabhängig (*independent*), andernfalls als erzwungen (*forced*) bezeichnet [Vyaa].

Initiierungsregeln der Übergänge

Wird bei NCES von einem Modul ausgegangen, das keine äußere Anschlüsse besitzt, dann wird das Modul für sich als ein Signal/Netz System (SNS) betrachtet. Wie sich nun ein solches SNS verhält, ist durch dessen interne Übergänge beschrieben. Dabei wird vor allem das Verhalten der gewöhnlichen Petri- Netze angewandt. So muss z.B. beim Übergang an einer Transition mit mehreren Vorgängerzuständen, mindestens jeder Zustand eine Markierung besitzen, damit dieser stattfindet. Zusätzlich sind noch die Übergangsregeln der Bedingungs- und Ereignissignale zu beachten. Da man von keinen einkommenden Ereignis- und Bedingungsanschlüsse ausgeht, wird in diesem Fall von unabhängigen Übergängen gesprochen. Diese unabhängige Übergänge werden dann weiters in spontane (*spontaneous*) und verpflichtende (*oblidged* oder *greedy*) Übergänge gegliedert [Vyab].

Modulare Behandlung

Das Besondere an NCES ist, dass sie in Modulen gekapselt und in Verbindung mit anderen NCES Modulen zusammengefügt werden können. Ein solches exemplarisches Modul eines NCES aus [HUB] zeigt Abbildung 2.7. Das Modul besteht aus einer Kombination von folgenden Ein- und Ausgängen:

1. Bedingungs- Ein/Ausgänge
2. Ereignis- Ein/Ausgänge

In Abbildung 2.7 ist nun ersichtlich, wie Bedingungseingänge und Ereigniseingänge intern im Modul auf Transitionen verbunden werden. Somit ist der Übergang einer Markierung auch abhängig von den einkommenden Bedingungs- und Ereignissignalen. Mit diesem Modellierungskonzept hat man nun die Möglichkeit, das Aktivieren/Deaktivieren von Transitionen sowie das Erzwingen

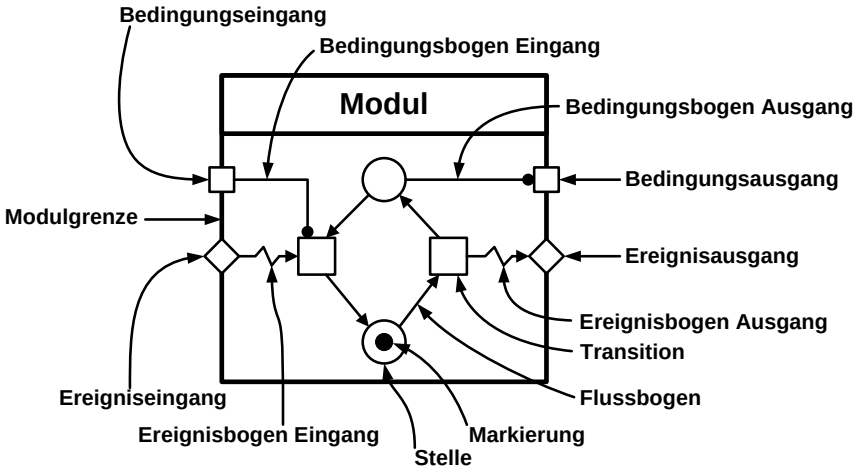


Abbildung 2.7: Ein Modul für NCES [HUB]

von Übergänge zu erreichen. Eine weitere Besonderheit dieser Module ist, dass man kleinere Module zu einem Netzwerk zusammenschalten kann. Dabei werden die Ausgänge eines Moduls mit den Eingängen des anderen Moduls mittels „Composition“ Bögen (composition condition arc und composition event arc) verbunden. Werden zwei Module zusammengeführt, so werden sie als Composition bezeichnet und es ergibt sich daraus ein Modul, das intern die beiden Module mit Bedingungs- und Ereignisbögen verbindet.

Zeitbehaftete NCES Netze (TNCES - Timed NCES)

Auf der Basis von NCES kann man als Erweiterung eine zeitliche Eigenschaft auf die Initiierungsregeln der Übergänge anwenden. So kann jedem Bogen vor einem Übergang ein Intervall $[l, h]$ zugewiesen werden. Dieses Intervall, auch als Permeabilitätsintervall bezeichnet, wird mittels den natürlichen Zahlen angegeben und wie folgt definiert: $0 < l < h < \infty$. Enthält ein Bogen kein Intervall, dann wird dem Bogen $[0, \infty]$ zugewiesen. Weiters besitzt jede Stelle p eine Uhr $u(p)$, welche erst dann läuft, wenn die Stelle p ein Markierung aufweist ($m(p) > 0$). Sobald die Stelle keine Markierung ($m(p) = 0$) mehr hat, wird die Uhr gestoppt und erst dann wieder neu gestartet, wenn sich die Anzahl der Marken in der Stelle ändert. Weiters laufen alle Uhren im gleichem Takt und es wird eine Zeit gemessen. Diese Messung der Zeit erfolgt nur, wenn sich der Markierungszustand der entsprechenden Stelle nicht ändert. Als Beispiel dient Abbildung 2.8, wo in der rechten Hälfte deutlich zu sehen ist, in

welcher Zeit eine Übergang aktiv wird. So wird z. B. die Transition t_4 erst nach 3 Zeiteinheiten aktiviert und die Markierung fließt von der Stelle p_6 nach p_7 [Vyaa].

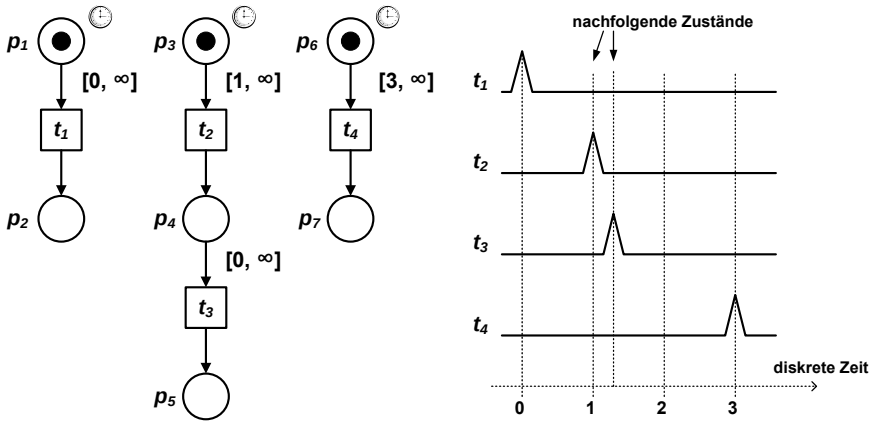


Abbildung 2.8: Timed S/E net [Vyaa]

Damit kann man nun einen Übergang t als zeitlich-aktiviert (*time-enabled*) bezeichnen, wenn die Uhr an der Vorgängerstelle $u(p)$ die Zeit anzeigt, wobei $l(p, t) < u(p) < h(p, t)$ gilt. Weiters bezeichnet man einen Zustand als tot (*dead*), wenn kein Übergang zeitlich-aktiviert ist und wenn kein Übergang nach einem Ablaufen der Uhr an der Vorgängerstelle aktiviert wird.
Initiierungsregeln in TNCES
Bei den Initiierungsregeln[4] in TNCES geht es darum die zeitlichen Intervalle zu definieren, wobei folgende definiert werden [Vyaa]:

- **Strong vs. Weak firing:** mit der *Strong* Regel müssen alle spontanen Übergänge in den Schritt einbezogen werden, wenn die Uhr der Vorgänger- Stelle den Wert l oder h erreicht hat. Wenn hingegen *Weak* gewählt wurde, dann muss mindestens einer der spontanen Übergänge in den Schritt einbezogen werden.

- **Earliest vs. Interval firing**: Im Falle einer *Interval* Initiierung wird der Übergang zeitlich- aktiviert, wenn die Uhr in der Vorgänger- Stelle einen Wert zwischen l und h besitzt. Hingegen bewirkt die *Earliest* Initiierung erst dann einen zeitlich-aktivierten Übergang, sobald die Uhr der Vorgänger- Stelle den Wert l erreicht hat.

[4] eng.: firing rules

- **Ultimo firing**: Diese Initiierung ist eine Kombination aus *Interval* und *Strong* firing. Dabei ist der Übergang während des Intervalls $[l, h]$ zeitlich aktiviert und muss spätestens zum Zeitpunkt h initiiert werden.

2.3 Standard IEC 61499

Seit Anfang der 80er Jahre wurde mit der Erarbeitung eines einheitlichen internationalen Standards für Speicherprogrammierbare Steuerungen (SPS) begonnen. Es handelt sich dabei um den Standard IEC 61131 [Lew98]. Inzwischen hat dieser Standard eine hohe Akzeptanz erreicht, wobei dieser als Hardwarestruktur eine zentrale Architektur verwendet. Mit dem neunen Standard IEC 61499 kann man hingegen komplexe Steuerungsprozesse in verteilten Automatisierungssystemen beschreiben.

2.3.1 Allgemein

Der Grundgedanke des Standards basiert auf der Verwendung von Funktionsblöcken. Das besondere an diesen Funktionsblöcken ist, dass deren Abarbeitung über Ereignisse angestoßen wird. Mittels Zusammenschalten von Funktionsblöcken erhält man ein Funktionsblocknetz, das nun verteilt auf einzelne Geräte einer Steuerung aufgeteilt werden kann. So bietet der Standard IEC 61499 einen Einsatz von SPS über Smart Devices (intelligente Peripheriegeräte) bis hin zu Feldbusprotokollen. Weiters ist der Standard in den folgenden 4 Abschnitten aufgeteilt:

- Teil 1: Architektur [IEC05a]
- Teil 2: Anforderungen für Software- Tools [IEC04a]
- Teil 3: Anwendungsrichtlinien [IEC04b]
- Teil 4: Regeln für die Einhaltung der Norm [IEC05b]

Für die Realisierung eines verteilten Steuerungssystems wird auf die Systemarchitektur in Abbildung 2.9 verwiesen. Dabei wird die Architektur auf drei Modelle aufgeteilt: Systemmodell[5], Gerätemodell[6] und Ressourcenmodell[7].

[5] engl.: System Model
[6] engl.: Device Model
[7] engl.: Resource Model

Das **Systemmodell** befindet sich auf der höchsten Ebene im Standard IEC 61499 und es kann aus mehreren Geräten bestehen, die über ein Kommunikationsnetzwerk miteinander verbunden sind. Auf diesen Geräten können nun Anwendungen[8] darauf laufen, die sich nach dem Standard IEC 61499 verteilt auf mehreren Geräten befinden können. Eine Anwendung auf einem oder mehreren Geräten besteht dabei aus einem Funktionsblocknetzwerk. Über ein Kommunikationsservice wird sichergestellt, dass die Verbindungen in den Anwendungen zwischen Ereignis- und Dateninterface miteinander korrekt verbunden werden.

Wird nun das **Gerätemodell** betrachtet, so kann ein Gerät aus mehreren Ressourcen bestehen. Eine Ressource ermöglicht eine unabhängige Steuerung und Ausführung von Funktionsblöcken. Mit dem Kommunikations- und Prozessinterface hat man die Möglichkeit eine Verbindung mit den physischen Prozessen über Ein- und Ausgänge herzustellen.

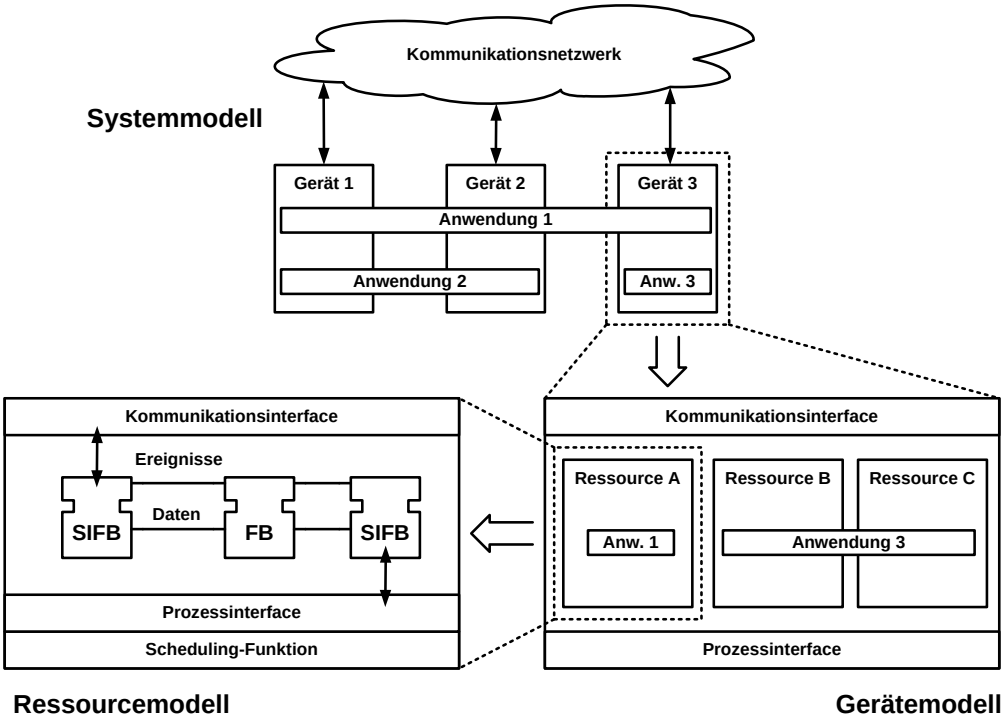


Abbildung 2.9: IEC 61499 Systemstruktur [FB04]

[8] engl.: Application

Im dritten Modell, dem **Ressourcemodell**, soll die Ressource eine Infrastruktur und Service zur Verfügung stellen, wobei diese zum Ablauf der Funktionsblockfragmente verwendet werden. Darüber hinaus existiert auch eine Verbindung zur Schedulingfunktion. Die Scheduling- Funktionen werden verwendet, um eine Reihenfolge bei der Abarbeitung der Algorithmen in den jeweiligen Funktionsblöcken zu beeinflussen [FB04].

Mit diesen drei Modellen sollte nun ein Überblick vom Standard IEC 61499 geschaffen werden. Ein weiteres wichtiges Modell, das im Standard angeführt wird, ist das **Funktionsblockmodell**. In diesem werden die Eigenschaften und der Aufbau eines Funktionsblockes beschrieben. Die Abbildung 2.10 zeigt einen solchen Funktionsblock als IEC 61499 *Basic Funktionsblock*. Ein Funktionsblock besteht aus einem Ereignisfluss und Datenfluss, mit jeweils mehreren Ein- und Ausgängen. Es werden drei Arten von Funktionsblöcken definiert: *Basic-FB*, *Composite-FB* und *Spezielle Kommunikations- und Schnittstellen FBs (SIFB)*.

Basic FB

Dieser Funktionsblocktyp wird durch einen internen Algorithmus definiert, der mittels der Ausführungssteuerung (siehe Abbildung 2.10) abgearbeitet wird. Diese Ausführungssteuerung wird durch einen Zustandsgraphen (ECC- Execution Control Chart) realisiert, wobei intern die Übergänge des ECCs durch Ereigniseingänge und interne („interface“) Variablen gesteuert werden. Der ECC hat also die Aufgabe Ereignisse und Daten mit den verschiedenen Algorithmen zu verbinden und dementsprechend abzuarbeiten. Mit den Datenein- und ausgängen werden die Algorithmen und internen Variablen in Verbindung gebracht [FB04].

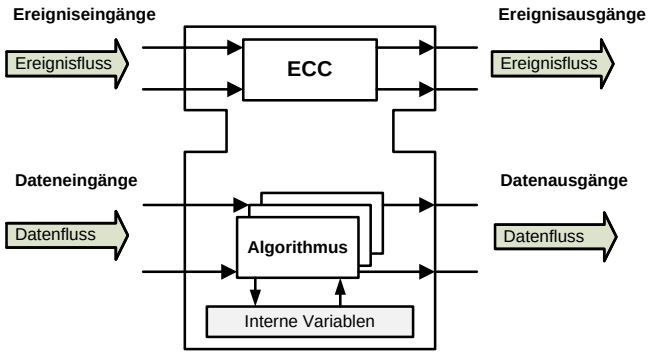


Abbildung 2.10: IEC 61499 Basic Funktionsblock

Die eigentliche Abarbeitung eines Basic FB erfolgt nach folgender zeitlicher Reihenfolge:

1. Eingangsdaten liegen an Dateneingängen an
2. Eingangsereignisse treffen ein und Abarbeitung kann beginnen
3. der Algorithmus kann mit Dateneingängen und internen Variablen die Datenausgänge bereitstellen
4. nach Abarbeitung wird Ausgangsereignis generiert

Composite FB

Bei einem Composite FB hat man als interne Realisierung ein FB- Netzwerk. Dabei werden die internen FBs als Component FBs bezeichnet. Die äußere Struktur (Interface) dieses FBs basiert auf jener des Basic FBs. Weiters kann das Component FB- Netzwerk mit dem Interface des Composite FBs verbunden werden. Tritt nun ein Eingangsereignis an einem Composite FB auf, dann wird dieses mit den intern verbundenen Component FBs weitergeleitet. Bei erfolgter Ereignisabarbeitung wird bei bestehender Verbindung ein Ausgangsereignis eines Component FBs an die äußere Struktur des Composite FBs generiert. Für die Datenverbindung gilt dasselbe.

Service Interface FB

Mit dem Service Interface Funktionsblock (SIFB) kann man unter Zuhilfenahme vom Kommunikations- bzw. Prozessinterface Daten und Ereignisse austauschen (service primitives). Das Interface des SIFB ist dabei analog zu den Basic FBs definiert. Für eine bessere Beschreibung des Interfaces eines SIFBs beschreibt der Standard ein zeitliches Sequenzdiagramm. Das in Abbildung 2.11 dargestellte zeitliche Sequenzdiagramm zeigt einige Beispiele für die Typen a) Requester und b) Responder. Die Zeit wird im Sequenzdiagramm nach unten hin aufgetragen und es wird der Ereignisfluss am Interface des SIFBs dargestellt. Mit diesem zeitlichen Sequenzdiagramm beschreibt der Standard die Bedeutung der einzelnen Ein- und Ausgänge der beiden Typen.

2.3.2 Laufzeitumgebungen für IEC 61499

Da ein Funktionsblocknetzwerk aus einer Anzahl von FBs bestehen kann, die untereinander über Ereignis- und Datenflüsse verbunden sind, ist dessen Abarbeitung von Interesse. Betrachtet man ein Funktionsblocknetzwerk in einem

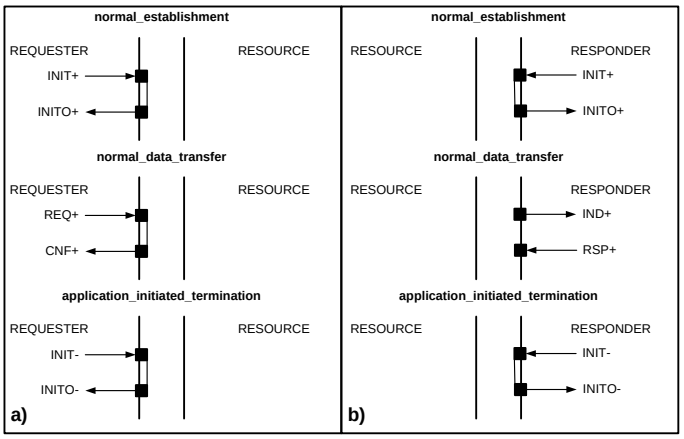


Abbildung 2.11: Sequenzdiagramm für a) Requester und b) Responder

eingebetteten System, dann ist dessen Ausführung mittels einer Laufzeitumgebung gegeben. Es werden nun im folgenden verschiedene IEC 61499 Laufzeitumgebungen angeführt. Dabei soll gezeigt werden, dass jede Laufzeitumgebung ein unterschiedliches Verhalten aufweist und das Weiterleiten von Ereignissen in Funktionsblocknetzwerken verschieden gehandhabt wird.

Die Function Block Runtime (FBRT)

Die von Dr. James H. Christensen entwickelte Laufzeitumgebung FBRT für IEC 61499, die im Zusammenhang mit der Programmierumgebung Function Block Development Kit (FBDK)[9] entstanden ist, ist in Java implementiert und dient zur Ausführung von Funktionsblocknetzwerken. In der FBRT gibt es eine Anzahl an Threads, die für die Ereignisgenerierung und deren Weiterleitung in Funktionsblocknetzwerken verantwortlich sind. Dabei arbeiten alle Threads auf demselben Prioritätslevel und können sich gegenseitig nicht unterbrechen. Hingegen können Threads in verschiedenen Ressourcen unterschiedliche Prioritäten besitzen. Weiters werden alle Prozesse die in einem kritischen Bereich beschrieben sind und beim Auftreten eines Ereignisses am Ereigniseingang von einem Basic- oder Composite Funktionsblock im selben Thread ausgeführt. Betrachtet man ein Funktionsblocknetzwerk nach diesen Konsequenzen, dann erfolgt die Ereignisweiterleitung über eine Verkettung von Ereignisverbindungen (Daisy-Chain). Damit die Ausführung anderer Threads gewährleistet ist, muss der Algorithmus kurz implementiert werden.

[9] www.holobloc.com

Laufzeitumgebungen an der Universität Patras

Das Konzept dieser Laufzeitumgebung wurde an der Technischen Universität Patras entwickelt und basiert auf drei Ebenen. Dabei handelt es sich um die Ebenen Application Execution Layer (AE Layer), Industrial Process-Control Protocol Layer (IPCP Layer) und Mechanical Process Interface Layer (MPI Layer). Die zentrale Einheit dieser drei Ebenen ist der AE Layer. Dieser enthält unter anderem ein Event Connection Manager (ECM), ein Data Connection Manager (DCM) und Function Block Containers (FBC), die als Threads implementiert werden. Der ECM liefert dem FBC Ereignisse, die mit einer integrierten Event- Queue als Ringbuffer im FBC aufgenommen werden können. Mittels eines Dispatchers werden Ereignisse den FB- Instanzen weitergeleitet. Mit dieser Architektur der Abarbeitungsumgebung wurden verschiedene Implementierung realisiert, wobei unter anderem Real-Time Application Interface (RTAI) und Real-Time Specification for Java (RTSJ) vorkommen [DT05].

Die C++FBRT Laufzeitumgebung

Die C++FBRT Laufzeitumgebung wurde an der Technischen Universität Wien von Dipl.- Ing. Alois Zoitl [Zoi02] entwickelt und eignet sich für kleine, eingebettete Systeme ohne Betriebssystem. Zentrales Element dieser Runtime ist der Event Dispatcher. Wird nun aus einem Funktionsblock ein Ausgangsereignis generiert, so wird dieses Ereignis in dem Event Dispatcher gestellt. Diese Queue basiert dabei auf dem FIFO Prinzip und es werden Ereignisse, die als erste in der Queue hineingestellt wurden, als erste durch den Event Dispatcher herausgenommen. Das entnommene Ereignis wird dann dem entsprechenden Ereigniseingang des Funktionsblocks weitergeleitet.

Die μCrons Laufzeitumgebung

Die IEC 61499 konforme Laufzeitumgebung μCrons (auch μCrons Advanced Runtime Environmet (MARTE)) wurde im Zuge des μCrons Projekt [MYC] an der Technischen Universität Wien gemeinschaftlich mit Profactor und der FH Wels entwickelt. Die μCrons Laufzeitumgebung ist in C++ implementiert und hat die Eigenschaften der besonderen Rekonfigurationsfähigkeit und die Echtzeitfähigkeit in IEC 61499- Netzwerken. So wie bei der C++FBRT ist das zentrale Element der Event Dispatcher, der jedoch in der μCrons Laufzeitumgebung ein etwas anderes Verhalten aufweist. Der Unterschied ist jener, dass in der Event-Queue mehrere gleiche Ereignisse enthalten sein können. Weiters wird im Unterschied zur C++FBRT das Eingangsereignis, das zu einem Funktionsblock weitergeleitet wird, zuerst in die Event- Queue reingestellt und erst

dann für den jeweiligen Funktionsblock zur Verfügung gestellt.

Laufzeitumgebung FUBER

Die Laufzeitumgebung FUBER (FUnction Block Execution Runtime) ist in der Programmiersprache Java implementiert und ist eine freie Software[10]. In FUBER hat man die Möglichkeit, eine Anzahl an Algorithmusabarbeitungs Threads und Ereignissabarbeitungs Threads zu erzeugen. Beim Ereignisabarbeitungs Thread wird das einkommende Ereignis von der Funktionsblock-Instanz in einer FIFO- Queue gespeichert. Weiters gibt es noch eine Algorithmus Ababreitungs Queue (FIFO- Prinzip), welche vom Algorithmusabarbeitungs Thread versorgt wird. Wenn der Thread mit der Abarbeitung fertig ist, wird ein neues Ereignis aus der Queue entnommen, um den Algorithmus der Funktionsblock- Instanz abzuarbeiten [CLA06].

Laufzeitumgebung aus Tampere

Bei diesem Prinzip wird von einem scan-basierten Ausführungsmodell ausgegangen, der Programme in einer vordefinierten Sequenz abarbeitet. Die Programme werden dabei in einem Kreislauf (Scan-Cycle) periodisch oder kontinuierlich ausgeführt. Dieses Prinzip ist typisch für Programme, die im Standard IEC 61131 implementiert sind. Für eine IEC 61499 Anwendung kann der scan-basierte Ansatz auch verwendet werden, trotz dessen Ansteuerung mit Ereignissen. Dabei wird jeder Funktiosblock als Programm mit einem zusätzlichen Event-Handler angesehen. Der Code des Programms (Funktionsblock) wird abgespeichert und wird erst gescannt und ausgeführt, wenn das entsprechende Eingangsereignis vorhanden ist. Somit wird der Event-Handler zum Event-Scanner, wobei die eintreffenden Ereignisse periodisch gescannt werden [LGLT].

Laufzeitumgebung ISaGRAF

Mit der Software ISaGRAF 5.0 von der Firma ICS Triplex lassen sich Automatisierungssteuerungen sowohl für den Standard IEC 61131 als auch IEC 61499 realisieren und mit der implementierten Laufzeitumgebung abarbeiten. Für die Abarbeitung von Anwendungen in IEC 61499 wird das vorhandene Scan-Cycle Prinzip von IEC 61131 verwendet. So wird nach dessen Ausführung der Abarbeitungsvorgang erst wiederholt, wenn der nächste Zyklus vom Funktionsblock selbst auftritt [ISA].

[10] engl. open source

2.3.3 Verifikation mit IEC 61499

In den letzten vergangenen Jahren hat man mit der Verifikation von IEC 61499 Funktionsblöcken begonnen. Grund dafür war es, das Verhalten der Funktionsblöcke besser zu erkennen und auf dessen Fehlfunktionen zu überprüfen. Somit soll im folgenden Kapitel ein Überblick gegeben werden, auf welcher Art und in wie weit mit dem Standard IEC 61499 verifiziert wurde. Dabei wird auf die unterschiedlichen Verifikationen von Funktionsblöcken und Funktionsblöcke mit deren Laufzeitumgebungen eingegangen. Diese Trennung der Verifikation mit IEC 61499 wurde deshalb eingeführt, da es zur Zeit keine wirkliche Verifikation einer IEC 61499 Laufzeitumgebung gibt, sondern nur jene von IEC 61499 Funktionsblöcken.

Formale Beschreibung von Funktionsblöcken

Der erste Ansatz für eine formale Beschreibung von Funktionsblöcken in IEC 61499 wurde von Vyatkin und Hanisch durchgeführt [VH99]. Für dessen Beschreibung wurden Net Condition Event Systems verwendet. Wie schon im Kapitel 2.2.3 beschrieben, kann man mit NCES Module erstellen, die miteinander über Ereignis- und Bedingungsbögen verschalten werden können. Somit kann eine Ereignis in IEC 61499 direkt mit einem Ereignisbogen (NCES) dargestellt werden.

Ein weiterer Ansatz für eine Verifikation von IEC 61499 wurde von Wurmus und Wagner [WW00] durchgeführt. Für die formale Beschreibung wurden steuerungstechnisch interpretierte Petri-Netze (SIPN) (siehe Kapitel 2.2.2) verwendet. Hierbei wird ein Ereignis durch den Markenfluss dargestellt. Später implementierten Hagge, N. und Wagner, B. ein formales Modell mittels CNet(Component Net)[HW]. Es handelt sich dabei um eine eigene Modellierungssprache. Mit CNet hat man die Möglichkeit, eine formale Verifikation zu definieren und es erleichtert, auf Basis einer Java Implementierung, eine Transformation von IEC 61499 Modelle.

Ein weiterer Ansatz für eine Verifikation von IEC 61499 führte laut [SFL02] Schankenbourg ein. Zur Modellierung eines Funktionsblockes verwendet Schankenbourg eine synchrone Sprache Namens SIGNAL. Um eine Synchronisation zwischen dem ECC und den Ereigniseingängen zu gewähren, verwendet SIGNAL interne Uhren. Auch hier ist noch kein Modell für die Weiterleitung eines Ereignisses implementiert worden.

Zhang, W.*et al.* hat einen Ansatz vorgeschlagen, wobei die formale Beschreibung von IEC 61499 Anwendungen mittels finite Zustands- Maschinen

(FSM)[11] verwendet werden kann [ZDH04]. Der erste Ansatz von dieser Arbeit war es, einen IEC 61499 Fuktionsblock als finites Zustands- Modell zu realisieren und eine formale Beschreibung unter Verwendung eines FSM basierten Tools durchzuführen.

Formale Beschreibung von FBs und deren Ausführungsumgebung

Das Modellieren von der Ausführungssemantik von Funktionsblöcken in IEC 61499 wird nach Vyatkin [Vya06] unter Verwendung von NCES durchgeführt. Diese Erweiterung von [VH99] konzentriert sich dabei auf die korrekte Weiterleitung der Ereignisse in einem Funktionsblocknetzwerk unter Verwendung von einem Scheduler. Trotzdem gibt es keine Laufzeitumgebung, die dieses Modell verwenden könnte.

Für ein simples Modell einer Laufzeitumgebung in IEC 61499 bietet Stanica [Sta05] hingegen einen Ansatz auf Basis von Timed Automata (siehe Kapitel 2.2.2). Es wird mit der formalen Beschreibung die Abarbeitung von den Algorithmen in einem einzigen Algorithmus eingeschränkt. Für die Weiterleitung von Ereignissen gibt es kein Modell.

Sowie Zhang, W.*et al.*, verwendet Khaligui *et al.* [DT05] den Ansatz von Zustands- Maschinen für die Verifikation von den Standard IEC 61499. Hierbei wird das Unterbrechungsverhalten beim gleichzeitigen Auftreten von Ereignissen mit einem offline design scheduling der Funktionsblöcke umgangen. Zur Verifikation der Korrektheit des Schedulings wird die Zustands- Maschine verwendet. Somit kann mit diesem Scheduler ein fest programmiertes Abarbeitungsmodell einer Laufzeitumgebung realisiert werden.

Eine formale Beschreibung über einer Laufzeitumgebung, namens FUBER, beschreibt Cenig *et al.* [CLA06]. Diese wurde mit interaktiven finiten Automaten in Supremica (Tool zur Entwicklung von effizienten robust control Systemen) entwickelt. Somit werden mit diesem formalen Modell mehrere Aspekte von der Laufzeitumgebung berücksichtigt. So beschreibt z. B. das Modell der Ereignisabarbeitung, dass jede Funktionsblock- Instanz erst dann abgearbeitet wird, wenn eine andere Instanz mit der Ereignisabarbeitung fertig ist. Weiters werden eingehende Ereignisse von einer Funktionsblock- Instanz in einer Queue gespeichert.

[11] engl.: Finite State Machines

2.4 Echtzeitbetriebssysteme

Ein Echtzeitbetriebssystem kann durch zwei wesentliche Eigenschaften charakterisiert werden: Pünktlichkeit und Determiniertheit. Die Pünktlichkeit fasst dabei die Funktionen der Rechtzeitigkeit und Gleichzeitigkeit miteinander zusammmen. So fordert die Echtzeit eine rechtzeitige Verarbeitung z. B. von Daten. Mit der Determiniertheit wird ermöglicht, dass das Systemverhalten vorhergesagt werden kann [FB04].

2.4.1 Klassifizierung

Weiters sind die Kriterien Echtzeitanforderung und Urheberrecht ein wichtiger Punkt für die Klassifizierung von Echtzeitbetriebssystemen. So wird bei der Echtzeitanforderung zwischen weicher und harter Echtzeit unterschieden. Bei einer weichen Echtzeit wird ein Überschreiten des Zeitlimits akzeptiert und es tritt kein Systemausfall ein. Jedoch steigen die Kosten infolge der Verzögerung leicht an. Kommt es dennoch zu einem Systemausfall beim Überschreiten des Zeitlimits, dann spricht man von harter Echtzeit. Hier steigen die Kosten rapide an und man muss harte Echtzeitanforderungen an das betrachtete System stellen. Als nächstes soll auf das Kriterium Urheberrecht eingegangen werden.

Hier unterscheidet man zwischen proprietärer und freier Software. Bei der proprietären Software hat der Eigentümer das volle Recht auf diese Software. Der Erwerb dieser Software ist möglich, dennoch können hohe Kosten auftreten. Dagegen ist die freie Software gebührenfrei. Jeder Nutzer hat einen freien Zugang zum Quellcode und die Möglichkeit den Quellcode für eigene Zwecke zu modifizieren.

2.4.2 Verhalten

Das Verhalten eines Echtzeitbetriebssystems wird sehr stark durch die Determiniertheit bestimmt. So ergeben sich mit dieser Eigenschaft für jeden möglichen aktuellen Systemzustand und Systemeingänge eindeutige Systemausgänge und Systemfolgezustände. Eine weitere Eigenschaft, die das Verhalten eines Echtzeitbetriebssystems charakterisiert, ist die Preemptivität. Mit dieser Funktionalität hat man die Möglichkeit, einen Prozess innerhalb eines Echtzeitbetriebssystems durch einen anderen Prozess zu jeder beliebigen Zeit zu unterbrechen, aber nur wenn er eine höhere Priorität besitzt. Eine nächste Betrachtungsweise ist die Verwaltung der Abarbeitungsreihenfolge der Prozesse. Diese Reihenfolge der Abarbeitung wird mit einem Scheduler gehandhabt, der

besagt, wann ein Prozess abgearbeitet werden soll. Im Weiteren unterscheidet man beim Scheduling zwischen einem statischen und dynamischen Verfahren. Beim statischen Schedulingverfahren wird die Abarbeitungsreihenfolge vor dem Systemstart getroffen, hingegen werden beim dynamischen Schedulingverfahren die Parameter während der Laufzeit geändert. Weiters können beide Schedulingverfahren jeweils in preemptives/non-preemptives Scheduling gegliedert werden. Wie schon erwähnt, können beim preemptiven Scheduling laufende Prozesse von höherprioren Prozessen unterbrochen werden. Hingegen werden beim non-preemptive Scheduling die, in Ausführung befindlichen, Prozesse nicht unterbrochen.

Möchte man einem Prozess eine Priorität zuweisen, dann wird das Verfahren des prioritätsbasierten Schedulingverfahren verwendet. Vertreter dieses Verfahrenes sind das Rate Monotic Scheduling (RMS), Erliest Deadline First (EDF), Least Slack-Time First(LST). Die Verfahren EDF und LST arbeiten mit variablen Prioritäten und jeder Prozess erhält eine Priorität mit einer entsprechenden Deadline[12]. Bei EDF hat ein Prozess mit einer kürzeren Deadline stets eine höhere Priorität als ein Prozess mit einer längeren Deadline. So unterbricht ein in Ausführung befindlicher Prozess mit höherer Priorität stets einen Prozess mit niedrigerer Priorität. Die Deadline beim LST Verfahren wird über die Differenz zwischen Deadline und (restlicher) Ausführungszeit (Slacktime) bestimmt. Je kürzer die Slacktime ist, desto höher ist die Priorität des Prozesses. (Ein laufbereiter Prozess mit höherer Priorität unterbricht stets einen Prozess mit niederer Priorität.) Bei beiden Verfahren handelt es sich um Scheduling mit dynamischen Prioritäten. Hingegen beschreibt das RMS ein Scheduling mit festen Prioritäten. Dabei erhält ein Prozess eine Priorität reziprok zu seiner Deadline. So unterbricht ein laufbereiter Prozess mit höherer Priorität stets einen Prozess mit niederer Priorität. Weiters wird mit so genannten Konditionsvariablen Mutex und Semaphore ein gleichzeitiges Zugreifen von mehreren Prozessen auf dieselbe Systemressource verhindert.

2.4.3 Echtzeitbetriebssyteme im Vergleich

In den folgenden Abschnitten soll ein Überblick über die gängigen Echtzeitbetriebssysteme angeführt werden. Dabei soll jedoch nur kurz auf die Eigenschaften der verschiedenen Echtzeitbetriebssysteme eingegangen werden. Mit dieser Auflistung soll deren unterschiedliches Verhalten angemerkt werden. Im speziellen wird dann in Kapitel 3.1 auf das Echtzeitbetriebssysteme eCos näher eingegangen.

[12] deut.: Frist

VxWorks

Das von der Firma Wind River entwickelte VxWorks ist eines der verbreitesten kommerziellen Echtzeitbetriebssysteme. VxWorks beinhaltet die Funktionalitäten der Preepmtivität und das Round-Robin- Scheduling. Als Besonderheit erweisst sich, dass die Applikationen innerhalb von VxWorks auf Tasks aufgeteilt werden. Diese Tasks können sich dabei den Adressraum teilen und können somit als Threads angesehen werden. Weiters unterstützt dieses Echtzeitbetriebssystem 256 Prioritätslevel und Konditionsvariablen (Mutex und Semaphore).

eCos

Dieses Betriebssystem ist ein freies Echtzeitbetriebssystem. Mit eCos hat der Anwender die Möglichkeit die Laufzeitumgebung für seine Zwecke zu konfigurieren. Das interessante an eCos ist, dass es mit sehr wenigen Ressourcen auskommt und für den Einsatz in eingebetteten Systemen[13] besonders gut geeignet ist. Weiters kann eCos erweitert und weiterentwickelt werden. eCos bietet bei seiner Anwendung zwei Scheduler an: Bitmap Scheduler und Multi Level Queue (MLQ) Scheduler. Beide können bis zu einem Prioritätslevel von 32 angewandt werden [Mas03].

QNX

Das Echtzeitbetriebssystem QNX wurde von der Firma QNX Software Systems entwickelt und findet Einsatz in eingebettete Systeme. QNX basiert auf einem Microkernel, in dem ein Scheduler und ein Messagepassingsystem implementiert ist. Weitere Betriebssystemkomponenten sind als Module zusätzlich einbindbar. Mit dem Messagepassingsystem kann eine Kommunikation zwischen den Modulen realisiert werden. Beim Scheduling unterstützt QNX folgende Verfahren: FIFO-, Round-Robin- und Adaptives Scheduling. Neu kommt hier das Adaptive Scheduling dazu, das ähnlich wie das Round-Robin-Scheduling funktioniert. Hier wird nur pro Abarbeitungsschritt die Prozesspriorität um maximal eins verringert. Wenn nun ein anderer Prozess blockiert, so wird den anderen Prozessen ihre Ausgangspriorität zurückgegeben [WEBa].

RTAI

Die Entwicklung von RTAI (Real Time Application Interface), das ein Linux basiertes Echtzeitbetriebssytem ist, begann im Jahre 1996 am Department of

[13] engl.: Embedded Systems

Aerospace Engineering am Politecnico di Milano. Es handelt sich dabei um ein kleines Echtzeitbetriebssystem, welches außerhalb des eigentlichen Linux-Kernels sitzt. In Zusammenhang mit dem Hardware Abstraction Layer basiert das Konzept von RTAI auf der Nutzung von Linux-Kernel Modulen und ist somit modular aufgebaut. So hat der Hardware-Abstraction-Layer (RTHAL) die primäre Aufgabe, Interrupts abzufangen und bei Bedarf weiterzuleiten. Weiters stehen RTAI drei Scheduler zur Verfügung: Uni-Prozessor (UP), Symetric Multiprocessing (SMP) und Multi-Uni-Prozessor (MUP) Scheduler. Der UP Scheduler unterstützt dabei zwei Schedulingverfahren: one-shot und periodic. Bei one-shot wird der Timer so programmiert, dass nach einer festgelegten Zeitspanne genau ein Interrupt ausgelöst wird, der den Scheduler aufruft. Bei periodic geschieht die Programmierung des Timers so, dass er in regelmäßigen Intervallen einen Interrupt auslöst, der ein Rescheduling veranlasst. Der SMP Scheduler ist für Multi-Prozessor Maschinen gedacht. Der Scheduler kann auch auf Systemen eingesetzt werden, die nur einen Prozessor haben, aber deren Kernel mit SMP- Option kompiliert wurde. Der MUP Scheduler ist ähnlich wie der SMP Scheduler. Der Unterschied ist hauptsächlich, dass es möglich ist, one-shot und periodic Scheduling simultan zu nutzen [WEBb].

μC/OS-II

Das μC/OS-II (MicroC/OS-II) wurde von Jean J. Labrosse entwickelt und wird speziell für Mikrocontrolleranwendungen verwendet. Es zeichnet sich durch ein prioritätsbasiertes preemptives Task- Scheduling, zahlreiche Intertask Kommunikations- und Synchronisationsmethoden und eine einfache Speicherverwaltung, sowie die Fähigkeit ROMfähige Applikationen zu realisieren, aus. Weiters kann der μC/OS-II Betriebssystemkern 64 Tasks verwalten, wobei acht Tasks für interne Aufgaben reserviert sind und die restlichen 56 Applikations-Tasks können aus parallelen Anwenderprozessen bestehen. In dieser laufen nun mehrere sequentielle Prozesse (Tasks) parallel zueinander ab. Es besteht somit ein Bedarf an Kommunikations- und Synchronisationsmechanismen. Der Betriebssystemkern μC/OS-II bietet folgende Verfahren an: Semaphore, Mutual Exclusion Semaphore, Message Mailboxes, Message Queues und Event Flags. Die Semaphores wurden zuvor schon angemerkt, hingegen sind die Mutual Exclusion Semaphore spezielle binäre Semaphore zur Klammerung von kritischen Abschnitten zwischen nebenläufigen Prozessen. Die Message Mailboxes und Message Queues erlauben die Kommunikation von Nachrichten zwischen Tasks. Während eine Mailbox maximal eine Nachricht aufnehmen kann, verwaltet eine Queue im Prinzip eine Warteschlange von Nachrichten. Event Flags können als ein Feld von binären Semaphoren verstanden werden [WEBc].

2.5 Zusammenfassung

Im Kapitel 2.1 wurde zu allererst ein Überblick über die Verifikation gezeigt. Um auf eine solche Verifikation zu gelangen wurde auf die Methode des Model-Checkings näher eingegangen. Dabei wurden zuerst die drei Phasen des Model-Checkings näher gebracht und anschließend verschiedene Modellierungsarten aufgelistet. Bei den Modellierungsarten wurde in Kapitel 2.2.3 speziell im Detail auf Net Condition Event Systems eingegangen, die im weiteren Verlauf die Grundlage der Modelle in Kapitel 4 sind. Im Kapitel 2.3 wurde auf den Standard IEC 61499 eingegangen. Als wesentlich in diesem Abschnitt haben sich die verschiedenen Implemtierungen von Laufzeitumgebungen in IEC 61499 gezeigt. Weiters hat die Verifikation mit dem Standard IEC 61499 näher gebracht, dass noch ein Bedarf für dieses gibt. Anschließend wurde noch im Abschnitt 2.4 über die Echtzeitbetriebssysteme berichtet. Anhand der unterschiedlichen Echtzeitbetriebssysteme hat man einen Überblick in die möglichen Varianten erhalten.

3 Verhaltensmodellierung von Echtzeitbetriebssystem und IEC 61499 Laufzeitumgebung

Mit dem vorigen Kapitel vom aktuellen Stand der Technik wurde ein Einblick in die Thematik gegeben, wobei gezeigt wurde, dass eine formale Beschreibung einer IEC 61499 Laufzeitumgebung noch wenig erforscht wurde. Mit diesen Erkenntnissen soll nun näher auf die IEC 61499 μCrons Laufzeitumgebung und das Echtzeitbetriebssystem eCos eingegangen werden.

Es werden zunächst die wichtigsten Elemente des Echtzeitbetriebssytems eCos erklärt und mit repräsentativen formalen Modellen dargestellt. Des Weiteren wird die μCrons Laufzeitumgebung im Detail beschrieben, wobei auch hier die einzelnen äquivalenten Modelle aufgelistet und erläutert werden.

3.1 Das Echtzeitbetriebssystem eCos

Dieses Kapitel gibt einen Einblick in das Echtzeitbetriebssystem eCos, das im Zusammenhang mit der μCrons Laufzeitumgebung modelliert werden soll. Im folgenden sollen die Eigenschaften und das Verhalten dieses Echtzeitbetriebssystems aus [Mas03] näher gebracht werden. Anschließend werden die formalen Modelle von eCos angeführt und dessen Verhalten erklärt.

3.1.1 Allgemein

Wie schon im Kapitel 2.4.3 erwähnt, ist dieses Betriebssystem ein open-source Echtzeitbetriebssystem. Der Anwender hat die Möglichkeit, die Laufzeitumgebung für seine eigenen Zwecke zu konfigurieren. eCos wird für den Einsatz in eingebetteten Systemen verwendet, da es mit einem geringen Ausmaß an Ressourcen auskommen kann.

Die Architektur von eCos zeichnet sich durch seine Konfigurierbarkeit aus. So kann die Architektur von eCos (siehe Abbildung 3.1) aus mehreren voneinander abgrenzenden Komponenten bestehen, wobei Hardware Abstracti-

on Layer (HAL) und Kernel die zentralen Elemente bilden. Der HAL bildet die Abstraktion zwischen der Hardware und dem Betriebssystem. Sämtliche Betriebsystem- Funktionen setzen auf dieser Abstraktionsschicht auf. Somit können eCos selbst, sowie Applikationen einfach auf andere Hardware- Plattformen portiert werden.

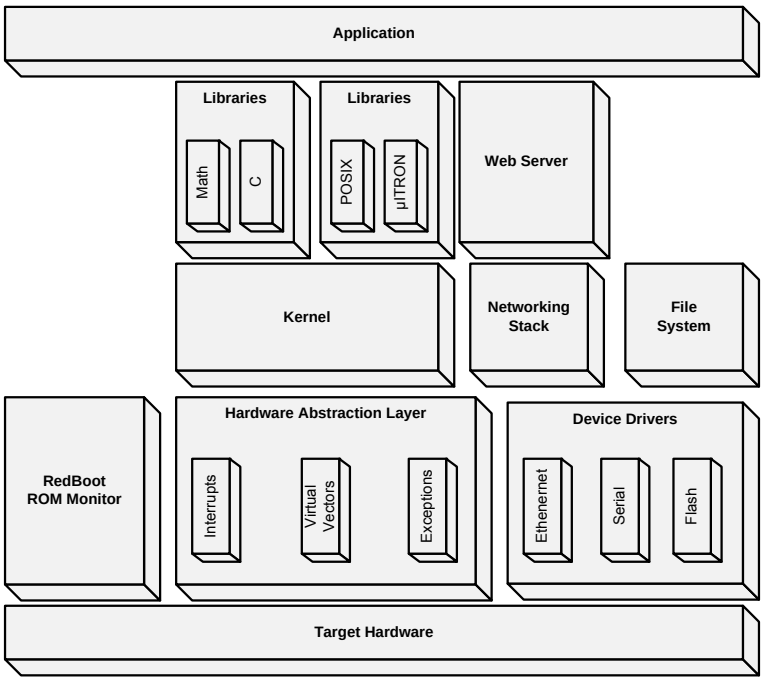


Abbildung 3.1: Beispiel einer eCos Architektur [Mas03]

Der Kernel ist das Herz des eCos Systems und bietet Standardfunktionen, die in einem Echtzeitbetriebssystem enthalten sind. Solche Standardfunktionen sind z. B. Interrupt und Exception Handling, Scheduling, Threads und Synchronisation. Alle Standardfunktionen, die im Kernel enthalten sind, sind für eigene Spezifikationen vollkommen konfigurierbar. Eine weitere Komponente, die in einer eCos Architektur enthalten sein kann, sind Libraries. In diesen können unterschiedliche Pakete von Bibliotheken vorgefunden werden (z. B. *Math* Bibliothek enthält mathematische Funktionen), die im Weiteren verwendet werden können. Mit einem Device Driver wird über einen Code eine spezifische Hardwarekomponente kontrolliert. eCos bietet noch die Komponente File System an, das drei unterschiedliche File System Implementierungen enthalten kann: ROM, RAM und JFFS2. Für das Senden und Empfangen von Daten über Internet oder Netzwerk ist der Networking Stack verantwortlich. Er

verwendet dabei mehrere unterschiedliche Protokolle für die Kommunikation. Möchte man Web Protokolle verwenden wie z. B. HyperText Transfer Protocol (HTTP), HyperText Markup Language (HMTL) und Java für das remote Device Managment, dann wird ein eingebetteter Web Server eingesetzt. Der RedBoot ROM Monitor ist für eingebettete Systeme ausgelegt und ermöglicht ein debugging[1] und bootstrap[2] Umgebung. Mit den Komponenten Application und Target Hardware wird eine Verbindung zu einer Anwendung und einer Zielhardware realisiert. Eine der wichtigsten Eigenschaften eines Echtzeitbetriebssystems sind die zur Verfügung stehenden Scheduler und die Synchronisations-Mechanismen.

3.1.2 eCos Scheduler

Die Aufgabe eines Schedulers ist es, einen entsprechenden Thread zur Abarbeitung auszuwählen. Weiters bietet er Mechanismen für die Synchronisation des ausführbaren Threads an und kontrolliert die einkommenden Interrupts, die ein ausführbarer Thread erhalten könnte. eCos stellt zwei Scheduler zur Verfügung. Es handelt sich dabei um den Bitmap und den Multi Level Queue (MLQ) Scheduler.

Bitmap Scheduler

Der Bitmap Scheduler ist ein schneller und deterministischer Scheduler. Mit diesem Scheduler hat man die Möglichkeit, 32 (0-31) Prioritäten zu verwalten, wobei in jedem Prioritätslevel nur ein Thread zugelassen ist. Somit ergibt sich eine maximale Anzahl von 32 Threads in der Scheduling Queue. Die Prioritätsnummer 0 entspricht der höchsten Priorität, 31 ist die niederste Priorität. Weiters unterstützt der Bitmap Scheduler die Funktionalität der Preemptivität, wobei ein niederpriorer Thread durch einen höherprioren Thread unterbrochen werden kann. Zur Veranschaulichung des Verhaltens des Bitmap Schedulers dient das Beispiel in Abbildung 3.2. Der Bitmap Scheduler enthält in diesem Beispiel drei Threads in seiner Queue, mit jeweils unterschiedlichen Prioritäten, wobei **Thread A** die Priorität 0 (höchste Priorität), **Thread B** Priorität 1 und **Thread C** Priorität 30 besitzt.

Bei der Ababreitung der Threads wird bei diesem Beispiel in Abbildung 3.2 nach einer Zeit **Thread C bereit und abarbeiten** mit dem **Thread C** be-

[1] Ein Debugger (von engl. bug) ist ein Werkzeug zum Auffinden, Diagnostizieren und Beheben von Fehlern in Hardware und Software

[2] In der Informatik bezieht sich der Begriff auf jeden Prozess, der aus einem einfachen System ein komplizierteres System aktiviert.

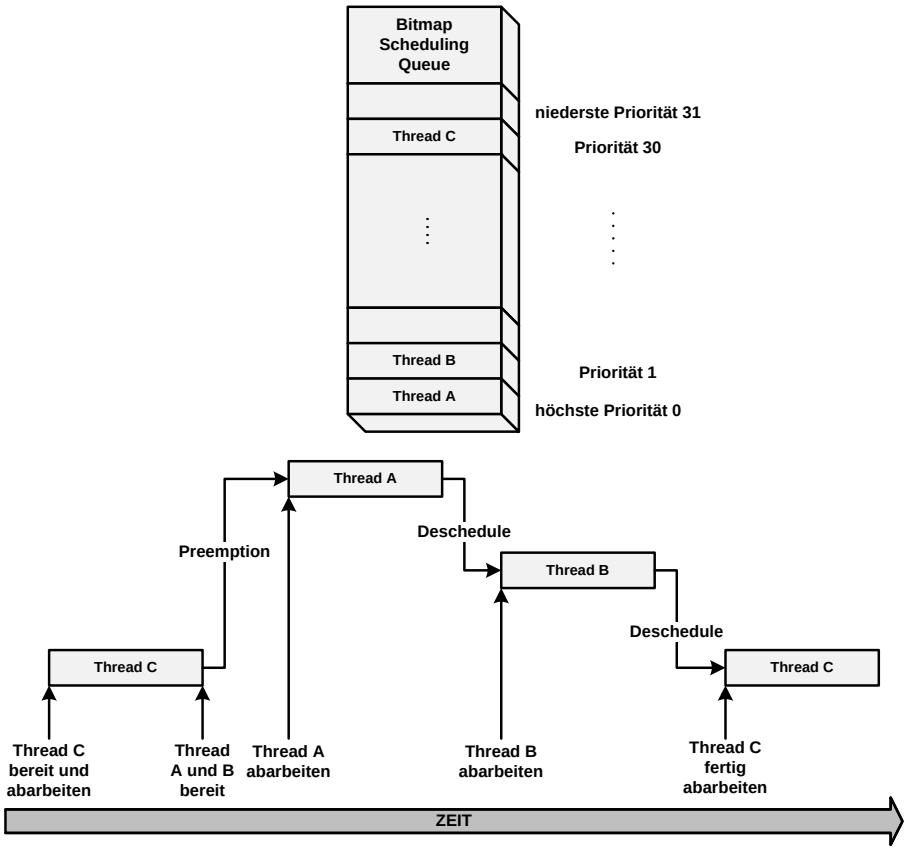


Abbildung 3.2: Bitmap Scheduler [Mas03]

gonnen. Die Abarbeitung von **Thread C** erfolgt bis zum Zeitpunkt **Thread A und B bereit**. Hier sind **Thread A** und **Thread B** bereit für die Abarbeitung und unterbrechen die Abarbeitung von **Thread C** wegen der Preemptivität. **Thread A** wird zur Zeit **Thread A abarbeiten** als nächster ausgeführt, da er die höhere Priorität als **Thread B** besitzt. Nach der Abarbeitung von **Thread A** wird dieser **descheduled** (aus dem Adressraum des Schedulers genommen) und es erfolgt erst zum Zeitpunkt **Thread B abarbeiten** die Ausführung von **Thread B**. Erst nach dessen Bearbeitung und anschließendem **descheduling** wird der **Thread C** fertig abgearbeitet.

Multi Level Queue (MLQ) Scheduler

Im Gegensatz zum Bitmap Scheduler erlaubt der MLQ Scheduler beliebig viele Threads pro Prioritätslevel. So wie beim Bitmap Scheduler ist auch hier die Preemptivität erlaubt. Die Anzahl der Prioritätslevel beträgt auch hier

32 und die Prioritätsnummer 0 entspricht der höchsten Priorität, 31 der niedersten Priorität. Befinden sich nun mehrere Threads auf derselben Priorität, dann wird die Abarbeitungsreihenfolge nach dem FIFO Prinzip angewandt. Des Weiteren bietet der MLQ Scheduler das Timeslicing (auch Round-Robin-Scheduling genannt) an, mit dem die Rechenzeit für den Thread begrenzt wird. Dem Thread wird also eine bestimmte Zeit für die Abarbeitung zugeteilt. Das Timeslicing wird verwendet, wenn mehrere Threads auf der gleichen Priorität sind. In Abbildung 3.3 ist ein Beispiel für einen MLQ Scheduler abgebildet. In diesem Szenario haben der **Thread A** und **Thread B** die selbe Priorität 0 (höchste Priorität). Hingegen besitzt **Thread C** die Priorität 30. Weiters wird bei diesem Beispiel das Timeslicing angewandt.

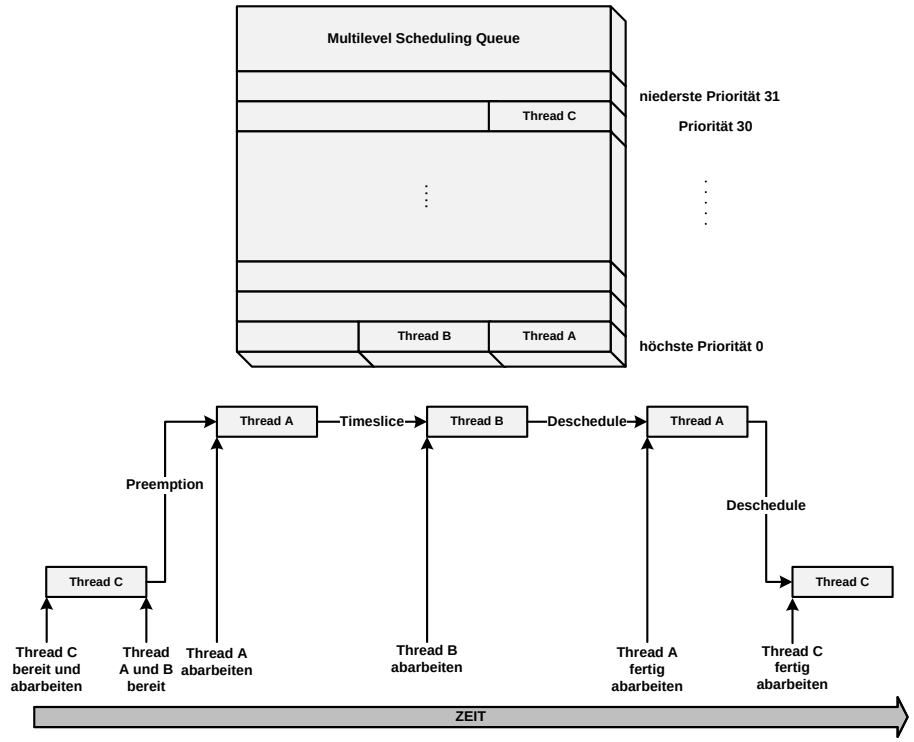


Abbildung 3.3: Multi Level Queue Scheduler [Mas03]

Bei der Ausführung der Threads wird zum Zeitpunkt **Thread C bereit** mit dem **Thread C** begonnen. Dieser wird mit **Preemption** durch den **Thread A** und den **Thread B** unterbrochen, da diese eine höhere Priorität besitzen. Beide Threads sind somit zur Zeit **Thread A und B bereit** bereit für die Abarbeitung. Welcher Thread nun zuerst abgearbeitet wird, wird nach dem FIFO Prinzip entschieden. In diesem Beispiel wurde der **Thread A** als erster im

Prioritätsstapel hineingelegt und wird zum Zeitpunkt **Thread A abarbeiten** somit auch als erster ausgeführt. **Thread A** wird abgearbeitet und erhält eine bestimmte Zeit zum Abarbeiten. In diesem Beispiel in Abbildung 3.3 wird der **Thread A** vorzeitig durch die Timeslicing Periode unterbrochen und der **Thread B**, der als nächster im Stapel nach **Thread A** liegt und auf der selben Priorität ist, wird zur Zeit **Thread B abarbeiten** ausgeführt. **Thread B** beendet die Abarbeitung vor der Timeslicing Periode und wird anschließend **descheduled**. **Thread A** setzt die Abarbeitung fort und nach dessen Fertigstellung wird **Thread C** fertig abgearbeitet.

3.1.3 eCos Synchronisations- Mechanismen

Der Kernel von eCos unterstützt mehrere Mechanismen, damit Threads in einem System untereinander kommunizieren können, und es wird eine Zugriffssynchronisation zu unterschiedlichen Ressourcen gewährleistet. Die folgenden Mechanismen unterstützt eCos:

- Semaphores
- Mutexes
- Condition variables
- Flags
- Message boxes
- Spinlocks

Die Mechanismen Semaphores und Mutexes werden im Folgenden erklärt, da diese für die Arbeit im Rahmen der μCorns Laufzeitumgebung Verwendung finden. Auf die restlichen Mechanismen wird jedoch nicht eingegangen.

Semaphores

Nach [Tan03] verwendet ein Semaphore die zwei Operationen *down* und *up*, um den Zugriff eines Prozesses zu steuern. Der Semaphore besitzt ganzzahlige Werte, mit denen die Anzahl an Weckrufen für zukünftige Verwendungen definiert wird. So prüft z. B. die *down*- Operation eines Semaphores nach, ob der gespeicherte Wert im Semaphore größer ist als 0. Ist dies der Fall, wird im Semaphore der Wert um eins verringert (um z. B. einen gespeicherten Weckruf zu verbrauchen) und es wird die Operation fortgesetzt. Die Ausführung

einer solchen Operation wird auch als *atomare Aktion* bezeichnet. Wurde eine Operation eines Semaphores gestartet, dann kann kein anderer Prozes auf dem Semaphore zugreifen. Der Zugriff ist erst dann erlaubt, sobald die Operation beendet oder gesperrt ist. Mit dieser Eigenschaft können somit Synchronisationsprobleme gelöst und Race Conditions[3] vermieden werden. Mit der *up*-Operation wird der Wert im Semaphore um eins erhöht.

Wird ein Semaphore mit 1 initialisiert und von zwei oder mehr Prozessen verwendet, dann spricht man von einem binären Semaphore. Mit diesem wird erreicht, dass nur einem Prozess der Zugriff auf die kritische Region erlaubt ist. Wenn nun jeder Prozess ein *down* vor dem Zugriff der kritischen Region und ein *up* kurz nach dem Verlassen der kritischen Region ausführt, dann ist ein wechselseitiger Ausschluss gewährleistet.

Mutexes

Der Mutex (*Mutual Exclusion*[4]) wird nach [Tan03] manchmal als eine vereinfachte Version eines Semaphores verwendet. Die Mutexe dienen der Verwaltung des gegenseitigen Ausschlusses von z. B. gemeinsam verwendeten Ressourcen. Bei diesem Synchronisationsmechanismus handelt es sich um einer Variable, die die Zustände *gesperrt* und *nicht gesperrt*[5] annehmen kann. Sobald ein Prozess (oder Thread) einen Zugang in einer kritischen Region erhalten möchte, wird der Zustand *gesperrt* aufgerufen. Wenn nun der Mutex *nicht gesperrt* ist, hat der Prozess die Erlaubnis in der kritischen Region einzutreten.

Besitzt jedoch der Mutex den Zustand *gesperrt*, wird der aufrufende Prozess *gesperrt* und muss so lange warten, bis der Prozess in der kritischen Region fertig ist und den Zustand **nicht gesperrt** aufruft. Dieser Fall wird auch *Priority Inversion*[6] genannt. Der kritischen Region wird somit temporär eine höhere Priorität für die Abarbeitung des Prozesses zugewiesen.

3.1.4 Timer und Callback- Function

Das Echtzeitbetriebssystem eCos verfügt über den Element Timer. Im Zusammenhang mit einer Callback- Function, die für die μCrons Laufzeitumgebung speziell implementiert wurde, kann dieser verwendet werden um z. B. Zeit-Funktionsblöcke in einem Funktionsblocknetzwerk anzusteuern. Die Callback-

[3] Wenn zwei oder mehr Prozesse einen gemeinsamen Speicher lesen oder beschreiben und das Endergebnis davon abhängt, wer wann genau läuft.
[4] deut.: gegenseitiger Ausschluss
[5] engl.: locked and unlocked
[6] deut.: Prioritätsumkehrung

Function steht somit im engen Kontakt mit dem Betriebssystem. In der Callback-Function selbst kann ein Code enthalten sein, der ausgeführt wird, sobald der Timer des Bestriebssystem diese Funktion aufruft. Der Ausführungscode des Timers könnte somit z. B. in einer solchen Funktion enthalten sein. Am Beispiel des Zeit- Funktionsblockes E_CYCLE, trägt sich dieser bei seiner Aktivierung in eine Liste, die in der Callback- Function enthalten ist, ein. Sobald die Callback- Function über dem Timer aufgerufen wird, wird in der Liste nachgeschaut welcher Funktionsblock als nächster abgearbeitet werden soll. Weiters wird beim Aufruf der Callback- Function ein interner Zähler um einen Wert erhöht, der für das Weiterzählen der enthaltenen Zeit- Funktionsblöcke zuständig ist.

3.2 Die μCrons Laufzeitumgebung

In diesem Kapitel wird ein Überblick über die IEC 61499 μCrons Laufzeitumgebung gegeben. Es wird auf die Architektur und auf deren formale Modelle eingegangen.

3.2.1 Allgemein

Wie schon im Kapitel 2.3.2 kurz erläutert, ist die μCrons Laufzeitumgebung im Zuge des μCrons Projekt [MYC] entwickelt worden. Das Ziel war es, eine Laufzeitumgebung für eingebettete Systeme zu entwickeln, die besondere Rekonfigurationsfähigkeiten und Echtzeitfähigkeit aufweist. Die Laufzeitumgebung wurde auf Basis des Standards IEC 61499 entwickelt, um die Eigenschaften der Rekonfigurierbarkeit und Echtzeitfähigkeit zu gewährleisten. Mit der Rekonfigurierbarkeit steht ein erweitertes Set an Management Befehle zur Verfügung und mit der Echtzeitfähigkeit erhält man eine zeitliche Konfiguration von Abarbeitungsketten.

3.2.2 Eigenschaften

Als Ausgangspunkt für die Beschreibung der μCrons Laufzeitumgebung dient Abbildung 3.4. Die wichtigsten Elemente der Laufzeitumgebung sind Hardware Abstracion Layer (HAL), External Event Manager, eine Anzahl an Ressourcen und ein Event Chain Executor (auch Event Dispatcher[7]) in jedem Thread. Weiters zeigt die Abbildung 3.4 noch die Einheit Device specific Hardware und Operating System, das getrennt zur Einheit μCrons Laufzeitumgebung

[7] deut.: Ereignis Versender

abgebildet ist. Unter Verwendung des Hardware Abstraction Layer können der μCrons Laufzeitumgebung externe Ereignisse über Interrupt-Handling, Device I/O, Memory, Timer oder Network zugeführt werden. Zum Unterschied des HALs vom Echtzeitbetriebssystem eCos, bezieht sich der HAL der μCrons Laufzeitumgebung auf das Betriebssystem und Device. Er bietet somit eine einheitliche Schnittstelle für die verschiedenen externen Ereignisse. Wie schon zuvor erwähnt, gibt es eine Anzahl an Ressourcen, die untereinander unabhängig sind. Jede Resource beinhaltet jeweils einen Resource Event Chain Executor. Dieser Resource Event Chain Executor wird im Zusammenhang mit einem IEC 61499 Funktionsblocknetzwerk verwendet und wird als Background Thread bezeichnet. Eine Ressource kann aber auch mehrere Realtime Event Chain Executor beinhalten, die im Zusammenhang mit einem IEC 61499 Funktionsblocknetzwerk einen Realtime Thread darstellen. Diese beiden Threads (Backgound und Realtime) befinden sich in den Ressourcen der μCrons Laufzeitumgebung.

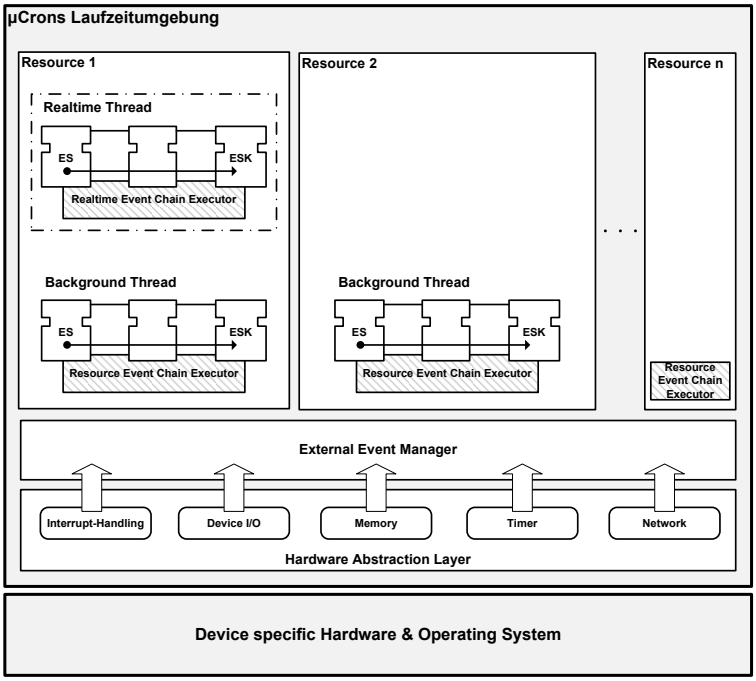


Abbildung 3.4: Übersicht der μCrons Laufzeitumgebung [MYC]

Weiters werden mit dem External Event Manager die unterschiedlichen Ereignisse, die vom Hardware Abstraction Layer generiert wurden, verarbeitet und anschließend der jeweiligen Ressource weitergeleitet, wobei der entspre-

chende Thread in der Ressource für dessen Abarbeitung angestoßen wird. Der Thread besteht aus einem IEC 61499 Funktionsblocknetzwerk und jeweils einem Event Chain Executor. Die Abarbeitung des Funktionsblocknetzwerkes erfolgt über IEC 61499 Ereignisse. Über Managment- Kommandos können Applikationen generiert und verändert werden. Näheres über die Weiterleitung der Ereignisse in Threads wird im Kapitel 3.2.3 beschrieben.

Ein weiteres wichtiges Element der μCrons Laufzeitumgebung ist der bereits erwähnte Event Chain Executor. Mit dem Event Chain Executor werden Ereignisse in einer Queue abgespeichert und es wird weiters noch die Art der Abarbeitung (Ereignisweiterleitung) der Ereignisse in der μCrons Laufzeitumgebung definiert. Betrachtet man weiters noch die Umgebung der μCrons Laufzeitumgebung, dann fehlt noch der Bereich von Device specific Hardware und Operating System. Mit diesem zusätzlichen Bereich werden die konkrete Hardware und das Betriebssystem dargestellt. Das Betriebssystem ist wesentlicher Bestandteil der μCrons Laufzeitumgebung. Mit dem Betriebssystem können Scheduler verwendet werden, mit denen man unterschiedlichen Threads der μCrons Laufzeitumgebung unterschiedliche Prioritäten vergeben kann, was im Zusammenhang mit der Echtzeitfähigkeit der μCrons Laufzeitumgebung besonders wichtig ist. Ein Betriebssystem das z. B. im Zusammenhang mit der μCrons Laufzeitumgebung verwendet werden kann ist das in Kapitel 3.1 angeführte Echtzeitbetriebssystem eCos.

Nun soll auf das interne Verhalten und den Aufbau eines Threads näher eingegangen werden. Dabei wird auf den Event Dispatcher (Event Chain Executor) und die Weiterleitung eines IEC 61499 Ereignisses in einem Thread eingegangen.

3.2.3 Weiterleitung von IEC 61499 Ereignisse

Wie schon im Kapitel zuvor erklärt, wird ein Thread mit einem Eingangsereignis vom External Event Manager für dessen Abarbeitung gestartet. Als Ausgangspunkt für die Erklärung der Weiterleitung von IEC 61499 Ereignisse dient die Abbildung 3.5. Zentrales Element in Abbildung 3.5 ist der **Event Dispatcher**, der eine intern **Event-Queue**[8] enthält. Der **Event Dispatcher** selbst legt alle auftretenden Eingangsereignisse nach der Reihe ihres Auftretens in der **Event-Queue** ab und holt die Ereignisse von der **Event-Queue** nach dem FIFO Prinzip (siehe Kapitel 2.3.2) heraus. Anfangs sollte man sich jedoch die **Event-Queue** in Abbildung 3.5 leer vorstellen. In der **Event-Queue** befinden sich somit keine Ereignisse. Sobald sich nun z. B. ein Ereignis **EI1** (**EI**

[8] deut.: Ereignisstapel

steht für Ereigniseingang) in der **Event-Queue** befindet, wird es dem Funktionsblock **FB1** weitergeleitet. Mit dem Ereignis **EI1** wird der Funktionblock **FB1** abgearbeitet und die **Event-Queue** ist wieder leer. Während der Abarbeitung von **FB1** werden in der **Event-Queue** die nötigen Ereignisse bei der Ereignisverzweigung nach **FB1** abgelegt. In diesem Beispiel handelt es sich um die Eingangsereignisse **EI2** und **EI3**. Nach erfolgreichem Einfügen von **EI2**, wird das Ereignis **EI3** in der **Event-Queue** eingefügt. Wenn der Funktionsblock **FB1** fertig ist, schaut der **Event Dispatcher** in der **Event-Queue** nach, ob Ereignisse herauszunehmen sind. In diesem Fall stehen zwei Einträge **EI2** und **EI3** in der **Event-Queue**. Somit werden nach dem FIFO Prinzip zuerst das Ereignis **EI2** aus der **Event-Queue** entnommen und dem Funktionsblock **FB2** als Ereigniseingang weitergeleitet. Das Ereignis **EI3** übernimmt den Platz von **EI2** in der **Event-Queue** ein, **FB2** wird ausgeführt und nach dessen Abarbeitung wird wieder mittels des **Event Dispatchers** in der **Event-Queue** nachgesehen, ob ein Ereignis vorhanden ist.

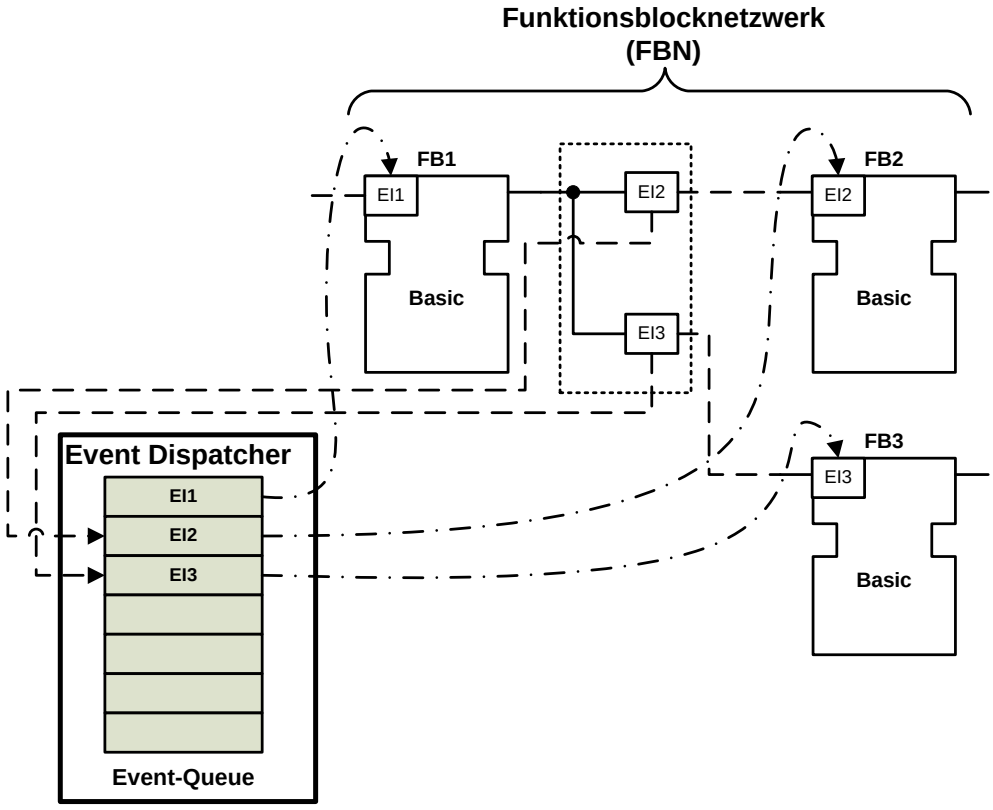


Abbildung 3.5: Ereignisabarbeitung in einem Thread mit FBN

In dem Fall von Abbildung 3.5 befindet sich das Ereignis **EI3** in der **Event-Queue** an oberster (erster) Stelle. Somit wird mit dem herausgenommenen Ereignis **EI3** der Funktionsblock **FB3** mit dem Ereignis **EI3** abgeabeitet. Die **Event-Queue** enthält in diesem Beispiel in Abbildung 3.5 keine Ereignisse mehr (**FB2** und **FB3** sind mit keinen FBs mehr verbunden) und die Abarbeitung des Threads ist somit beendet, bis durch Auftreten eines externen Ereignisses der Thread wieder angestoßen wird.

Abbildung 3.6 zeigt nun, wie eine **Ereignis Quelle**[9] ein IEC 61499 Ereignis generiert. Mit dem **External Event Manager** wird der Thread mit dem **Event Dispatcher** aktiviert. Die **Event-Queue** ist wiederum leer vorzustellen. Es wird über den **External Event Manager** zuerst eine Identifikationsnummer **ID** in der **Event-Queue** hineingestellt. Nach erfolgter „Anmeldung“ des **External Event Manager** in der **Event-Queue** mit **ID**, wird die Abarbeitung fortgesetzt. Das Ereignis **ID** wird mittels den **Event Dispatcher** aus der **Event-Queue** herausgeholt und es wird der dem externen Ereignis zugeordnete Funktionsblock, in diesem Beispiel in Abbildung 3.6 der **SUBSCRIBE-FB**, angestoßen und die **Event-Queue** ist wieder leer.

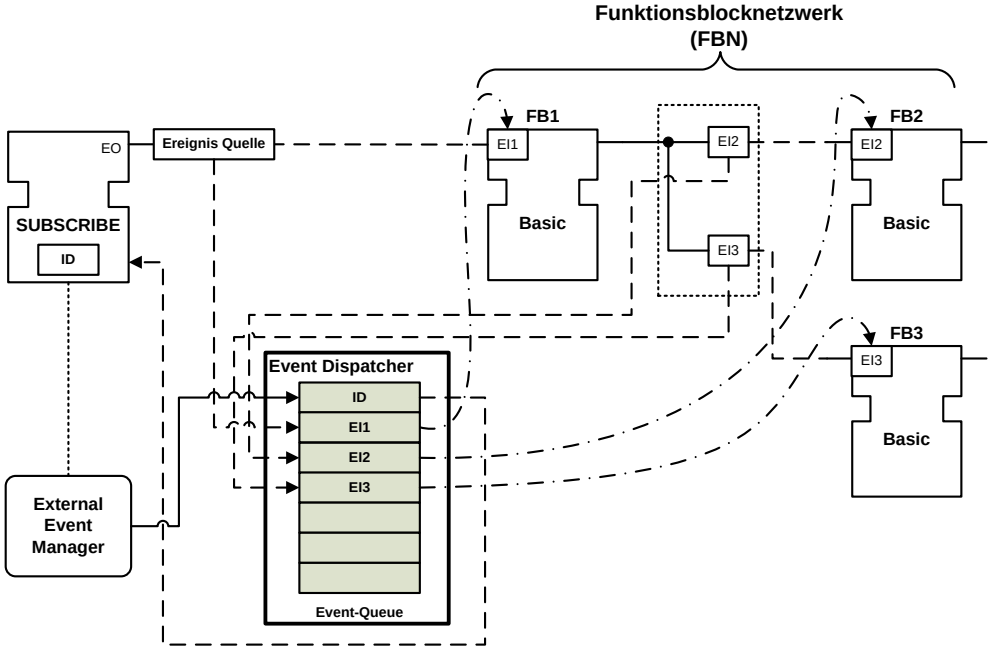


Abbildung 3.6: Generierung eines IEC 61499 Ereignisses in einem Thread durch eine Ereignis Quelle

[9] engl.: Event Source (ES)

Der **SUBSCRIBE** stößt mit einem Ereignis die **Ereignis Quelle** in Abbildung 3.6 an, die das erste IEC 61499 Ereignis (**EI1**) im **Event Dispatcher** ablegt. Die folgende Abarbeitung der restlichen IEC 61499 Ereignisse des **Funktionsblocknetzwerkes** ist die selbe, wie jene zuvor in Abbildung 3.5 beschrieben. Der Event Dispatcher der μCrons Laufzeitumgebung, wie bereits schon erwähnt, verfügt über eine Event-Queue. Diese Event-Queue ist ein Stapel mit x Stellen, wobei die Anzahl x der Stellen implentierungsabhängig ist. Ist nun die Event-Queue vollkommen mit Ereignissen besetzt, dann geht das zusätzlich eingefügte Ereignis verloren. Weiters können in dem Event Dispatcher der μCrons Laufzeitumgebung mehrere gleiche Ereignisse in der Event-Queue eingetragen werden.

3.3 Formale Modelle des Echtzeitbetriebssystems eCos

In diesem Kapitel werden die formalen Modelle der eCos Scheduler und der Synchronisations- Mechanismen, die in den Kapiteln 3.1.2 und 3.1.3 beschrieben wurden, angeführt. Bei den formalen Modellen der beiden Scheduler entspricht die Priorität 0 der höchsten Priorität, also so wie in Kapitel 3.1.2 erklärt. Als Beispiel für die Modelle wird jeweils das Prioritätslevel 1 gezeigt. Das wesentliche Verhalten der Modelle wird mittels eines Aktivitätsdiagramms beschrieben, da die interne Realisierung der Modelle in NCES zu groß und unübersichtlich erscheinen würde. Die kommenden Abbildungen stellen auf der linken Seite das äußere Interface eines Moduls in NCES dar und auf der rechten Seite gibt es das dazugehörige Aktivitätsdiagramm. Die Wahl der Darstellung der formalen Modelle als Aktivitätsdiagramme hat sich unter anderem auch deshalb als günstig erwiesen, da sich in der UML2[10] die Semantik der Aktivitätsdiagramme stark den Petri Netzen annähert und damit auch zum Teil an Net Condition Event Systems. Für das Verständnis der in den kommenden Abbildungen vorhandenen Aktivitätsdiagramme dienen die im Anhang A.1 angeführte Erklärung der Elemente für Aktivitätsdiagramme.

3.3.1 Modell des Bitmap Schedulers

Der Grundgedanke zur Realisierung eines Bitmap Schedulers war es, dass ein NCES Modul pro Priorität erstellt wurde das für die Steuerung des zugehörigen Threads verantwortlich ist. Im Zuge dieser Arbeit wurden Module bis zum

[10] engl.: Unified Modeling Language 2

fünften Prioritätslevel erstellt. Diese Module können nun miteinander verbunden werden und bilden z. B. einen Bitmap Scheduler mit fünf Prioritäten. Die Erweiterung der Prioritäten bis zur Priorität 31 ist gegeben, indem das Interface der Module zum Teil gleich bleibt und für die entsprechende Priorität erweitert werden kann. Ein solches Modul ist in Abbildung 3.7 ersichtlich. Die wesentlichen Merkmale des Moduls und der anderen Module des Bitmap Schedulers sind deren Aufbau der Ein- und Ausgangsereignisse/bedingungen. Jedes Modul besitzt das Eingangsereignis **activate** für die Aktivierung des Moduls (der Priorität). Dieses Eingangsereignis wird entweder von einem Modul höherer Priorität oder niederer Priorität angesteuert. Mit dem Eingangsereignis **th_sus** teilt der Thread dem Modul (der Priorität) mit, dass er mit der Abarbeitung fertig ist und mit **th_wactiv**, dass er bereit für die Abarbeitung ist. Mit dem Ausgangsereignis **next_prio** wird auf die nächste Priorität verwiesen und im Fall von Abbildung 3.7 wird mit **next_p0** auf die Priorität 0 (höhere Priorität) verzweigt. Diese Ausgangsereignisse (**next_prio** und **next_p0**) werden mit dem jeweiligen Eingangsereignis **activate** des höherprioren und niederprioren Moduls verbunden. Mit den Eingangsbedingungen **p0_sus**, **p0_wa**, **p2_sus** und **p2_wa** aus Abbildung 3.7 wird dem Modul der Status der Priorität 0 und Priorität 2 bekannt gegeben. Der Status der Priorität selbst in Abbildung 3.7 wird über die Ausgangsbedingungen **p1_sus** und **p1_wactiv** den nächsten Modulen weitergeleitet. Weiters besitzt jedes Modul die Ausgangsbedingungen **run_th** mit dem der Thread aktiviert wird und **stop_th** mit dem er deaktiviert wird. Mit der Tabelle 3.1 erhält man einen Überblick über die Verbindungen zu/von einem Thread und zu/von einer niederen/höheren Priorität.

Für die interne Realisierung des formalen Modells des Bitmap Schedulers wird das Prioritätslevel 1 verwendet, das in Abbildung 3.7b zu sehen ist. Das wesentliche Element des Bitmap Schedulers ist **Status von Thread**. Dieses gibt eine Information über den momentanen Zustand des Threads in Priorität 1 an, das durch die Ereigniseingänge **th_wactiv** und **th_sus** bestimmt wird. Diese Ereignisse generieren die Ausgangsbedingungen **p1_wactiv** oder **p1_sus**, mit denen der Zustand des Threads an die nachfolgenden Prioritäten übergeben werden kann.

Interface zu	**Bezeichnung**
Modul Thread	
enable	Aktivierung des Threads
NOTenable	Deaktivierung des Threads
Modulen niederer Priorität	
next_prio	Aktivierung der niederen Priorität 2
p1_sus	Priorität 1 ist deaktiviert
p1_wactiv	Priorität 1 ist aktiviert
Modulen höherer Priorität	
next_p0	Aktivierung der höheren Priorität 0
Interface von	**Bezeichnung**
Modul Thread	
th_sus	Thread mit Abarbeitung feritg
th_wactiv	Thread für Abarbeitung bereit
Modulen niederer Priorität	
p2_sus	Priorität 2 ist deaktiviert
p2_wa	Priorität 2 ist aktiviert
Modulen höherer Priorität	
p0_sus	Priorität 0 ist deaktiviert
p0_wa	Priorität 0 ist aktiviert
Modulen niederer/höherer Priorität	
activate	Aktivierung der Priorität 1

Tabelle 3.1: Interface für das Modul Scheduler als Priorität 1

Beginnend von **START** gelangt man nach einem Eingangsereignis **activate** von **Aktivierung** nach **Abfrage Priorität 0**. Hier wird die Abfrage anhand der Eingangsbedingungen **p0_sus** und **p0_wa** gemacht, ob auf die Priorität 0 verzweigt werden soll. Ist die Bedingung **p0_wa** wahr, dann wird mit **Priorität 0** das Ausgangsereignis **next_p0** generiert, das mit der Priorität 0 verbunden ist und man gelangt wieder zu **START** zurück. Diese Implementierung wurde deshalb gewählt, da bei einem Wechsel von einer Priorität zur anderen Priorität Zeit vergeht. Es könnte somit eine höhere Priorität wieder aktiv werden wollen und es müssen somit alle höheren Prioritäten geprüft werden. Mit **p0_sus** wird mit **Priorität 1** die Priorität 1 abgefragt, ob der Thread bereit für die Abarbeitung ist. Ist nun der Thread bereit für die Abarbeitung, wird mit **Thread bereit** auf **Thread ausführen** verzweigt. Der Thread wird abgearbeitet und durch das Eingangsereignis **th_sus** oder durch die Eingangsbedingung **p0_wa** gestoppt. Liegt am Eingangsereignis **th_sus** ein Ereignis an, dann beendet der Thread seine Abarbeitung und verzweigt mit **fertig** auf **Thread fertig**. Anschließend wird ein Ausgangsereignis **next_prio** generiert und auf die nächste Priorität verwiesen. Weiters wird noch auf **START** zurückgekehrt. Durch die Eingangsbedingung **p0_wa** wird dem Modul bekannt gegeben, dass die Priorität 0 bereit für die Abarbeitung ist. Es ist somit ein höherpriorer Thread bereit für die Abarbeitung. In diesem Fall wird über der Bedingung **Priorität 0** auf **Priorität 0 möchte drankommen** verzweigt. Der Thread wird über **Status von Thread** unmittelbar gestoppt, das Ausgangsereignis **next_p0** wird generiert, das auf Priorität 0 verbunden ist, und es wird auf **START** zurückgegangen. Betrachtet man andere Prioritätslevel, dann kann das Level z. B. auch von anderen höherprioren Modulen unterbrochen werden und es wird auf die entsprechenden Prioritäten verzweigt.

Möchte nun der Thread an der Priorität 1 nicht abgearbeitet werden, so ergibt sich nach **Abfrage Priorität 1** eine Verzweigung **Thread nicht bereit** und es wird über **nächste Priorität** das Ausgangsereignis **next_prio** generiert. Anschließend kehrt man wiederum auf **START** zurück.

Weiters ist noch anzumerken, dass in Abbildung 3.7b die Eingangsbedingungen **p2_sus** und **p2_wa** nicht angeführt wurden. Diese beiden Eingänge geben lediglich, wie schon am Anfang erwähnt, den Status der Bereitschaft zum Abarbeiten der Priorität 2 an Priorität 1 bekannt. Die Eingänge werden verwendet um eine Abarbeitungszeit (Kontextwechsel) im formalen Modell einzubauen, was jedoch im Moment nicht von Interesse ist.

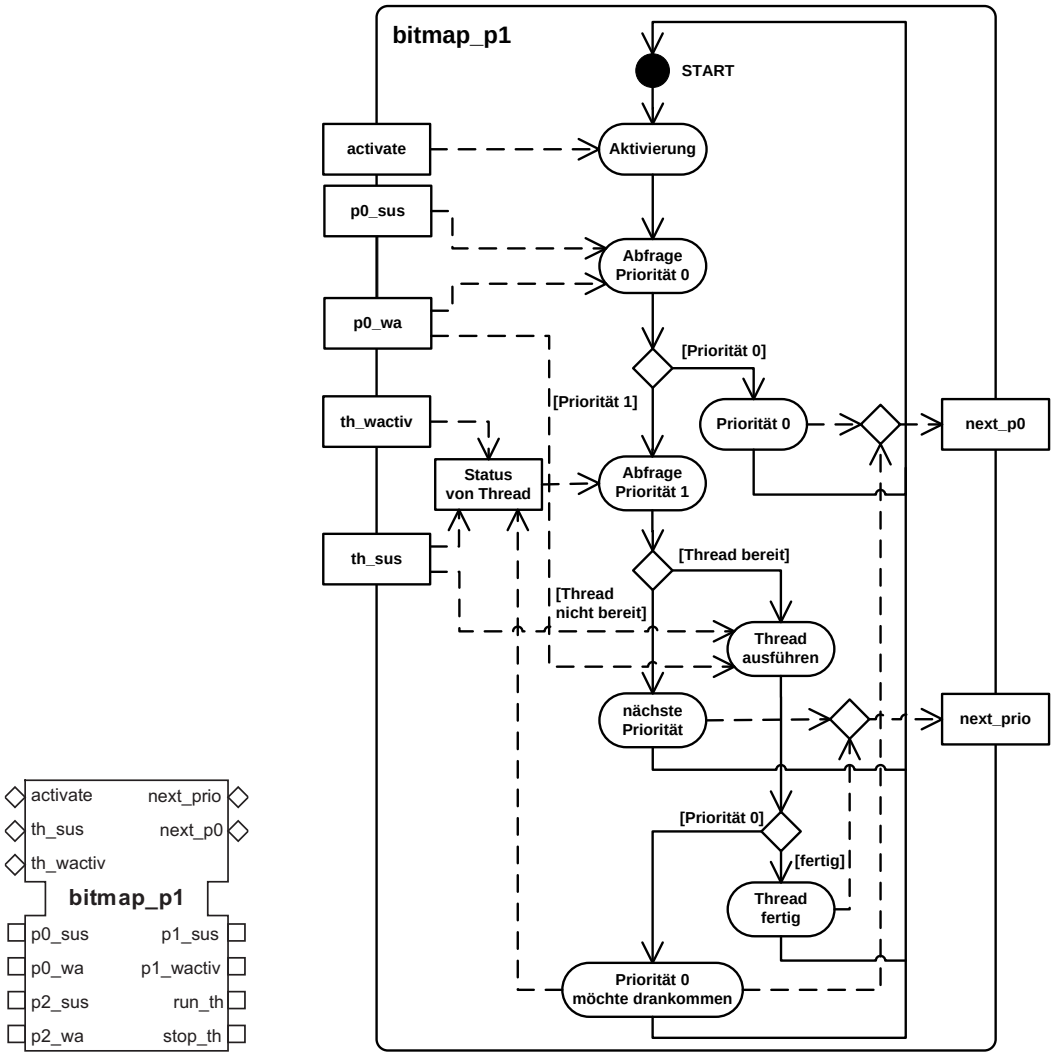


Abbildung 3.7: Bitmap Scheduler a) NCES Modul b) Verhalten

3.3.2 Modell des MLQ Schedulers

So wie beim Bitmap Scheduler wurde auch beim MLQ Scheduler pro Priorität ein NCES-Modul realisiert. Das Modul des MLQ Schedulers besitzt, wie das Modul des Bitmap Schedulers, ein Eingangsereignis **activate**, Ausgangsereignis **next_prio** und **next_p0** (im Falle von Abbilding 3.8a), die Eingangsbedingungen **p0_sus**, **p0_wa** und **p2_sus**, **p2_wa** der höheren Priorität und niederen Priorität sowie die Ausgangsbedingungen **p1_sus**, **p1_wactiv**.

Der einzige Unterschied zum Aufbau des Moduls gegenüber dem Bitmap Scheduler ist jener, dass mehrere Threads pro Priorität angeführt werden können. Somit besitzt das Modul des MLQ Schedulers pro Thread die Eingangsereignisse **th_sus** und **th_wactiv**. Das Modul in Abbilding 3.8a besitzt zwei Threads und hat somit die Eingangsereignisse **th1_sus**, **th1_wactiv** und **th2_sus**, **th2_wactiv**. Für die Ansteuerung der Threads, wie schon beim Bitmap Scheduler erläutert, dienen die Ausgangsbedingungen **run_th** und **stop_th**. Das Modul des MLQ Schedulers besitzt im Falle von Abbilding 3.8a zwei Threads, wobei dementsprechend jeweils zwei Bedingungsausgänge pro Thread für die Ansteuerung verwendet werden (**run_th1**, **stop_th1** und **run_th2**, **stop_th2**). Die Verbindungen zu/von einem Thread und zu/von einer niederen/höheren Priorität sind die selben wie in Tabelle 3.1 angeführt.

Für das interne Verhalten des Moduls des MLQ Schedulers in Abbildung 3.8a dient das Aktivitätsdiagramm in Abbildung 3.8b. Es handelt sich wieder um das Prioritätslevel 1, jedoch mit zwei Threads. Der erste Teil in Abbildung 3.8b ist ident zu Abbildung 3.7b und wird im Folgenden nicht erklärt. Der Unterschied zu Abbildung 3.7b ist ab der Aktion **Abfrage Priorität 1**, wo mit **Status Thread 1** oder **Status Thread 2** nachgeschaut wird, ob einer der beiden Threads abgearbeitet werden möchte. Sind beide Threads noch nicht bereit für die Abarbeitung, wird auf **nächste Priorität** verwiesen, das Ausgangsereignis **next_prio** generiert und auf **START** retourniert. Das Ausgangsereignis **next_prio** wird dann unmittelbar auf die nächste Priorität geleitet. Ist ein Thread hingegen bereit für die Ausführung, dann wird über **Abfrage Priorität 1** mit **Thread bereit** entweder der Thread 1 oder der Thread 2 ausgeführt. Welcher von den beiden nun abgearbeitet wird, hängt von dem jeweiligen Status der Threads ab.

Wird z.B. über **Thread 1 ausführen** der Thread 1 ausgeführt, dann wird dieser entweder von der Priorität 0 (höhere Priorität), vom Timeslicing oder von sich selbst unterbrochen. Tritt die Verzweigung **Priorität 0** auf, dann ist die Eingangsbedingung **p0_wa** aktiv und der Thread 1 wird mit **Unterbrechung TH1** unterbrochen. Anschließend wird auf die Priorität 0 mit **next_p0** verwiesen und auf **START** retourniert. Hingegen wird mit **fertig/Timeslice** die Aktion **nächste Auswahl** aktiviert. Bei dieser Aktion wird mit Hilfe von **Status Thread 2** überprüft, ob der Thread 2 abgearbeitet werden will oder nicht. Es wird entweder der nächste Thread abgearbeitet, oder es kommt wieder der Thread 1 dran oder es wird auf die nächste Priorität mit **next_prio** verzweigt. Das selbe Verhalten gilt auch bei der Auswahl von Thread 2 bei **Thread 2 ausführen**.

Weiters soll noch angemerkt werden, dass mit **Status von Thread 1** und **Status von Thread 2** der Status der Priorität 1 selbst bestimmt wird.

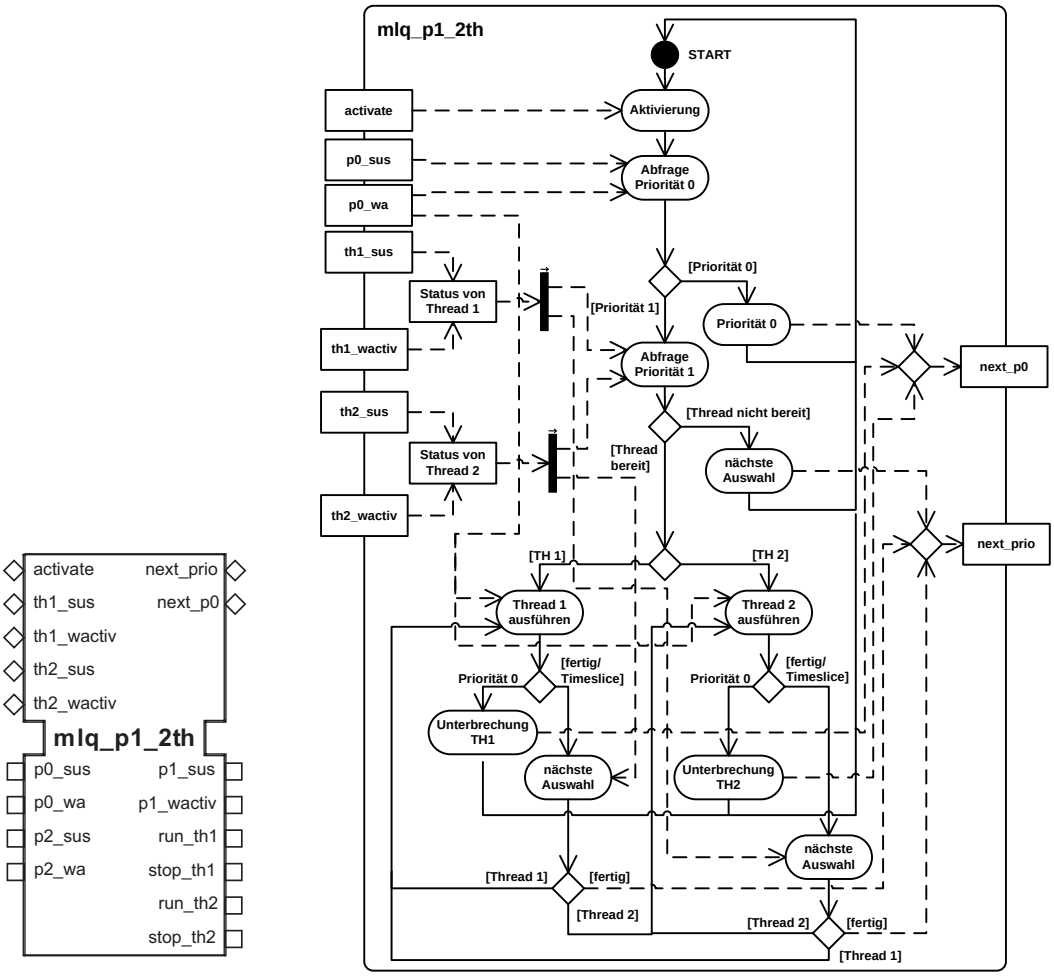


Abbildung 3.8: Multi Level Queue Scheduler a) NCES Modul b) Verhalten

Dies wird über die Bedingungsausgänge **p1_sus** (Priorität 1 ist inaktiv) und **p1_wa** (Priorität 1 ist bereit) bekanntgegeben.

Auch hier wurden wie in Abbildung 3.7b die Eingangsbedingungen **p2_sus** und **p2_wa** in Abbildung 3.8b nicht angeführt.

Mit diesem Interface des MLQ Schedulers hat man nun die Möglichkeit die Module der beiden Scheduler zu kombinieren, d. h. man kann z. B. einen Scheduler erstellen, der aus Modulen der Scheduler Bitmap und MLQ besteht. So kann pro Priorität entweder ein Bitmap oder MLQ Modul eingesetzt werden.

3.3.3 Modelle für Semaphore und Mutex

Semaphore

Der Semaphore (siehe Kapitel 3.1.3) wurde in dieser Arbeit nur soweit implementiert, als für die μCrons Laufzeitumgebung nötig war und steht im engen Zusammmenhang mit dem Event Dispatcher. Es wird beim Eintreten eines Ereignisses in den Event Dispatcher der Semaphore aktiviert. Sobald das Ereignis im Event Dispatcher hineingelegt wurde, wird der Semaphore deaktiviert und ein weiteres Ereignis kann in den Event Dispatcher hineingestellt werden. Der Semaphore überwacht sozusagen den Event Dispatcher und blockiert den Zugriff auf den Event Dispatcher, das gleiche gilt auch beim Herausnehmen eines Ereignisses aus der Event-Queue. Mit den Eingangsereignissen **SET** und **RESET** in Abbildung 3.9b wird das Modul des Semaphores aktiviert bzw. deaktiviert. Mit den Bedingungsausgängen **FALSE** und **TRUE** wird nach außen hin bekannt gegeben ob nun ein Zugriff, in diesem Fall am Event Dispatcher, erlaubt ist oder nicht.

Die Abbildung 3.9a zeigt nun einen Semaphore im Zusammenhang mit dem Event Dispatcher. Mit den Modulen **INregister** und **OUTregister** wird über das Ausgangsereignis **action** der **SEMAPHORE** aktiviert bzw. deaktvviert. Die Registermodule dienen somit für die Ansteuerung des Semaphores sobald ein Ereignis über **ei1 ... eix** im Event Dispatcher hineingestellt/herausgenommen werden möchte. In Abbildung 3.9b ist das Verhalten des Moduls **SEMAPHORE** und in Abbildung 3.9c das Verhalten des Moduls **INregister/OUTregister** dargestellt als Aktivitätsdiagramm. Betrachtet man das Verhalten vom **SEMAPHORE** in Abbildung 3.9b, dann gelangt man von **START** auf die Aktion **Bereit**, wobei die Bedingung **TRUE** gesetzt und **FALSE** rückgesetzt wird. Die Ausgangsbedingungen **TRUE** und **FALSE** werden jeweils immer gesetzt/rückgesetzt bzw. rückgesetzt/gesetzt. Sobald ein Ereignis an **SET** anliegt, wird von **Bereit** die Aktion **Aktiv** angesprochen. Hier wird die Bedingung **TRUE** rückgesetzt und **FALSE** wird gesetzt. Erst wenn das Eingangsereignis **RESET** generiert wird, gelangt man in den Zustand **Inaktiv**. Es wird wieder die Bedingung **TRUE** gesetzt und **FALSE** wird rückgesetzt. Schlußendlich wird auf *START* verzweigt und mit der Aktion **Bereit** kann ein weiteres Ereignis aufgenommen werden..

Die Module **INregister** und **OUTregister** haben dasselbe Verhalten. Sobald ein Ereignis am Ereigniseingang **eix** anliegt, werden zugleich die Ausgangsereignisse **eox** und **action** generiert. Die Bezeichnung **INregister** sagt aus, dass Ereignisse in den Event Dispatcher hineingestellt werden und mit **OUTregister** werden Ereignisse aus dem Event Dispatcher weitergeleitet.

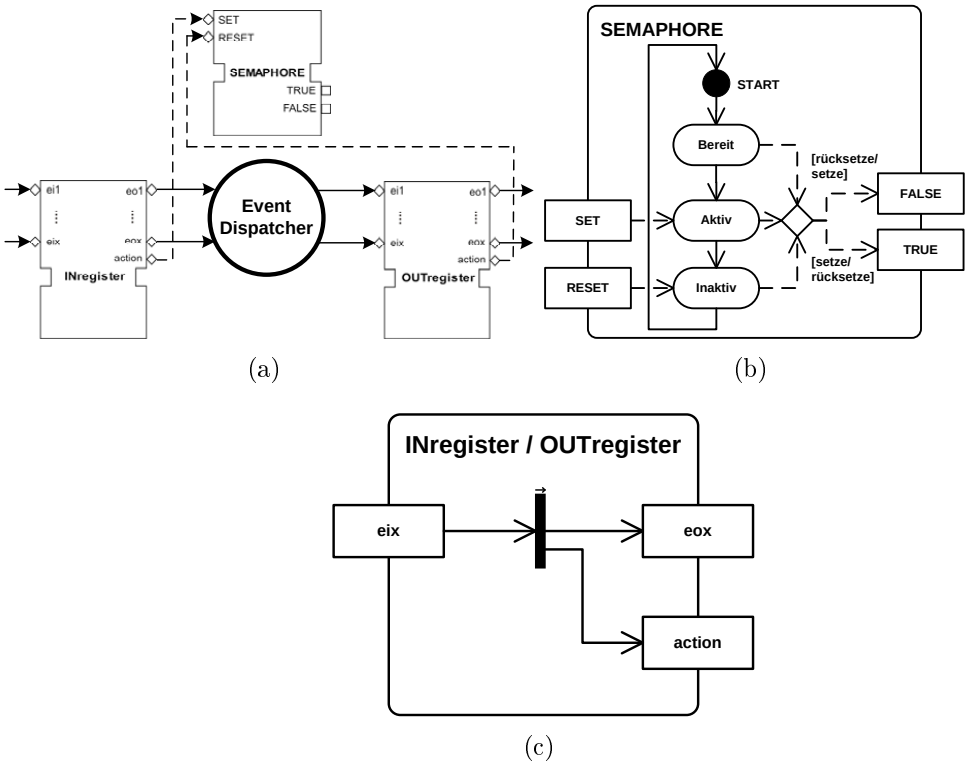


Abbildung 3.9: Semaphore im Überblick a) Allgemeines NCES Modell b) Semaphore Verhalten c) INregister und OUTregister Verhalten

Mutex

Wie schon im Kapitel 3.2.2 beschrieben, können einem Thread mehrere externe Ereignisse vom External Event Manager übergeben werden. Beim Thread handelt es sich dabei entweder um einen Realtime oder Background Thread. Die Verwaltung der externen Ereignisse in einem solchen Thread wird von einem Mutex gehandhabt. Speziell für die Verwendung in der μCrons Laufzeitumgebung wurde der Mutex wie folgt an diese angepasst. Der Mutex hat die Aufgabe, das externe Ereignis, das in einem Thread gelangt, zu „kapseln“ und dem Thread nach dessen „Erlaubnis“ weiterzuleiten. Der Mutex ist somit für die Kontrolle der Ablage von externen Ereignisse in einem Thread zuständig.

Das in Abbildung 3.10a dargestellte Modul zeigt einen Mutex der x externe Ereignisse aufnehmen kann. Die externen Ereignisse gelangen somit zuerst in die Eingangsereignisse **ExEv_x** (x steht für die Anzahl der externen Ereignisse) des Mutex in Abbildung 3.10a und werden dann dem Thread weitergeleitet.

Bevor aber das externe Ereignis dem Thread zur Verfügung steht, muss eine Kontrolle für die Ablage des Ereignisses im Thread durchgeführt werden. Wie im Kapitel 3.2.3 enthält ein Thread einen Event Dispatcher, der für die Ablage der Ereignisse in der Event-Queue zuständig ist. Im Zusammenhang mit dem Event Dispatcher wird dem Mutex die „Erlaubnis“ der Ablage des externen Ereignisses entweder erteilt oder nicht erteilt. Die „Erlaubnis“ der Ablage des externen Ereignisses hängt nun davon ab, ob im Event Dispatcher gerade ein Ereignis abgearbeitet wird oder nicht. Wird im Event Dispatcher nun kein Ereignis verarbeitet, dann kann der Mutex das externe Ereignis dem Event Dispatcher weiterleiten. Befindet sich jedoch ein Ereignis im Event Dispatcher für die Abarbeitung, so kann das externe Ereignis erst dann weitergeleitet werden, wenn das Ereignis im Event Dispatcher erfolgreich abgearbeitet wurde. Die Weiterleitung der externen Ereignisse erfolgt über das Ausgangsereignis **EvExSet_x** und mit dem Eingangsereignis **ExEvReady_x** wird über den Event Dispatcher dem Mutex mitgeteilt, dass das Ereignis im Event Dispatcher abgelegt wurde. Das Ausgangsereignis **ReadyExEv_x** gibt nach außen hin der Ereignisquelle bekannt, dass das externe Ereignis im Event Dispatcher nun enthalten ist. Mit den Bedingungseingängen **enable** und **NOTenable** in Abbildung 3.10a, wird der Mutex über die Ausgangsbedingungen **run_th** und **stop_th** vom Modul des Bitmap oder MLQ Schedulers angesteuert. Zugleich bilden diese Eingangsbedingungen einen Teil des Interfaces eines Threads, der in Kapitel 3.4.3 erklärt wird.

Das interne Verhalten des Mutex beschreibt das Aktivitätsdiagramm in Abbildung 3.10b. Im Mutex wird ab **Kontrolle für Ereignisablage im Event Dispatcher** überprüft, ob Ereignisse, die an **ExEv_x** anliegen, in den Event Dispatcher hineingestellt werden können. Für das Ablegen eines Ereignisses aus einer externen Ereignisquelle ergeben sich zwei Verzweigungen: **erlaubt** und **nicht erlaubt**. Diese Verzweigungen werden durch die Eingangsbedingungen **TRUE** und **FALSE** bestimmt. Diese Bedingungen erhält man vom zuvor beschrieben Semaphore, der im Zusammenhang mit dem Event Dispatcher integriert wurde. Mit der Verzweigung **erlaubt** wird bekannt gegeben, dass im Event Dispatcher kein Ereignis aktuell in Abarbeitung ist und der Event Dispatcher muss mit **Event Dispatcher aktivieren** bzw. mit der Ausgangsbedingung **enableD** aktiviert werden. Nun kann das Ereignis mit **ExEvSet_x** im Event Dispatcher abgelegt werden und nach erfolgtem Ablegen in der Event-Queue wird das Eingangsereignis **ExEvReady_x** generiert. Der Event Dispatcher wird mit **Event Dispatcher deaktivieren** über die Ausgangsbedingung **enableD** wieder deaktiviert und das Ausgangsereignis **ReadyExEv_x** wird gefeuert. Dieses Ereignis gelangt in der entsprechenden Ereignisquelle und teilt der Ereignisquelle mit, dass das Ereignis, das von

ExEv_x generiert wurde, im Event Dispatcher erfolgreich abgelegt wurde. Anschließend wird auf **START** zurückgekehrt.

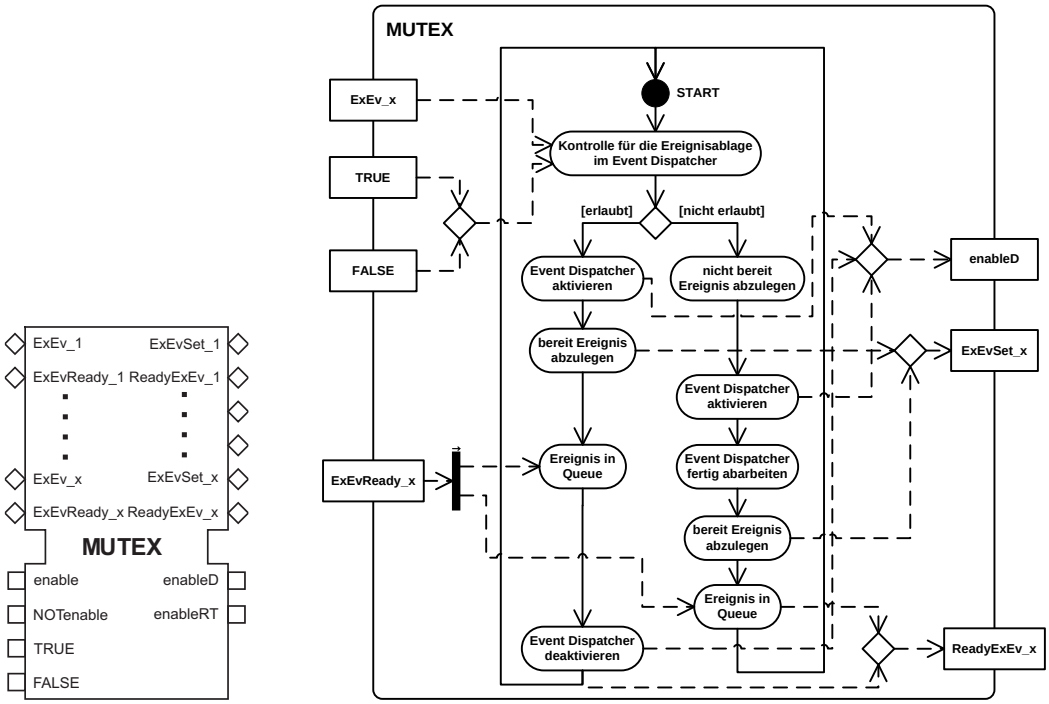


Abbildung 3.10: Mutex für x Ereignisquellen a) NCES Modul b) Verhalten

Befindet sich jedoch im Event Dispatcher ein Ereignis, das fertig abgearbeitet werden muss, dann gilt die Verzweigung **nicht erlaubt**, die durch die Eingangsbedingung **FALSE** verursacht wurde. Bei dieser Verzweigung kann das Ereignis aus der externen Ereignisquelle, wie zuvor beschrieben, nicht sofort im Event Dispatcher abgelegt werden. Man befindet sich im Zustand **nicht bereit Ereignis abzulegen**. Der Event Dispatcher muss mit **Event Dispatcher aktivieren** bzw. **enableD** aktiviert werden. Der Event Dispatcher führt eine Ereignisabarbeitung durch und nach dessen Abarbeitung kann mit **bereit Ereignis abzulegen** das externe Ereignis von **ExEv_x** mit **ExEvSet_x** in dem Event Dispatcher hineingestellt werden. Mit dem Eingangsereignis **ExEvReady_x** wird dem Mutex bekannt gegeben, dass das Ereignis in der Event-Queue des Event Dispatchers abgelegt wurde und es wird das Ausgangsereignis **ReadyExEv_x** generiert. Wie bei der Verzweigung **erlaubt** wird auch hier wieder auf **START** verzweigt.

Die Eingangsbedingung **NOTenable** und Ausgangsbedingung **enableRT** wurden in der Abbildung 3.10b nicht eingezeichnet. Mit der Bedingung **NOTenable** wird der Mutex vom Thread bzw. vom Scheduler deaktiviert. Mit dem Ausgang **enableRT** wird lediglich das Modul **RT_Control** (siehe Kapitel 3.4.2) aktiviert.

3.3.4 Modell des External Event Timer

In diesem Kapitel soll nun der External Event Timer als formales Modell beschrieben werden, welcher aus den Elementen Timer und Callback-Function (siehe Kapitel 3.1.4) modelliert wurde. In dieser Arbeit wurde der External Event Timer als Thread modelliert und nicht als Callback-Function. Weiters wird der External Event Timer im Zusammenhang mit den formalen Modellen des eCos Schedulers auf der höchsten Priorität implementiert. Somit wird eine aktive Abarbeitung des External Event Timers gewährleistet.

Der External Event Timer hat die Aufgabe, Zeit- Funktionsblöcke (z.B. E_CYCLE), die in verschiedenen Funktionsblocknetzwerken enthalten sein können, zu verarbeiten. Jeder Zeit- Funktionsblock enthält im External Event Timer jeweils einen fix zugeordneten *TimerInterface* (Abk.: TI). Der *TimerInterface* selbst definiert z.B. die Zeitdauer DT (duration time) eines Zeit-Funktionsblockes. Diese Zeitdauer im *Timerinterface* wurde mit einem Zähler implementiert. Mit einem Element *TimerInterface Control* wird jeder *TimerInterface* auf dessen Abarbeitung kontrolliert. Diese beiden Elemente, *TimerInterface* und *TimerInterface Control*, repräsentieren die Callback-Function. Der *Timer* im Modul External Event Timer generiert jede 1ms einen Interrupt, womit dem Scheduler bekannt gegeben wird, dass der als Thread implementierte External Event Timer bereit zur Abarbeitung ist. Der *Timer* entspricht somit dem Element des eCos Timers.

Das Verhalten des External Event Timers nach außen hin wird anhand der Abbildung 3.11a erklärt. Jeder TimerInterface im External Event Timer wird über die Eingangsereignisse **TIxstart**, **TIxstop** und **TIxFBready** angesteuert (das x im Ereignisnamen steht für den entsprechenden Zeit- Funktionsblock). Mit dem Eingangsereignis **TIxstart** teilt der Zeit- Funktionsblock dem External Event Timer mit, dass der Zähler des TimerInterface aktiv werden soll und mit jedem Interrupt[11] des External Event Timers weitergezählt werden soll. **TIxstop** teilt dem External Event Timer über dem Zeit- Funktionsblock mit, dass der Zähler deaktiviert werden soll. Mit dem Ausgangsereignis **TIxinvoke** wird dem entsprechenden Zeit- Funktionsblock mitgeteilt, dass der

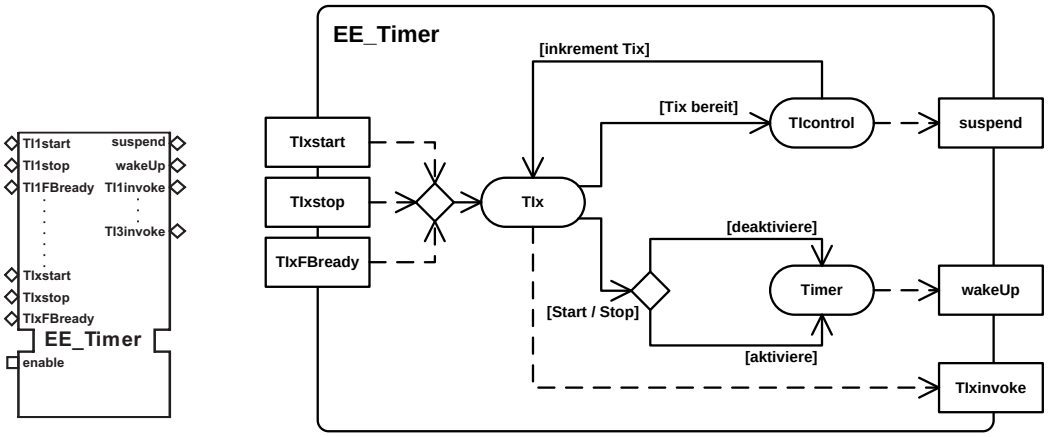
[11] deut.: Unterbrechung

Zähler im TimerInterface abgelaufen ist. Im folgenden wird der Zeit- Funktionsblock aktiviert und die entsprechenden Aktionen können vom Zeit- Funktionsblock ausgelöst werden. Über das Eingangsereignis **TIxFBready** übergibt der Zeit- Funktionsblock dem External Event Timer die Information, dass die Aktionen nach dem Ereignis **TIxinvoke** erfolgreich waren. Weiters wird die Abarbeitungskontrolle dem External Event Timer wieder übergeben. Die Ausgangsereignisse **suspend** und **wakeUp** haben die selbe Funktion wie im Kapitel 3.4.3 erläutert. Mit der Eingangsbedingung **enable** wird der External Event Timer über die Ausgangsbedingung **run_th** von einem Scheduler (siehe Kapitel 3.3.1 und 3.3.2) entweder aktiviert oder deaktiviert.

Für das interne Verhalten des External Event Timers wird Abbildung 3.11b betrachtet. Das Modul **EE_Timer** (EE steht für External Event) besteht im allgemeinen aus den formalen Modellen **Timer**, **TIcontrol** und einer Anzahl an **TIx**. Das formale Modell **TIx** entspricht dem TimerInterface, das den Zeit-Funktionsblock (z. B. E_CYCLE) im External Event Timer repräsentiert. Sobald ein Ereignis am Ereigniseingang **TIxstart** anliegt, wird der entsprechende TimerInterface **TIx** aktiviert und der interne Zähler gestartet. Zugleich wird über **Start/Stop** der **Timer** mit der Verzweigung **aktiviere** aktiviert. Dieser generiert in bestimmten Zeitabständen mit dem Ausgangsereignis **wakeUp** ein Ereignis. Deaktiviert wird der **Timer** durch die Bedingung **deaktiviere**, die erst dann anspricht, wenn alle TimerInterfaces **TIx** nicht mehr aktiv sind. Mit diesem Ausgangsereignis steht der External Event Timer für die Abarbeitung bereit. Der Scheduler erhält die Information, dass der External Event Timer abgearbeitet werden möchte. Sobald der External Event Timer Abarbeitungszeit vom Scheduler bekommt, wird über **TIcontrol** mit **inkrement TIx** der Zähler des entsprechenden TimerInterface **TIx** erhöht. Es werden dabei nur die Zähler jener TimerInterface erhöht, die zuvor mit **TIxstart** aktiviert wurden. Anschließend wird mit **TIx bereit** die Aktion **TIcontrol** für die nächste Kontrolle in einem TimerInterface fortgesetzt und am Ende der Kontrolle wird das Ausgangsereignis **suspend** generiert.

Hat nun ein Zähler in einem **TIx** den vorgegebenen Wert erreicht, dann wird der Zähler auf Null gesetzt und das Ausgangsereignis **TIxinvoke** wird zum jeweiligen Zeit- Funktionsblock generiert. Sobald die ID des zugehörigen Zeit-Funktionsblocks in den Event Dispatcher gestellt wurde (siehe Abbildung 3.6), wird über das Eingangsereignis **TIxFBready** die Kontrolle von **TIcontrol** fortgesetzt.

Tritt jedoch vor dem Zählerablauf ein Eingangsereignis **TIxstop** auf, so wird der aktuelle Zählerstand auf Null zurückgesetzt und der TimerInterface **TIx** deaktiviert.



(a) (b)

Abbildung 3.11: External Event Timer a) NCES Modul b) Verhalten

3.4 Formale Modelle der μCrons Laufzeitumgebung

Das folgende Kapitel gibt einen Überblick über die erstellten Modelle für die μCrons Laufzeitumgebung. Wie schon in Kapitel 3.3 angemerkt, stellen die folgenden Abbildungen auf der linken Seite das äußere Interface des Moduls in NCES dar und auf der rechten Seite das dazugehörige Aktivitätsdiagramm.

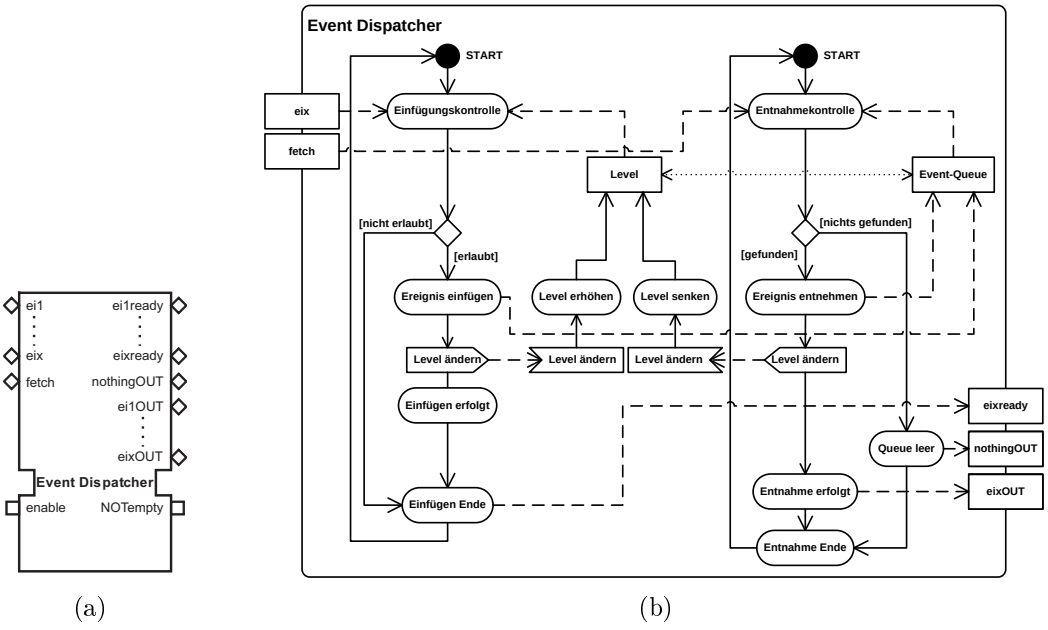
3.4.1 Modell des Event Dispatchers

Der Event Dispatcher bildet die Grundlage der Laufzeitumgebung. Wie bereits in Kapitel 3.2.3 erläutert, werden mit dem Event Dispatcher IEC 61499 Ereignisse in einer Event-Queue abgelegt und wieder für ein IEC 61499 Funktionsblocknetzwerk zur Verfügung gestellt. Die IEC 61499 Ereignisse gelangen von einer Ereigniquelle in den Event Dispatcher. Dieser kann nun im Falle der μCrons Laufzeitumgebung beliebig viele IEC 61499 Ereignisse enthalten. Für die Differenzierung von unterschiedlich ankommenden Ereignissen wurde jedem Ereignis im Event Dispatcher eine eindeutige Nummer zugewiesen. Zur Veranschaulichung des formalen Modells des Event Dispatchers dient zum einen das Modul in NCES in Abbildung 3.12a und zum anderen das Verhalten in Abbildung 3.12b. Der Event Dispatcher in Abbildung 3.12 besteht aus x verschiedenen Ereigniseingängen **eix** und einer internen **Event-Queue**, die eine Anzahl (implementierungsabhängig) an Ereignissen beinhalten kann. Mit

diesen Ereigniseingängen werden IEC 61499 Ereignisse definiert.

Möchte man ein IEC 61499 Ereignis in den Event Dispatcher hineinstellen, dann muss am entsprechenden Eingangsereignis **eix** ein Ereignis anliegen. Wurde dieses Ereignis erfolgreich in der **Event-Queue** abgelegt, erfolgt eine Generierung vom entsprechenden Ereignisausgang **eixready**. Liegt z. B. ein Ereignis am Eingangsereigis **ei6**, dann wird im Dispatcher das IEC 61499 Ereignis mit der Nummer 6 in der **Event-Queue** abgespeichert.

Beim Herausnehmen eines Ereignisses aus dem Event Dispatcher muss ein Ereignis am Ereignisseingang **fetch** anliegen. Mit **fetch** wird ein Ereignis aus der **Event-Queue** nach dem FIFO Prinzip herausgenommen und es wird der entsprechende Ereignisausgang **eixOUT** generiert. Besitzt hingegen die **Event-Queue** kein Ereignis (**Event-Queue** ist leer) dann wird der Ereignisausgang **nothingOUT** aktiviert. Mit der Eingangsbedingung **enable** in Abbildung 3.12a wird das Modell aktiviert bzw. deaktiviert. Mit dieser Bedingung kann das Modul Event Dispatcher Ereignisse abarbeiten bzw. nicht abarbeiten. Weiters ist noch zu erwähnen, dass für die Aktivierung/Deaktivierung des Moduls Event Dispatcher das Modul Mutex mit der Ausgangsbedingung **enableD** (siehe Kapitel 3.3.3) zuständig ist. Die Ausgangsbedingung **NOTempty** gibt nach außen hin bekannt, ob die **Event-Queue** leer oder nicht leer ist.



(a) (b)

Abbildung 3.12: Event Dispatcher a) NCES Modul b) Verhalten

Intern besteht das Modell des Event Dispatchers aus den Modellen **Ein-**

fügungskontrolle, **Entnahmekontrolle**, **Level** und **Event-Queue**, die jedoch im folgenden nur im Aktivitätsdiagramm in Abbildung 3.12b dargestellt werden und im nachkommenden Text erklärt werden. Das interne Verhalten basiert auf den zentralen Elementen **Einfügungskontrolle** und **Entnahmekontrolle**. Weiters gibt **Level** den Füllstand der **Event-Queue** an, der durch **Level erhöhen** und **Level senken** geändert werden kann.

Mit der **Einfügungskontrolle** wird anhand eines Eingangsereignisses **eix** und mit **Level** eine Kontrolle für das Hineinstellen eines Ereignisses in die **Event-Queue** durchgeführt. Zum einen wird in der **Event-Queue** überprüft, ob sie voll mit Ereignissen oder ob sie noch nicht voll ist. Ergibt sich nach der **Einfügungskontrolle** ein **erlaubt**, dann wird das Ereignis in der **Event-Queue** über **Ereignis einfügen** hineingestellt, der Wert von **Level** wird durch **Level erhöhen** erhöht und anschließend wird nach **Einfügen erfolgt** über **Einfügen Ende** das Ereignis **eixready** gesendet. Zum Schluß wird mit **Einfügen Ende** auf **START** zurückgekehrt. Ist die **Event-Queue** hingegen voll, wird mit **nicht erlaubt** auf **Einfügen Ende** verzweigt, das Ereignis **eixready** generiert und wieder auf **START** retourniert.

Für das Herausnehmen eines Ereignisses aus der **Event-Queue** muss die **Entnahmekontrolle** durchgeführt werden. Liegt ein Ereignis an **fetch** an, wird in der **Event-Queue** überprüft, ob Ereignisse vorhanden sind. Wie schon zuvor erwähnt, werden in der **Event-Queue** Ereignisse nach dem FIFO Prinzip gehandhabt. Das erste Ereignis, das in die **Event-Queue** hineingestellt wurde, wird auch als erstes entnommen. Beim Entnehmen eines Ereignisses aus der **Event-Queue** ergibt sich als Ergebnis in Abbildung 3.12b entweder **gefunden** oder **nichts gefunden**. Bei **gefunden** wird mit **Ereignis entnehmen** das Ereignis aus der **Event-Queue** entnommen und der Füllstand von **Level** mit **Level ändern** bzw. **Level senken** erniedrigt. Anschließend wird mit **Entnahme erfolgt** das Ausgangsereignis **eixOUT** generiert und auf **Entnahme Ende** verzweigt, wobei zur **Entnahmekontrolle** zurückgekehrt wird. Enthält die **Event-Queue** kein Ereignis, dann wird mit **Event-Queue leer** das Ereignis **nothingOUT** generiert und auf **Entnahmekontrolle** verwiesen.

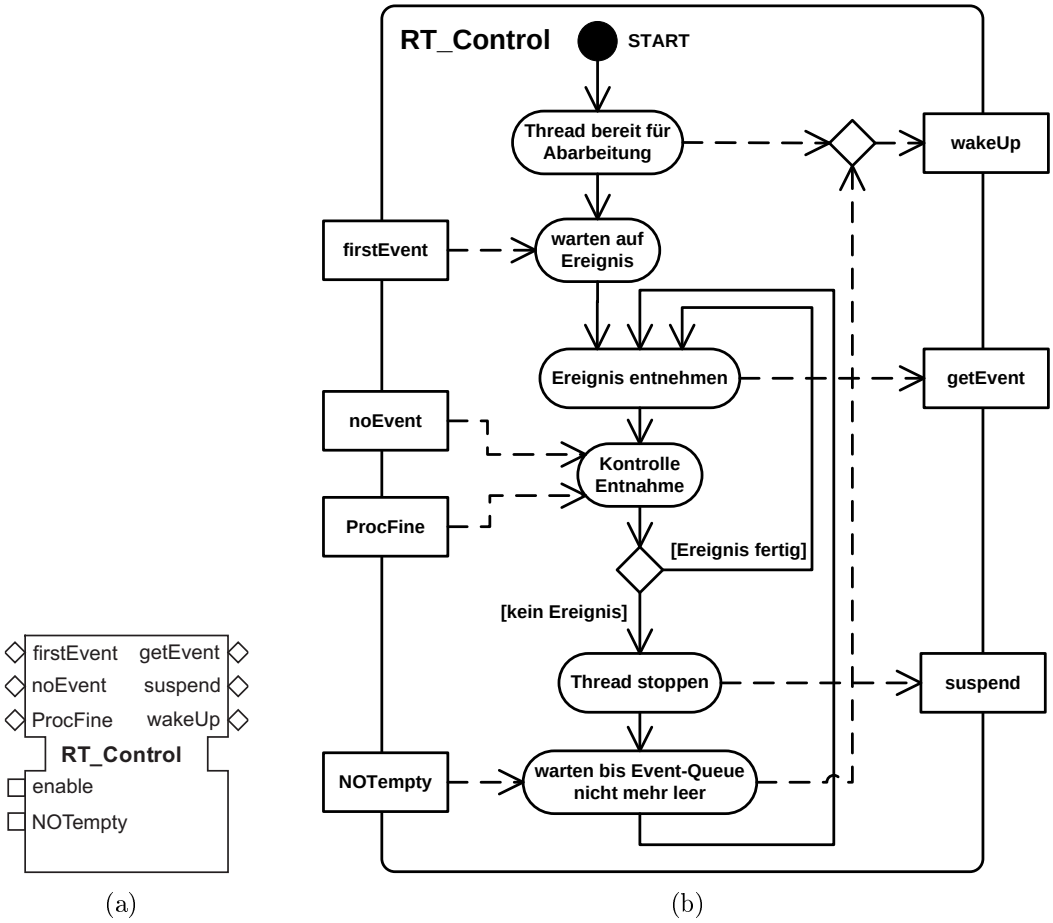
Wie man sieht wird immer wieder auf **START** zurückgesprungen. Dies ermöglicht ein wiederholtes Prüfen, ob nun ein Ereignis in der **Event-Queue** hineingestellt oder entnommen werden soll. Weiters ist noch anzumerken, das die Elemente **Level** und **Queue** miteinander „reden“. Diese geben untereinander und auch nach außen hin Informationen zum Füllstand der Event-Queue bekannt.

3.4.2 Kontrolle für einen Thread mit Funktionsblocknetzwerk

Mit dieser Art von Kontrolle für einen Thread wird gewährleistet, dass der Thread am Anfang für die Abarbeitung aktiviert wird. Weiters wird mit dieser Kontrolle das Weiterleiten von Ereignissen in einem Thread aufrecht erhalten, indem im Event Dispatcher nachgeschaut wird, ob ein Ereignis für die Abarbeitung des Funktionsblocknetzwerkes vorhanden ist oder nicht. Sobald kein Ereignis mehr im Event Dispatcher vorhanden ist, wird der Thread unterbrochen. Somit wird zum einen ein Thread angesteuert und zum anderen sein internes Funktionsblocknetzwerk überwacht.

Abbildung 3.13 zeigt links das Modul in NCES für die Kontrolle eines Threads und rechts ist das interne Verhalten des Moduls als Aktivitätsdiagramm beschrieben. Für das Verhalten nach außen hin betrachte man das Modul in Abbildung 3.13a. Geht man von einem Thread mit einem Funktionsblocknetzwerk wie in Abbildung 3.6 aus, dann wird für den ersten Funktionsblock das Eingangsereignis im Event Dipatcher abgelegt. Sobald dieses erste Ereignis in den Event Dispatcher erfolgreich hineingestellt wurde, wird das Eingangsereignis **firstEvent** angestoßen. Im Normalfall wird das erste Ereignis vom IEC 61499 Funktionsblock E_RESTART generiert. Der Funktionsblock E_RESTART selbst wird erst dann aktiv, wenn die dazugehörige Applikation gestartet wurde. Anschließend wird das Ausgangsereignis **getEvent** generiert. Dieses Ausgangsereignis ist mit dem Eingangsereignis **fetch** vom Event Dispatcher verbunden, wobei in der Event-Queue nachgeschaut wird, ob ein Ereignis für die Abarbeitung zur Verfügung steht oder nicht. Das Eingangsereignis **noEvent** ist in Verbindung mit dem Ausgangsereignis **nothingOUT** vom Event Dispatcher zu sehen. Wurde kein Ereignis im Event Dispatcher gefunden, dann wird das Eingangsereignis **noEvent** angestoßen und die Abarbeitung des Threads wird mit **suspend** angehalten. Der Thread wird solange angehalten, bis die Bedingung **NOTempty** anspricht, also wenn die **Event-Queue** im Event Dispatcher nicht mehr leer ist. Die Eingangsbedingung **NOTempty** ist mit der gleichnamigen Ausgangsbedingung **NOTempty** des Event Dispatchers verbunden. Befand sich jedoch ein Ereignis im Event Dispatcher, dann wird nach dessen erfolgreicher Weiterleitung der Ereigniseingang **ProcFine** generiert. Dadurch wird das Ereignis **getEvent** generiert und es wird im Event Dispatcher wieder nach einem Ereignis gesucht. Das geht so lange, bis kein Ereignis mehr in der **Event-Queue** vorhanden ist. Mit den Ausgangsereignissen **suspend** und **wakeUp** wird nach außen hin bekannt gegeben, dass der Thread unterbrochen wurde oder für die Abarbeitung bereit gestellt wurde. Mit der Eingangsbedingung **enable** in Abbildung 3.13a wird,

wie zuvor beim Modell des Event Dispatchers erklärt, das Modell aktiviert bzw. deaktiviert.



(a) (b)

Abbildung 3.13: Kontrolle für einen Thread der μCrons Laufzeitumgebung a) NCES Modul b) Verhalten

Betrachtet man das interne Verhalten in Abbildung 3.13b, dann gelangt man von **START** nach **Thread bereit für Abarbeitung**, wobei das Ausgangsereignis **wakeUp** generiert wird. Im folgenden gelangt man auf **warten auf Ereignis**, wo auf die Generierung des Eingangsereignisses **firstEvent** gewartet wird. Sobald der Thread abgearbeitet wird, wird E_RESTART ausgeführt, der dann mit **firstEvent** endet. Mit diesem Ereignis wird auf die Aktion **Ereignis entnehmen** verzweigt und das Ausgangsereignis **getEvent** gesendet. In **Kontrolle Entnahme** wird gewartet, bis einer der beiden Eingangsereignisse **noEvent** oder **ProcFine** anspricht. Mit dem Ereignis **noEvent** (kein Ereig-

nis in Event-Queue) wird über **kein Ereignis** zur Aktion **Thread stoppen** verzweigt und zugleich der Thread mit **suspend** unterbrochen. Erst sobald die Eingangsbedingung **NOTempty** anspricht, wird über der Aktion **Warten bis Event-Queue nicht mehr leer** das Ausgangsereingnis **wakeUp** generiert und auf **Ereignis entnehmen** zurückverzweigt. Die Eingangsbedingung **NOTempty** wird erst dann aktiviert, wenn ein externes Ereignis im Thread für die Abarbeitung zur Verfügung steht. Im Falle von **ProcFine** (Ereignis in Event-Queue) wird über **Ereignis fertig** zur Aktion **Ereignis entnehmen** retourniert. Das Funktionsprinzip der Kontrolle für einen Thread ist ab der Aktion **Ereignis entnehmen** gekennzeichnet, wo mit einer Schleife immer auf diese Aktion zurückgekehrt wird. Diese Schleife bewirkt somit eine ständige Kontrolle für den Thread.

3.4.3 Laufzeitumgebung als formales Modell

Bei diesem Modell handelt es sich im Grunde um den, in Kapitel 3.2.2 erwähnten, Event Chain Executor. Mit diesem Modell soll erreicht werden, dass Ereignisse in einem Thread mit Funktionsblocknetzwerk auf korrekte Weise weitergeleitet werden. Somit reicht für die Weiterleitung von Ereignissen in einem Thread mit Funktionsblocknetzwerk das formale Modelle des Event Dispatcher in Kapitel 3.4.1 nicht aus. Im Zusammenhang mit dem formalen Modell der Kontrolle für einen Thread (siehe Kapitel 3.4.2) kann ein Weiterführen der Ereignisse in einem Thread gewährleistet werden, da mit dem Modell **RT_Control** im Modell **Event Dispatcher** immer wieder nachgeschaut wird, ob Ereignisse zum Weiterleiten vorhanden sind. So werden mit dem Modell **Event Dispatcher** Ereignisse in einer Event-Queue abgelegt und mit dem Modell **RT_Control** werden Ereignisse aus der Event-Queue entnommen. Die Verknüpfung der beiden Modelle ist somit die wesentliche Grundlage des formalen Modells der Laufzeitumgebung.

Für das Verhalten nach außen hin soll das Modul **Thread_exe** in Abbildung 3.14a herangezogen werden. Mit den Eingangsereignissen **inEVx** und **endEVx** wird dem Modul bekannt gegeben, dass Ereignisse in der Event-Queue abgelegt werden wollen und Ereignisse erfolgreich abgearbeitet wurden. Die Aktivierung des Eingangsereignisses **firstEV** bewirkt die Entnahme des ersten Ereignisses aus der Event-Queue, das zuvor mit **inEVx** hineingestellt wurde. Mit den beiden Ausgangsereignissen **suspend** und **wakeUp** wird nach außen hin bekannt gegeben, dass der Thread entweder fertig abgearbeitet wurde oder bereit für die Abarbeitung ist. Diese Ausgangsereignisse werden mit den Eingangsereignissen **th_sus** und **th_wactiv** eines Schedulers in Kapitel 3.3.1 oder 3.3.2 verbunden. Das Ausgangsereignis **readyEVx** teilt dem

Funktionsblock mit, dass ein Ereignis in der Event-Queue erfolrgreich hineingestellt wurde. Mit dem Ausgangsereignis **outEVx** wird das Ereignis dem entsprechenden Funktionsblock weitergeleitet. Mit diesem Ereignis kann der Funktionsblock abgearbeitet werden.

Für die Aktivierung des Moduls **Thread_exe** in Abbildung 3.14a dienen die beiden Eingangsbedingungen **enableD** und **enableRT**. Mit diesen beiden Eingangsbedingungen werden die formalen Modelle **Event Dispatcher** und **RT_Control** unabhängig voneinander aktiviert bzw. deaktiviert. Die Aktivierung der Eingänge erfolgt über die gleichnamigen Ausgangsbedingungen **enableD** und **enableRT** des Moduls **MUTEX**, das in Kapitel 3.3.3 erklärt wurde. Wie bereits in den Kapiteln 3.2.2 und 3.3.3 können einem Thread externe Ereignisse zugeführt werden, die zuvor ins formale Modell des **Mutex** gelangen. So wird mit den Ausgangsbedingungen **true** und **false** in Abbildung 3.14a dem Mutex mitgeteilt, ob ein Zugriff auf das Modul **Thread_exe** erlaubt oder nicht erlaubt ist. Die Ausgangsbedingungen werden dann direkt mit den gleichnamigen Eingangsbedingungen des Mutex Moduls (siehe Abbildung 3.10) verbunden. Mit der Tabelle 3.2 wird gezeigt, wie das Modul **Thread_exe** nach außen hin in Verbindung gebracht wird.

Das interne Verhalten des Moduls **Thread_exe** in Abbildung 3.14a soll nun mit Abbildung 3.14b näher gebracht werden. Das Modul besteht aus den formalen Modellen **Event Dispatcher** (siehe Kapitel 3.4.1), **RT_Control** (siehe Kapitel 3.4.2) und **SEMAPHORE** (siehe Kapitel 3.3.3). Diese formalen Modelle werden mit den gleichnamigen Aktionen in Abbildung 3.14b in Verbindung gesetzt, d. h. sie repräsentieren die zuvor angeführten formalen Modelle. Von **START** beginnend, wird zuerst über der Aktion **RT_Control** der Thread mit **wakeUp** zur Abarbeitung beim Scheduler angemeldet (siehe Kapitel 3.4.2). Die Aktion **RT_Control** generiet erst dann das Signal **Ereignis holen**, sobald ein Ereignis an **firstFV** oder **endEV** anliegt. Mit einem beliebigen Eingangsereignis **inEVx** wird das Ereignis zuerst in **Ereigniseingabe** gestellt, wobei mit dem **SEMAPHORE** über die Verzweigung **kein Zugriff** die Ausgangsbedingung **true** aktiviert wird. Mit dieser Bedingung wird verhindert, dass kein weiteres Ereignis im Event Dispatcher hineingestellt werden darf. Erst wenn das aktuelle Ereignis erfolgreich im **Event Dispatcher** abgelegt wurde, wird auf die Aktion **Ereignisausgabe** geführt. Ab der Aktion **Ereignisausgabe** werden die Ausgangsereignisse **readyEVx** und **outEVx** generiert und die Aktionen **SEMAPHORE** und **RT_Control** aktiviert. Mit dem **SEMAPHORE** wird über die Verzweigung **Zugriff** die Ausgangsbedingung **false** gesetzt. Mit dieser ist es wieder erlaubt, ein Ereignis in den Event Dispatcher hineinzustellen.

Bei **RT_Control** wird gewartet bis ein Eingangsereignis **firstEV** (beim

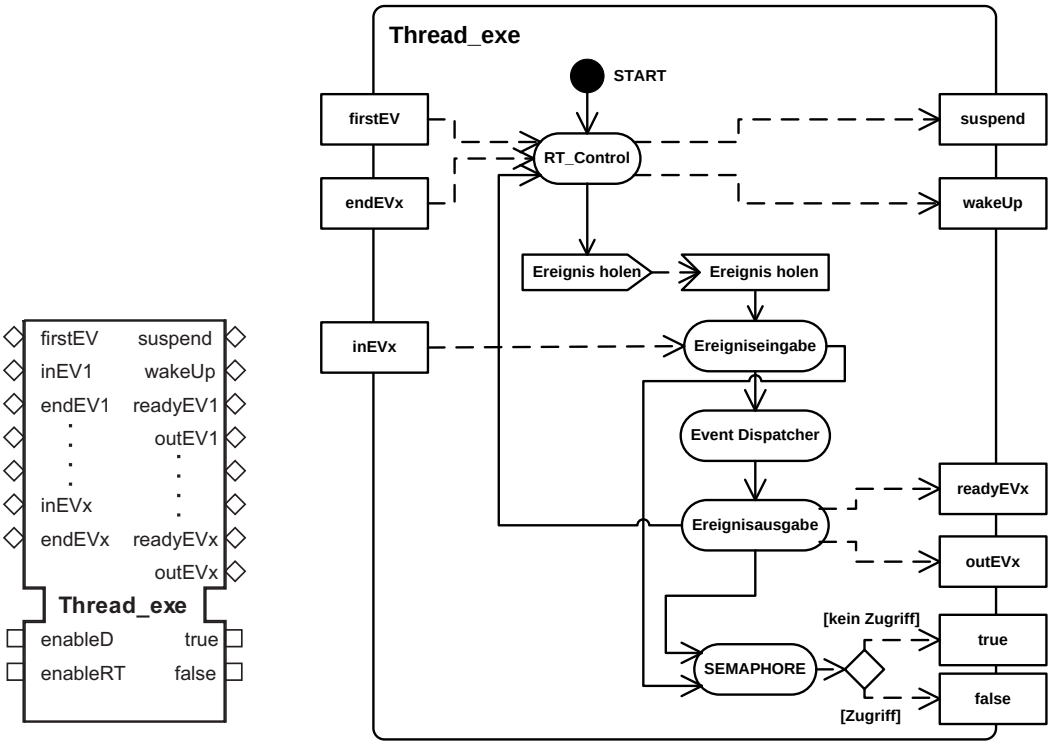
Interface zum	**Bezeichnung**
Funktionsblocknetzwerk	
readyEVx	Ereignis in Event Dispatcher erfolgreich hineingestellt
outEVx	Ausgangsereignis für FBN Abarbeitung
Mutex	
true	kein Zugriff auf Event Dispatcher
false	Zugriff auf Event Dispatcher
Scheduler	
suspend	Abarbeitung von Thread fertig
wakeUp	bereit für Abarbeitung von Thread
Interface vom	**Bezeichnung**
Funktionsblocknetzwerk	
firstEV	erstes Ereignis vom FB E_RESTART
inEVx	Ereignis wird in Event Dispatcher hineingestellt
endEVx	Ereignis wurde abgearbeitet
Mutex	
enableD	Aktivierung/Deaktivierung von Event Dispatcher
enableRT	Aktivierung/Deaktivierung von Kontrolle für einen Thread

Tabelle 3.2: Interface für das Modul Thread_exe

ersten Ereignis) oder **endEVx** (bei den folgenden Ereignissen) angestoßen wrid. Bei der Aktivierung einer der beiden Eingangsereignisse kann nun über **RT_Control** ein nächstes Ereignis für die Abarbeitung eines Funktionsblockes aus dem **Event Dispatcher** mit **Ereignis holen** herausgenommen werden. Dabei wird wiederum der **Event Dispatcher** auf **true** gesetzt und ein Zugriff auf diesen wird, bis die Bedingung **false** auftritt, verweigert.

3.4.4 Ereignisverzweigung als formales Modell

In diesem Kapitel wird das formale Modell für die Ereingnisverzweigung in IEC 61499 dargestellt. Als Beispiel für eine Ereignisverzweigung in IEC 61499



(a) (b)

Abbildung 3.14: Ausführbarer Event Dispatcher a) NCES Modul b) Verhalten

dient Abbildung 3.15a. Der Funktionsblock **FB1** in Abbildung 3.15a generiert mit dem Ausgangsereignis **EO** ein Ereignis, das durch eine Verzweigung zu den Funktionsblöcken **FB2** und **FB3** weitergeleitet wird. Durch diese Ereignisverzweigung entstehen zwei Eingangsereignisse, die in dieser Arbeit für ihrer Unterscheidung nummeriert wurden. Die Ereignisverzweigung im Standard IEC 61499 nach Abbildung 3.15a wurde in NCES durch das Modul in Abbildung 3.15b realisiert. Bei Verwendung von mehreren Ereignisverzweigungen kann dieses NCES Modul selbstverständlich variiert werden.

Das Modul **SWITCH2** in Abbildung 3.15b verhält sich nach außen hin wie folgt. Sobald ein Ereignis am Ereigniseingang **ev_in** anliegt, wird über das Ausgangsereignis **ev_out1** ein Ereignis im Modul **Thread_exe** über das Eingangsereignis **inEVx** (siehe Abbildung 3.14) hineingestellt. Wurde das Ereignis im Event Dispatcher erfolgreich abgelegt, wird über das Ausgangsereignis **readyEVx** vom Modul **Thread_exe** das Eingangsereignis **readyev1** angestoßen und das Ausgangsereignis **ev_out2** generiert. Hier gilt wiederum

das selbe Verfahren wie zuvor beschrieben. Wird nun **readyev2** aktiviert, dann ist die Ablage der beiden Ereignisse im Event Dispatcher erfolgt und es wird mit dem Ausgangsereignis **end** z. B. das Eingangsereignis **readyCNF** vom Modul **simpleFB** (siehe Abbildung 3.16) aktiviert. Mit diesem Ausgangsereignis wird dem Modul **simpleFB** mitgeteilt, dass die Ereignisse erfolgreich „verzweigt“ (in den Event Dispatcher hineingestellt) wurden.

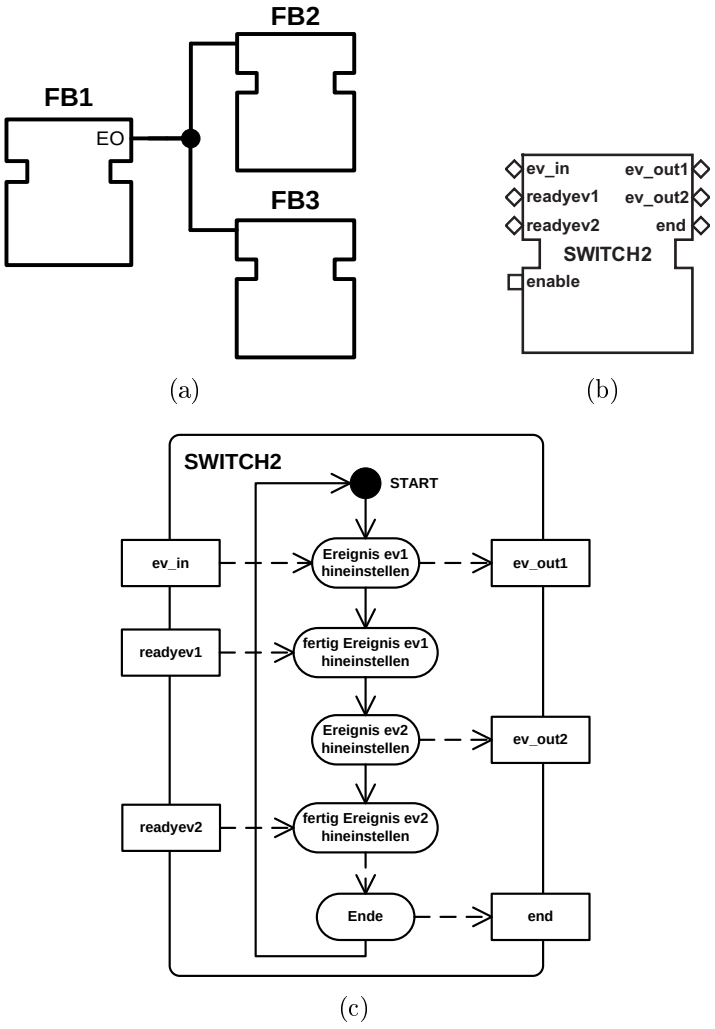


Abbildung 3.15: Verbindung von einem Ausgangsereignis auf 2 Eingangsereignisse a) IEC 61499 Ereignisverzweigung b) NCES Modul c) Verhalten

Betrachtet man das interne Verhalten des Moduls **SWITCH2** in Abbildung 3.15c, so gelangt man von **START** zur Aktion **Ereignis ev1 hinein-**

stellen, wo auf die Aktivierung des Ereigniseinganges **ev_in** gewartet wird. Sobald das Ereignis **ev_in** anspricht, generiert die Aktion das Ausgangsereignis **ev_out1**. Hier wird, wie zuvor beschrieben, das Ereignis im Event Dispatcher über das Ausgangsereignis **ev_out1** hineingestellt. Nach erfolgreicher Ablage des Ereignisses, wird der Ereigniseingang **readyev1** aktiviert und über die Aktion **fertig Ereignis ev1 hineinstellen** wird auf **Ereignis ev2 hineinstellen** verzweigt. Hier wird das Ausgangsereignis **ev_out2** generiert und in der Aktion **fertig Ereignis ev2 hineinstellen** wird auf die Aktivierung des Eingangsereignisses **readyev2** gewartet. Nach Generierung des Eingangsereignisses **readyev2** gelangt man auf die Aktion **Ende**, das das Ausgangsereignis **end** evaluiert und auf **START** retourniert wird.

3.4.5 Funktionsblöcke als formale Modelle

Im Kapitel 2.3.1 wurden die IEC 61499 Funktionsblöcke *Basic FB*, *Composite FB* und *Service Interface FB* bereits erklärt. Für diese Arbeit wurden jedoch diese Funktionsblöcke nicht berücksichtigt. Für die Darstellung eines Funktionsblocknetzwerkes als formales Modell wurden zwei Arten von Funktionsblöcken realisiert. Zum einen handelt es sich um das formale Modell eines einfachen Funktionsblockes und zum anderen um die formalen Modelle der Zeit- Funktionsblöcke E_CYCLE und E_DELAY.

Einfacher Funktionsblock als formales Modell

Für die Erklärung des einfachen Funktionsblockes soll auf Abbildung 3.16 eingegangen werden. Der Funktionsblock ist gemäß Standard IEC 61499 mit jeweils einem Ereignisein- und ausgang **REQ** und **CNF** ausgestattet. Sobald ein Ereignis an **REQ** anliegt, ist der Funktionsblock bereit für die Abarbeitung und nach dessen Fertigstellung wird das Ausgangsereignis **CNF** gefeuert. Die interne Realisierung ist in diesem Fall nicht von Bedeutung, muss jedoch bei der formalen Modellierung in NCES berücksichtigt werden. Dies wird mit einer Zeitverzögerung implementiert. Für die Realisierung dieses Funktionsblockes ergibt sich das Modul in NCES Namens **simpleFB** (siehe Abbildung 3.16b), mit dem internen Verhalten in Abbildung 3.16c.

Für das Verhalten des Moduls betrachte man Abbildung 3.16b und 3.16c. Das Modul **simpleFB** besteht aus jeweils zwei Ereignisein- und ausgängen und den schon bekannten Bedingungseingang **enable**. Die Abarbeitung des Moduls beginnt sobald die Eingangsbedingung **enable** aktiv ist und man gelangt vom Ausgangszustand **START** zur Aktion **Aktivierung FB**. Wenn nun ein Ereignis am Ereigniseingang **REQ** anliegt, wird auf **eine Zeit abwarten**

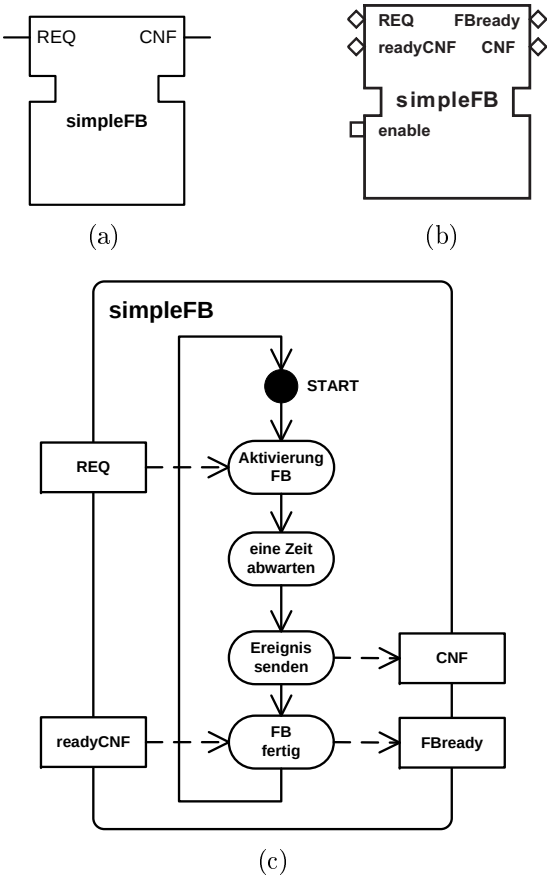


Abbildung 3.16: Einfacher Funktionsblock a) Standard IEC 61499 b) NCES Modul c) Verhalten

verzweigt. Das Ereignis, das dem Modul übergeben wurde, kommt direkt vom Ereignisausgang **outEVx** des Moduls **Thread_exe** in Abbildung 3.14. Mit diesem Ereignis kann der einfache Funktionsblock nun in der Aktion **eine Zeit abwarten** abgearbeitet werden. Nach der Abarbeitung wird über die Aktion **Ereignis senden** das Ausgangsereignis **CNF** generiert. Mit diesem Ausgangsereignis wird nachgeschaut, mit welchen Eingangsereignissen dieser Funktionsblock verbunden ist und stellt diese Eingangsereignisse der verbundenen Funktionsblöcke (siehe voriges (die Verzweigung) Kapitel) in das Modul **Thread_exe**. Über das Eingangsereignis **readyCNF** wird angezeigt, dass das Senden des CNFs Ereignisses fertig ist und mit dem Ausgangsereignis **FBready** wird noch dem Modul **Thread_exe** über das Eingangsereignis **endEVx** (siehe Abbildung 3.14) bekannt gegeben, dass das Modul **simpleFB**

mit der Abarbeitung fertig ist.

Zeit- Funktionsblock als formales Modell

Als nächstes gilt es Zeit- Funktionsblöcke als formale Modelle zu beschreiben, wobei in dieser Arbeit die Funktionsblöcke E_CYCLE und E_DELAY realisiert wurden. Um überhaupt Zeit handhaben zu können, stellt das Echtzeitbetriebssystem eCos im Zusammenhang mit der μCorns Laufzeitumgebung den External Event Timer (siehe Kapitel 3.1.4) zur Verfügung, der in gewissen Zeitabständen einen entsprechenden Interrupt auslöst.

Für die Realisierung der formalen Modelle der Zeit- Funktionsblöcke wird das formale Modell des External Event Timers in Kapitel 3.3.4 mit diesen in Verbindung gebracht. Es ergibt sich ein definiertes Interface zwischen einem Zeit- Funktionsblock und dem External Event Timer, welches für jeden Zeit- Funktionsblock ident ist. Jedes formale Modul eines Zeit- Funktionsblocks (z. B. E_CYCLE) besitzt das Eingangsereignis **TIinvoke** und die Ausgangsereignisse **startTI**, **stopTI** und **FBreadyTI** (siehe Abbildung 3.17b). Die Ausgangsereignisse **startTI** und **stopTI** sind mit den Eingangsereignissen des External Event Timer Moduls **TIxstart** und **TIxstop** (siehe Abbildung 3.11a) in Verbindung gesetzt und bilden zugleich das Interface eines Threads mit Zeit- Funktionsblöcken. Hingegen steht das Eingangsereignis **TIinvoke** und das Ausgangsereignis **FBreadyTI** in Verbindung mit dem Modul Thread_exe. Die Aktivierung des Eingangsereignisses **TIinvoke** über das Ausgangsereignis **outEVx** des Moduls Thread_exe bewirkt eine Generierung des Ausgangsereignisses **EO**. **TIinvoke** bewirkt in Verbindung mit dem Modul External Event Timer eine fortlaufende Aktivierung des Moduls. Mit dem Ausgangsereignissen **FBreadyTI** und **FBready** wird dem Modul Thread_exe über das Eingangsereignis **endEVx** das Ende der Abarbeitung eines Ereignisses bekannt gegeben. Mit diesem Ausgangsereignis wird das Ereignis weitergeleitet. Sobald das Ereignis in den Event Dispatcher erfolgreich hineingestellt wurde, wird dem Modul des Zeit- Funktionsblockes über **readyEO** die erfolgte Abarbeitung bekannt gegeben. Über die Eingangsereignisse **START** und **STOP** wird das Modul des Zeit- Funktionsblocks im Zusammenhang mit dem Ausgangsereignis **outEVx** des Moduls Thread_exe (siehe Abbildung 3.14) aktiviert bzw. deaktiviert. Nach dieser Beschreibung des Moduls eines Zeit-Funktionsblock soll nun noch mit der nachfolgenden Tabelle 3.3 ein Überblick gegeben werden, mit welchen Elementen das Modul in Verbindung steht.

Nun soll auf das Verhalten dieser zwei Funktionsblöcke eingegangen werden. Da die Funktionsweise der beiden sehr ähnlich ist, wird hier nur der E_CYCLE genauer beschrieben. Abbildung 3.17a und 3.17b zeigen den E_CYCLE im

Interface zum	**Bezeichnung**
External Event Timer	
startTI	TimerInterface wird über *TIxstart* gestartet
stopTI	TimerInterface wird über *TIxstop* gestoppt
Thread_exe	
FBready	Ereignis abgearbeitet
FBreadyTI	Ereignis von External Event Timer abgearbeitet
Funktionsblocknetzwerk	
EO	senden von Ausgangsereignis
Interface vom	**Bezeichnung**
External Event Timer	
TIinvoke	Aufruf mit Thread_exe
Thread_exe	
START	Aktivierung
STOP	Deaktivierung
Funktionsblocknetzwerk	
readyEO	EO Ausgangsereignis fertig

Tabelle 3.3: Interface das Modul eines Zeit- Funktionsblocks

Standard IEC 61499 und das entsprechende formale Modul in NCES dazu. Über den Dateneingang DT des E_CYCLE in Abbildung 3.17a wird eine bestimmte Zeit angegeben. Nach Anliegen eines Ereignisses an dem Ereigniseingang START wird beim E_CYCLE entsprechend dieser Zeit (DT) ein Ausgangsereignis EO generiert, bis mit STOP unterbrochen wird.

Die Funktionsweise des formalen Moduls für den E_CYCLE soll anhand des Aktivitätsdiagramms in Abbildung 3.17c erläutert werden. Über die Eingangsereignisse **START** oder **STOP** wird der E_CYCLE entweder durch **FB aktiv** aktiviert oder durch **FB inaktiv** deaktiviert. Nach einer **Aktivierung** wird der entsprechende TimerInterface (TIx) (siehe Kapitel 3.3.4), das sich im Modul External Event Timer befindet, über das Ausgangsereignis **startTI** gestartet und zugleich das Ausgangsereignis **FBready** erzeugt. Bei einer **Deaktivierung** durch **STOP**, wenn der Funktionsblock noch nicht aktiv ist, wird auf **START** zurückgekehrt und das Ereignis **FBready** generiert.

Wird eine erfolgreiche Aktivierung angenommen, dann wird die Aktion **Ti-**

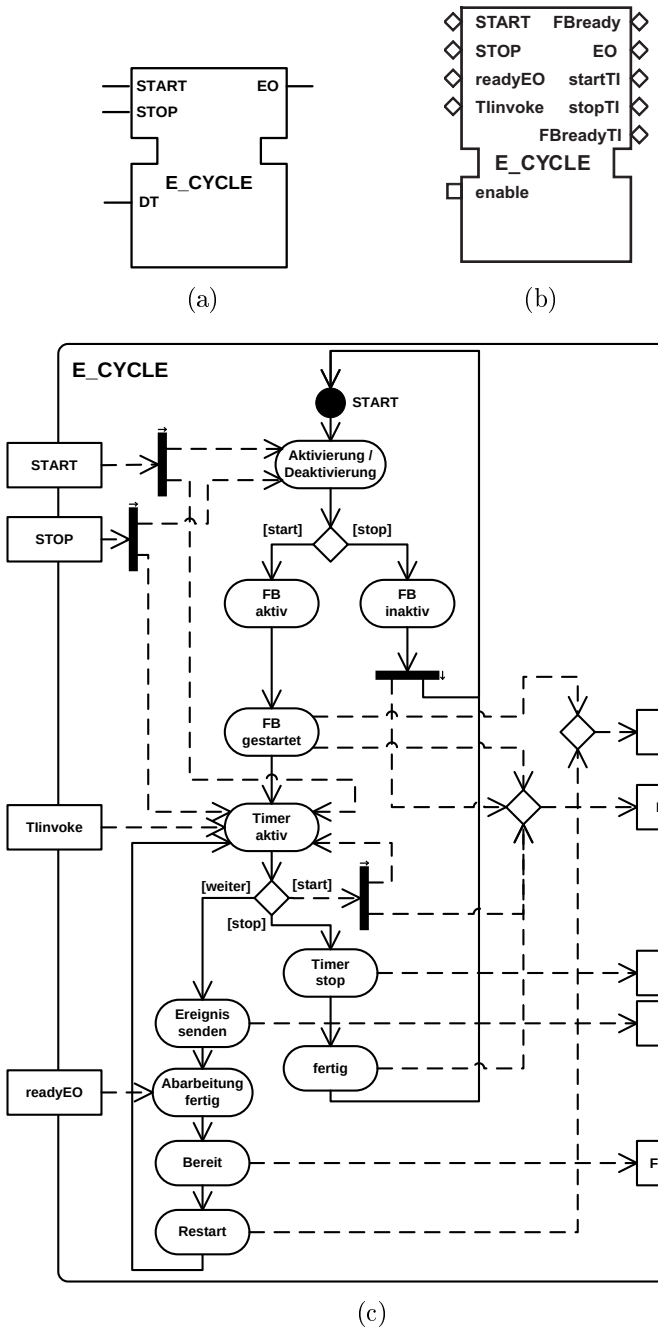


Abbildung 3.17: E_CYCLE Funktionsblock a) Standard IEC 61499 b) NCES Modul c) Verhalten

mer aktiv erreicht. Wenn die Zeit im Modul External Event Timer abgelaufen ist, wird mit **TIinvoke** über **weiter** auf **Ereignis senden** verzweigt. Anschließend wird ein Ausgangsereignis **EO** generiert, mit der Bestätigung **readyEO** ist die Abarbeitung fertig und mit der Aktion **Bereit** wird das Ausgangsereignis **FBreadyTI** generiert. Dieses wird dem External Event Timer übergeben. Anschließend wird von der Aktion **Restart** der External Event Timer mit dem Ausgangsereignis **startTI** wieder gestartet und auf **Timer aktiv** retourniert.

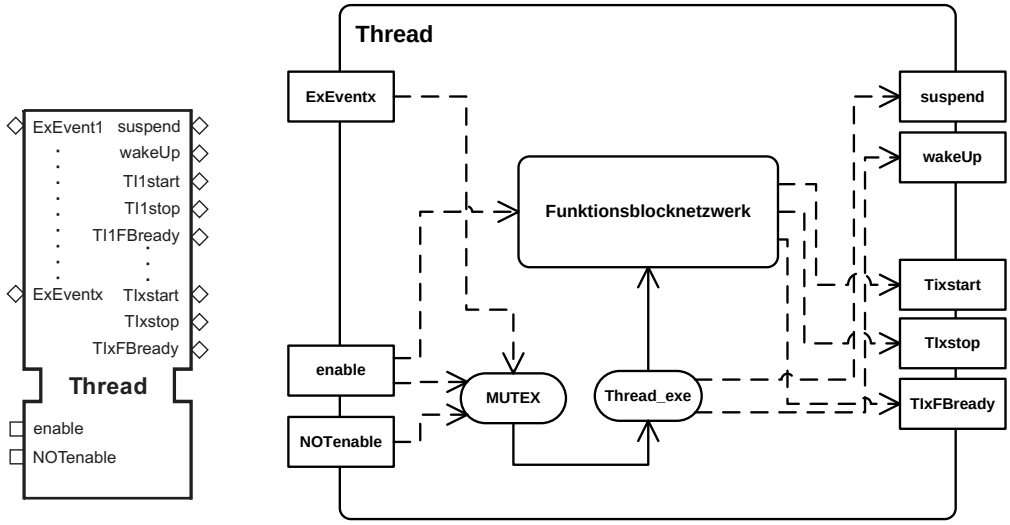
Die Aktion **Timer aktiv** kann auch durch die Eingangsereignisse **START** und **STOP** aktiviert werden. Erhält das Modul ein weiteres **START** Eingangsereignis, dann wird mit **start** auf die Aktion **Timer aktiv** retourniert und das Ausgangsereignis **FBready** generiert. Mit **STOP** wird der aktive Funktionsblock über **stop** mit **Timer stop** deaktiviert und der entsprechende TimerInterface im External Event Timer durch das Ereignis **stopTI** gestoppt. Weiters wird das Ausgangsereignis **FBready** gesendet und auf **START** verzweigt.

Das Verhalten des E_DELAY Funktionsblocks ähnelt sehr stark dem des E_CYCLE. Es gibt zwei Unterschiede die anhand von Abblidung 3.17 nachvollzogen werden können. Die Unterschiede sind jene, dass nach einer Generierung des Ausgangsereignisses **EO** nicht auf **Timer aktiv** verzweigt wird, sondern direkt auf **START** und der TimerInterface wird über **startTI** nicht erneut gestartet.

3.4.6 Thread als formales Modell

Bei diesem Modell werden die zuvor angeführten Module Laufzeitumgebung, Funktionsblöcke und Mutex (siehe Kapitel 3.3.3) zusammen in Verbindung gebracht. Aus dieser Gesamtheit kann ein beliebiger Thread für die μCrons Laufzeitumgebung realisiert werden. Abbildung 3.18 stellt einen solchen Thread dar. Das NCES Modul eines solchen Threads (Abbildung 3.18a) besteht aus einer Anzahl an Eingangsereignissen **ExEventx** (x steht für die Nummerierung eines Ereignis). Über diese können externe Ereignisse (z. B. External Event Timer, External Event Manger, ...) in den Thread gelangen. Mit den Ausgangsereignissen **suspend** und **wakeUp** wird im Zusammenhang mit einem Scheduler (Bitmap oder MLQ siehe Kapitel 3.3.1 und 3.3.2) der Status des Threads übergeben. Mit **suspend** teilt der Thread dem Scheduler mit, dass er nicht mehr abgearbeitet werden möchte. Über das Ausgangsereignis **wakeUp** hingegen wird dem Scheduler bekanntgegeben, dass der Thread bereit ist für die Abarbeitung. Mit der Ausgangsereignis- Kombination **TIxstart**, **TIxstop** und **TIxFBready** gibt der Thread nach außen hin dem External Event Timer (siehe Kapitel 3.3.4) bekannt, dass Zeit- Funktionsblöcke in einem Thread

abgearbeitet werden möchten. Schließlich wird mit den Eingangsbedingungen **enable** und **NOTenable** der Thread über dem Scheduler (Bitmap oder MLQ) aktiviert bzw. deaktiviert.



(a) (b)

Abbildung 3.18: Implementierung eines Threads a) NCES Modul b) Verhalten

Das interne Verhalten eines solchen Threads zeigt Abbildung 3.18b als Aktivitätsdiagramm. Das Modell eines solchen Threads ist beliebig im Sinne des Funktionsblocknetzwerkes. Die NCES Module **MUTEX** (siehe Abbildung 3.10) und **Thread_exe** (siehe Abbildung 3.14) sind im Modell vom **Thread** immer enthalten. Mit dem formalen Modell des **MUTEX** werden über die Eingangsereignisse **ExEventx** externe Ereignisse „gekapselt“ und dem formalen Modell **Thread_exe** weitergeleitet. Weiters wird mit den Eingangsbedingungen **enable** und **NOTenable** über den **MUTEX** das formale Modell **Thread_exe** angesteuert. Das formale Modell **Thread_exe** gibt den Status des Threads nach außen hin über die Ausgangsereignisse **suspend** und **wakeUp** bekannt und arbeitet das Funktionsblocknetzwerk mit Ereignissen ab, die im Event Dispatcher enthalten sind. Das Funktionsblocknetzwerk kann aus beliebigen Funktionsblöcken bestehen, die miteinander vernetzt sind. Enthält ein solches Funktionsblocknetzwerk Zeit- Funktionsblöcke, dann werden diese nach außen hin mit den Ausgangsereignissen **TIxstart**, **TIxstop** und **TIxFBready** (x steht für den entsprechenden Zeit- FB) über den External Event Timer angesteuert.

3.5 Zusammenspiel der Modelle

In diesem Abschnitt wird das Zusammenspiel der bis jetzt angeführten formalen Modelle betrachtet. Die Abbildung 3.19 dient zum näheren Verständnis des Zusammenspiels der Modelle vom Echtzeitbetriebssystem eCos und der μCorns Laufzeitumgebung. Anstatt der Darstellung der formalen Modellen wurden in der Abbildung 3.19 repräsentative Konstrukte herangezogen, um das Zusammenwirken der Modelle besser veranschaulichen zu können. Die Abbildung 3.19 ist in zwei Teilen aufgeteilt: links das Echtzeitbetriebssystem eCos mit einem Scheduler und External Event Timer, rechts die μCorns Laufzeitumgebung mit einer Anzahl an Threads.

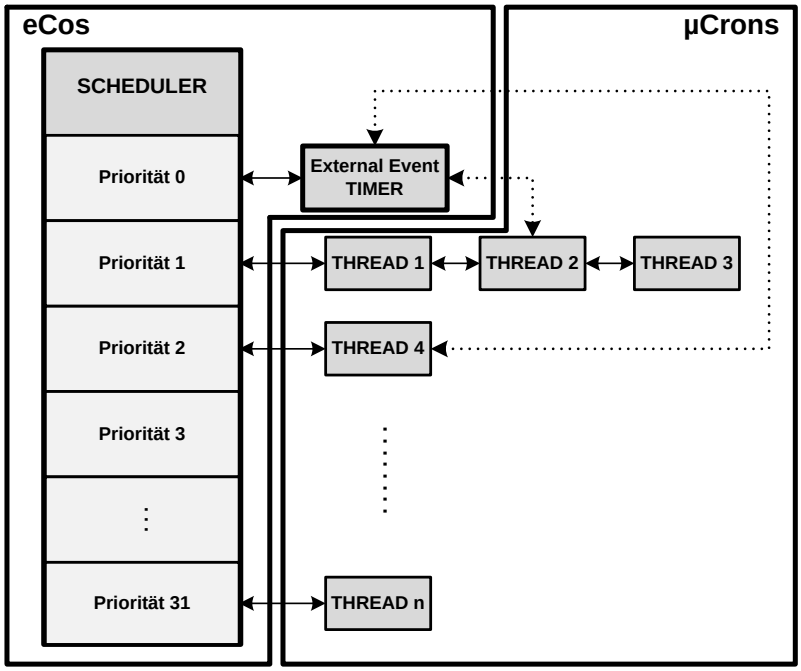


Abbildung 3.19: Zusammenspiel des Echtzeitbetriebssystem eCos und Laufzeitumgebung μCrons

Betrachtet man nun Abbildung 3.19 genauer, so besteht das Szenario aus einem eCos Scheduler, der bis zu 32 Prioriäten verfügen kann, wobei jede Priorität entweder einem Bitmap oder einem MLQ Scheduler entspricht. Mit den Prioritäten des Schedulers werden die unterschiedlichen Threads angesteuert und abgearbeitet. Die Priorität 0 des Schedulers in Abbildung 3.19 enspricht der höchsten Priorität und einem Bitmap Scheduler. Der External Event Ti-

mer ist auf dieser Priorität angeführt, da er für die ständige Abarbeitung der Zeit- Funktionsblöcke in den unterschiedlichen Funktionsblocknetzwerken in den Threads verantwortlich ist. Wie schon im Kapitel 3.1.4 erwähnt, handelt es sich beim External Event Timer um eine Call-Back Funktion, die in dieser Arbeit als höchste Priorität modelliert wurde. An der Priorität 1 befindet sich ein MLQ Scheduler, der 3 Threads ansteuert und Priorität 2 ist wieder ein Bitmap Scheduler mit einem Thread. Ein Thread selbst kann ein Funktionsblocknetzwerke mit jeweils einem Event Chain Executor (Event Dispatcher, siehe Kapitel 3.2.2) enthalten. Besitzt ein Thread ein Funktionsblocknetzwerk mit Zeit- Funktionsblöcken, dann kommt der External Event Timer auf der Priorität 0 zum Einsatz. In diesem Szenario enthalten Thread 2 und Thread 4 Zeit-Funktionsblöcke, da diese mit dem External Event Timer in Verbindung gebracht sind. Sobald ein Zeit- Funktionsblock in einem dieser Threads aktiviert wurde, gibt dieser dem External Event Timer bekannt, dass er abgearbeitet werden möchte und wird vom External Event Timer angesteuert.

3.6 Ausführungszeiten in formalen Modellen

Die Implementierung von Ausführungszeiten in den bis jetzt beschriebenen formalen Modellen bewirkt ein reales Verhalten der Modelle. Anhand des Echtzeitbetriebssystems eCos und der μCrons Laufzeitumgebung soll nun gezeigt werden, wo Ausführungszeiten in den Modellen einfließen.

Die Zeiten in den Modellen werden unterteilt in *unterbrechbare* und *nicht unterbrechbare Ausführungszeiten*. Die Modellierung dieser Zeiten in NCES zeigen die Modelle in Abbildung 3.20. Betrachtet man das NCES Modell in Abbildung 3.20a, so stellt dies eine *unterbrechbare Ausführungszeit* dar. Die Ausführungszeit ist so realisiert worden, dass, sobald die Markierung die Stelle **p2** erreicht hat, die Zeitverzögerung aktiviert wird. Die Zeitverzögerung für unterbrechbare Ausführungszeit wurde als Zähler implementiert. Das Konstrukt des Zählers besteht aus der Stelle **p2** und der Transition **t2**. Die Transition **t2** sorgt dafür, dass nach jeder Zeiteinheit [1;-2] (-2 entspricht ∞) die Markierung in der Stelle **p2** um eins erhöht wird. Für die Realisierung dieses Verhaltens wird nach einer Zeiteinheit eine Markierung aus der Stelle **p2** entnommen und zugleich 2 Markierungen von **t2** zu **p2** hinzugefügt. Die Transition **t3** wird erst dann aktiv, wenn in der Stelle **p2** 300 Markierungen vorhanden sind.

Die Bedingung fürs Weiterschalten wird durch das Gewicht 300 des Bogens von der Stelle **p2** zur Transition **t3** definiert, wobei das Gewicht 300 ein Zeitverzögerung von $(300 - 1) = 299$ entspricht. Die Zeitverzögerung von 299 ist damit zu verstehen, da bereits vor dem Anfang der Zeitverzögerung eine Mar-

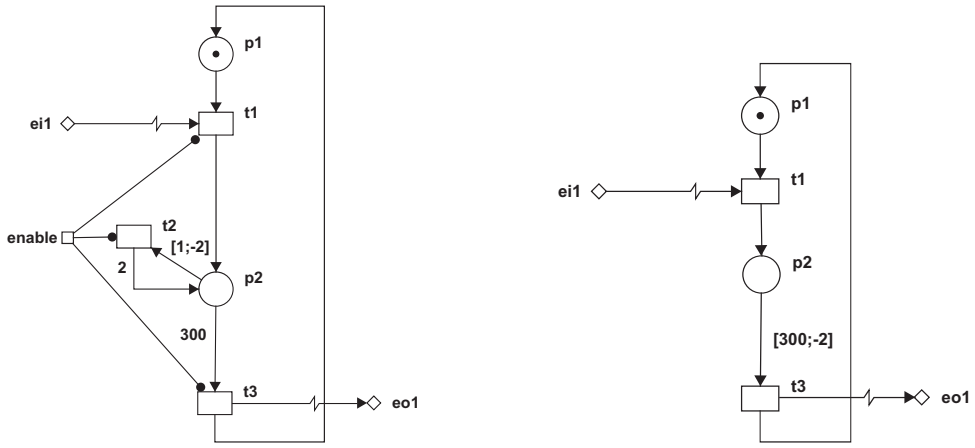


Abbildung 3.20: NCES Modelle für a) unterbrechbare Ausführungszeit b) nicht unterbrechbare Ausführungszeit

kierung in der Stelle **p2** enthalten ist. Mit der Eingangsbedingung **enable**, die auf allen Transitionen verbunden ist, kann die Zeitverzögerung unterbrochen werden. Sobald die Eingangsbedingung **enable** nicht aktiv ist, wird die Zeitverzögerung unterbrochen und bei einer erneuten Aktivierung wird mit der Zeitverzögerung fortgesetzt.

Für *nicht unterbrechbare Ausführungszeiten* wird das NCES Modell in Abbildung 3.20b verwendet. In diesem Fall wird die Zeitverzögerung aktiviert, sobald eine Markierung in der Stelle **p2** vorhanden ist. Nach einer Zeitverzögerung von 300 Zeiteinheiten wird die Transition **t3** angesprochen und das Ausgangsereignis **eo1** generiert.

3.6.1 Zeiten in den Modellen von eCos

Das Echtzeitbetriebssystem eCos besitzt eine Anzahl an unterschiedlichen Ausführungszeiten. In den formalen Modellen von eCos (Kapitel 3.3) werden jedoch nur einige Zeiten implementiert, wobei diese Zeiten aus [Fer07] entnommen wurden.

Zeiten, die nun in den formalen Modellen des Echtzeitbetriebssystems eCos einfließen, betreffen den Bitmap und Multi Level Queue Scheduler (Kapitel 3.3.1 und 3.3.2). Sobald ein Wechsel von einem *Thread x* zu einem anderen *Thread y* vorkommt, wird eine gewisse Zeit verbraucht. Diese Zeit wird mit Kontextwechsel bezeichnet und in den Modellen der Scheduler implementiert. Weiters wird bei einer Suspendierung und Wiederaufnahme eines Threads je-

weils eine Zeit verbraucht. Auch diese wurden in den formalen Modellen der Scheduler eingefügt. Für den MLQ Scheduler wurde weiters noch das Timeslicing (siehe Kapitel 3.1.2) implementiert. Alle diese Zeiten wurden mit dem NCES Modell der nicht unterbrechbaren Ausführungszeit realisiert. Diese Realisierung wurde deshalb gewählt, da der Scheduler ein Betriebssystem abbildet, das im Hintergrund immer aktiv ist.

Auch der External Event Timer besitzt eine Anzahl an Zeiten. Die wesentliche Zeitimplementierung, die im External Event Timer (siehe Kapitel 3.3.4) vorkommt, ist jene vom internen Timer in Abbildung 3.11b. Nach der Aktivierung des External Event Timers über einen Zeit-FB, generiert dieser in gewissen Zeitabständen ein Ausgangsereignis. Mit diesem Ausgangsereignis wird dem Scheduler bekannt gegeben, dass der External Event Timer bereit für die Abarbeitung ist. Diese Zeit wurde mit dem Modell der nicht unterbrechbaren Ausführungszeit implementiert, da der External Event Timer auf der höchsten Priorität des Schedulers liegt.

Für die formalen Modelle Semaphore und Mutex (siehe Kapitel 3.3.3) wurden keine Zeiten implementiert.

3.6.2 Zeiten im Modell der μCrons Laufzeitumgebung

Auch bei der μCrons Laufzeitumgebung gibt es eine Anzahl an unterschiedlichen Zeiten. Unter Zuhilfenahme der Messungen in [Bru06] wurden die wichtigsten Zeiten verwendet und in den formalen Modellen der μCrons Laufzeitumgebung implementiert. Für die Messungen der μCrons Laufzeitumgebung in [Bru06] wurde das Echtzeitbetriebssystem ThreadX verwendet. Die Messungen aus [Bru06] werden somit als Referenz verwendet und können in den formalen Modellen der Laufzeitumgebung im Zusammmenhang mit eCos als Anhaltspunkte verwendet werden. Im Modell des Event Dispatchers (siehe Kapitel 3.4.1) wurden jeweils Zeiten für das Hineinlegen und Herausholen von Ereignisse implementiert. Beim Hineinlegen eines Ereignisses in dem Event Dispatcher gibt es zwei charakteristische Zeiten: eine für das Hineinstellen eines Ereignisses, wenn die Queue noch nicht voll ist und eine, wenn die Queue voll ist. Beim Herausholen eines Ereignisses aus dem Event Dispatcher vergeht eine Zeit für das Herausnehmen eines Ereignisses aus der Queue und eine für das Herausnehmen, wenn die Queue leer ist.

Beim formalen Modell der Ereignisverzweigung im Kapitel 3.4.4 wurde eine Zeitverzögerung aus [Bru06] implementiert.

Auch die im Kapitel 3.4.5 angeführten formalen Modelle für Funktionsblöcke enthalten Zeiten. Das Modell **simpleFB** in Abbildung 3.16 enthält eine Interne Verarbeitungszeit, die aus [Bru06] in das Modell eingefügt wurde. Nach einem

Ereignis an **REQ** wird der **simpleFB** aktiviert und es wird eine bestimmte Zeit abgearbeitet. Bei den Zeit- Funktionsblöcken z.B. **E_CYCLE** treten im ganzen fünf Zeiten auf. Sobald ein Ereignis am Ereigniseingang **START** auftritt, vergehen, in Abhängigkeit ob der FB aktiv oder inaktiv ist, zwei Zeiten. Das selbe gilt auch sobald am Ereigniseingang **STOP** ein Ereignis anliegt. Für die Deaktivierung des Zeit- FBs am Ende der Abarbeitung vergeht auch eine Zeit.

Die Zeitverzögerungen in den formalen Modellen der μCrons Laufzeitumgebung wurden mit dem NCES Modell der unterbrechbaren Ausführungszeit realisiert. Grund dafür ist, dass diese Modelle z. B. ein Thread im Zusammenhang mit einem Scheduler von einem höherprioren Thread unterbrochen werden kann. Sobald der unterbrochene Thread wieder aktiviert wird, kann vom letzten Stand der Zeitverzögerung fortgesetzt werden.

3.7 Zusammenfassung

In den Kapiteln 3.1 und 3.2 wurde ein näherer Überblick über das Echtzeitbetriebssystem eCos und die μCrons Laufzeitumgebung gegeben. Mit diesen Grundlagen wurden die entsprechenden formalen Modelle in den Kapiteln 3.3 und 3.4 angeführt und beschrieben. Mit diesen Modellen hat man nun die Möglichkeit, verschiedene Anwendungen zu konstruieren, die ein beliebiges Verhalten aufweisen. So wurde im Kapitel 3.5 ein exemplarisches Beispiel angeführt, das einen Überblick über das Zusammenspiel der wichtigsten Elemente vom Echtzeitbetriebssystem eCos und der μCorns Laufzeitumgebung wiedergibt. Im letzten Abschnitt wurden noch die nötigen Ausführungszeiten, die in den formalen Modellen implementiert werden, angeführt. Damit kann man die reale Abarbeitung mit physikalischer Ausführungszeit bei vorhandenen Messungen in die formalen Modellen einfließen lassen. Die formalen Modelle und die Ausführungszeiten in Kapitel 3 bilden die Grundlage für die im Kapitel 4 angeführte Realisierung einer exemplarischen Applikation.

4 Formale Modellierung mit NCES

Das Kapitel 4 beinhaltet den parktischen Teil der Diplomarbeit. Im ersten Teil werden der Anwendungsbereich, Ziele der Arbeit und die Zeitimplementierungen in den formalen Modellen erläutert. Danach werden kurz die verwendeten Programme zur Realisierung und Verifikation der NCES Modelle angeführt. Weiters werden einige formale Modelle näher betrachtet, wobei ein besonderes Augenmerk auf die interne Realisierung gelegt wird. Anschließend wird mit den formalen Modellen in Kapitel 3 eine Applikationen auf dessen Verhalten verifiziert.

4.1 Anwendungsbereich

Die folgenden Abschnitte geben einen Einblick über den Nutzen der formalen Modelle in Kapitel 3 und in wieweit Einschränkungen für die praktische Umsetzung vorgenommen wurden. Weiters werden noch die Messungen von eCos und μCrons angeführt und den äquivalenten formalen Modellen gegenübergestellt.

4.1.1 Nutzen der formalen Modelle

Mit den formalen Modellen des Echtzeitbetriebssystems eCos in Kapitel 3.3 und der μCrons Laufzeitumgebung in Kapitel 3.4 hat man nun die Möglichkeit, unterschiedliche Szenarien von Threads und Scheduler zu realisieren. Ziel dieser Arbeit war es somit die formalen Modelle so zu realisieren, dass man auch größere Applikation (Funktionsblocknetzwerk) damit verifizieren kann. Geht man von Abbildung 3.19 in Kapitel 3.5 aus, dann soll zuerst der eCos Scheduler und dann die μCrons Laufzeitumgebung betrachtet werden.

Wie bereits im Kapitel 3.1.2 angeführt, kann man in eCos einen Scheduler, aus einer Kombination von Bitmap und MLQ, bis zu 32 Prioritäten realisieren. In dieser Arbeit wurden formalen Modelle des Bitmap und MLQ Scheduler bis zu 5 Prioritäten erzeugt. Mit diesen 5 Prioritäten kann man einen beliebigen

eCos Scheduler aufbauen und darauf Applikationen (sind in Threads enthalten, siehe Abbildung 3.19) testen. Beim MLQ Scheduler, der mehrere Threads pro Priorität enthält, wurden formalen Modelle erzeugt, bei denen pro Priorität zwei Threads verwendet werden können. Zusätzlich wurde an der Priorität 4 des MLQ Schedulers eine Ausführung von drei Threads erstellt. Die formalen Modelle des Bitmap und MLQ Schedulers wurden so erstellt, dass die Anzahl der Prioritäten und Anzahl der Threads pro Priorität erweiterbar sind. Weiters wurde der Mutex (siehe Kapitel 3.3.3) für drei externe Ereignisquellen und der Timer (siehe Kapitel 3.3.4) für die Abarbeitung von sechs Zeit- Funktionsblöcken realisiert. Diese formalen Modelle kann man auch so wie die Scheduler einfach erweitern.

Für die Realisierung der Threads auf den unterschiedlichen Prioritäten des eCos Schedulers (siehe Abbildung 3.19) werden die formalen Modelle aus Kapitel 3.4 herangezogen. Wie bereits in Kapitel 3.2.2 erläutert, besteht ein Thread aus einem Funktionsblocknetzwerk (Applikation) und einem Event Chain Executor (Event Dispatcher). Das formale Modell des Event Dispatchers (siehe Kapitel 3.4.1) wurde für 30 Eingangsereignisse und 10 Positionen in der Event-Queue realisiert. Im Event Dispatcher können somit bis zu 30 unterschiedliche Ereignisse in der Event-Queue abgelegt werden, jedoch nur bis einer Anzahl von 10 Ereignissen. Mit einem solchen Event Dispatcher hat man nun die Möglichkeit, größere Funktionsblocknetzwerke auszuführen und zu verifizieren. Um ein Funktionsblocknetzwerk zu erstellen, werden die formalen Modelle der Funktionsblöcke aus Kapitel 3.4.5 und die Verzweigungen das formale Modell aus Kapitel 3.4.4 herangezogen. Für die Funktionsblöcke stehen sowohl der E_CYCLE und E_DELAY zur Verfügung, auch ein Funktionsblock, der mit einer bestimmten Abarbeitungszeit versehen wurde. Für die Ereignisverzweigung wurde ein formales Modell für zwei Ausgangsereignisse erzeugt, das auf mehrere Ausgangsereignisse erweitert werden kann.

4.1.2 Implementierung der Messungen von eCos und μCrons

Im Kapitel 3.6 wurden bereits die Zeiten angeführt, die in die formalen Modellen dieser Arbeit einfließen. Die folgenden Tabellen 4.1 und 4.2 sollen einen Überblick über die implementierten Zeiten in den formalen Modellen des Echtzeitbetriebssystems eCos und der μCrons Laufzeitumgebung wiedergeben. Betrachtet man die Tabellen, so werden in der ersten Spalte die formalen Modelle angegeben in denen eine Zeit implementiert wurde. Für die Überführung der realen Zeiten in den formalen Modellen wurde die folgende Definition gewählt:

$1\mu s \widehat{=} 1t$ (Zeiteinheit in TNCES).

Die zweite und dritte Spalte enthalten den Namen der unterschiedlichen Zeiten und die entsprechenden Werte der Messungen. Die Wertangaben der Messungen in Tabelle 4.1 wurden aus [Fer07] und jene in Tabelle 4.2 aus [Bru06] entnommen. Zuletzt sind in der vierten Spalte die eigentlichen Zeiten angeführt, die in den jeweiligen formalen Modellen implementiert wurden. Diese folgen aus der zuvor angeführten Definition für die Überführung der realen Zeit in den formalen Modellen.

formales Modell	**Zeit**	**Messung [μs]**	**Zeit im Modell [t]**
Bitmap und MLQ Scheduler	Kontextwechsel	82	82
MLQ Scheduler	Timeslicing	5000	5000
IDLE_Prio5	Thread Unterbrechung	10,1	10
	Thread Wiederaufnahme	13,1	13
Timer	Aktivierung	1000	1000
	DT (Duration Time)	variabel	variabel

Tabelle 4.1: Zeiten für die formalen Modelle vom Echtzeitbetriebssystem eCos

4.2 Programme für NCES

Für die Erstellung der formalen Modelle wurden Net Condition Event Systems (NCES) (siehe Kapitel 2.2.3) verwendet. Die Realisierung solcher NCES Modelle wurde mittels (T)NCES- Editor durchgeführt, der von der Martin-Luther- Universität Halle-Wittenberg Lehrstuhl Automatisierungstechnik zur Verfügung gestellt wurde. Für die Verifikation und Veranschaulichung des Verhaltens der formalen Modelle in NCES wurde das Programm ViVe (Visual Verifier) verwendet. Dieses Programm wurde von Valeriy Vyatkin [Vyaa] an der University of Auckland Department of Electrical and Computer Engineering entwickelt.

4.2.1 (T)NCES- Editor

Mit dem (T)NCES- Editor, der in Abbildung 4.1 dargestellt ist, können NCES Modelle (siehe Kapitel 2.2.3) mit Zeitimplementierung erstellt werden. Als

formales Modell	Zeit	Messung [μs]	Zeit im Modell [t]
Event Dispatcher	Ereignis hineinstellen (Queue voll)	8	8
	Ereignis hineinstellen (Queue nicht voll)	27,4	27
	Ereignis herausnehmen (Queue mit Ereignis)	29,5	29
	Ereignis herausnehmen (Queue ist leer)	9,8	10
Ereignis-verzweigung	Wartezeit Verzweigung	29,8	30
einfacher FB	Dauer FB	143,7	144
Zeit FB	Aktivierung (FB aktiv)	-	7
	Aktivierung (FB inaktiv)	-	13
	Deaktivierung (FB aktiv)	-	9
	Deaktivierung (FB inaktiv)	-	7
	Deaktivierung FB	-	5

Tabelle 4.2: Zeiten für die formalen Modelle der μCrons Laufzeitumgebung

Exemplarisches NCES Modell dient das in Abbildung 4.1 dargestellte Beispiel eines Flip Flops (keine Zeitimplementierung). In der Umrahmung 1 in Abbildung 4.1 hat man die Möglichkeit NCES Module zu erstellen oder zu bearbeiten und als XML[1] zu speichern. In der Umrahmung 3 ist das eigentliche NCES Modell enthalten. Dieses kann aus Bedingungsein- und ausgänge (Condition in- output) sowie Ereignisein- und ausgänge (Event in- output) bestehen, die in der Umrahmung 1 ausgewählt werden können und in der Umrahmung 3 erzeugt werden. Weiters können über der Umrahmung 1 Transitionen (transition), Stellen (place) und Verbindungen (arc) eingefügt werden. Verbindet

[1] Extensible Markup Language (engl. für „erweiterbare Auszeichnungssprache“), ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien.

man z. B. Transitionen mit Stellen über Verbindungen (arc), dann generiert das Programm die entsprechende richtige Verbindung. Weiters kann man noch die erzeugte Verbindung ändern z. B. von einer normalen Verbindung (Ordinary arc) zu einem Bedingungsbogen (Condition arc). Ein NCES Modul kann auch aus mehreren NCES Modulen bestehen (siehe Abbildung 4.2), die über **fbType** (Umrahmung 1) in der Umrahmung 3 eingefügt und mit anderen Modulen verbunden werden können.

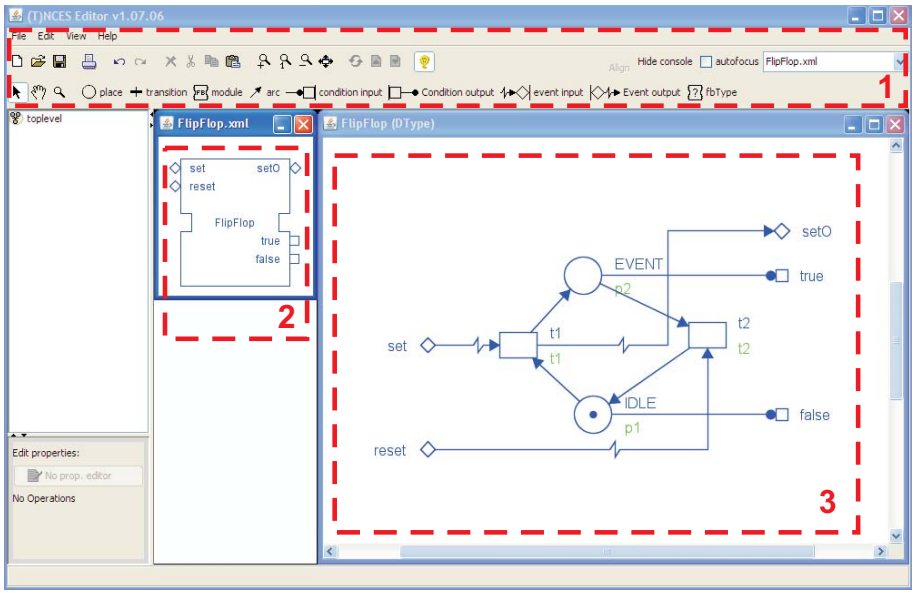


Abbildung 4.1: Flip Flop realisiert mit dem TNCES- Editor

Die Umrahmung 2 zeigt das NCES Modul nach außen hin. Das Beispiel in Abbildung 4.1 besitzt zwei Ereigniseingänge **set** und **reset** und einen Ereignisausgang **setO**. Weiters gibt es noch die Bedingungsausgänge **true** und **false**.

4.2.2 Visual Verifier - ViVe

Mit dem Programm ViVe, das auf einem früheren Prototypen Namens iMatch [Vyaa] von Valeriy Vyatkin aufgebaut wurde, kann man die Verifikation von NCES Modulen durchführen. Abbildung 4.2 zeigt für das NCES Modul **FLIP FLOP** (siehe Abbildung 4.1) ein Testbeispiel, wo mit einer Eingangsbeschaltung das Verhalten des Moduls überprüft wird. Die Eingangsbeschaltung besteht aus drei NCES Modulen, wobei das Modul **EventSequence** in der Umrahmung 2 in Abbildung 4.2 alle 5 Zeiteinheiten ein Ausgangsereignis generiert

(beginnend mit **eo1**). In diesem NCES Modul wurde somit Zeit implementiert. Die NCES Module **SET** und **RESET** stellen nur eine *ODER*- Verküpfung für die Ereignisse, die vom Modul **EventSequence** generiert wurden, dar.

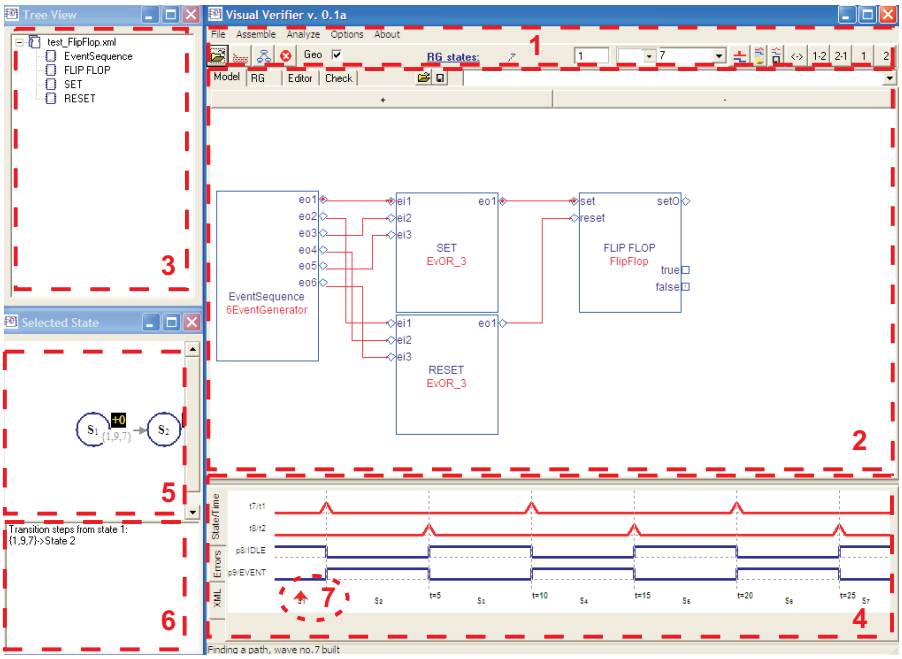


Abbildung 4.2: Flip Flop als Testbeispiel mit ViVe

In der Umrahmung 1 in Abbildung 4.2 können NCES Module geöffnet, zusammengefügt (assembliert) und deren Erreichbarkeitsgraph erstellt werden. Für die Erstellung des Erreichbarkeitsgraphen hat man die Möglichkeit, in der Umrahmung 2 über der Unterteilung *Check*, unterschiedliche Arten von Erreichbarkeitsgraphen zu erstellen. Das Programm unterstützt zwei Checker: einen *CTL checker*, der im Zusammenhang mit dem Programm SESA [Vyaa] arbeitet, und den *ViVe model checker*. Beide Checker unterstützen bei der Erstellung eines Erreichbarkeitsgraphen die Eigenschaft der *Firing rules*, wobei unterschiedliche Einstellungen ausgewählt werden können. Mit der Eigenschaft *Firing rule* wird die Ansteuerung der Ereignisse in einem NCES Modul definiert. Beim *CTL checker* kann man die Optionen *Single spontaneous* und *Maximal steps* auswählen. Der *ViVe model checker* bietet die Optionen *Single spontaneous*, *All combinations* und *Maximum set of spontaneous*. Weiters kann man noch mit dem *ViVe model checker* die Eigenschaft der *Greedy transitions* und *Maximum RG size* definieren. Mit der Option *Greedy transitions* wird definiert wie Transition aktiviert werden. Dabei kann man die Optionen *Fi-*

re together (gemeinsam feuern) und *In combination* (in Kombination feuern) auswählen. Mit der Eigenschaft *Maximum RG size* kann die maximale Größe eines Erreichbarkeitsgraphen angegeben werden.

In der Umrahmung 2 hat man weiters noch die Möglichkeit die Eingabe einer Spezifikationsabfrage zu tätigen. Im Zusammenhang mit den beiden Checker *CTL checker* und *ViVe model checker* können diese Abfragen kontrolliert werden. Der *CTL checker* ünterstützt im Zusammenhang mit dem Programm SESA CTL Abfragen (siehe Kapitel 2.2.1), der *ViVe model checker* hingegen kennt nur Logikabfragen (AND, OR, NOT). In der Umrahmung 1 wird weiters noch die Anzahl der Zustände des Erreichbarkeitsgraphen angezeigt und der zeitliche Ablauf des gesamten NCES Modells (im Falle von Abbildung 4.2 *test_FlipFlop.xml*) kann definiert werden und wird anschließend in der Umrahmung 4 angezeigt.

Die Umrahmung 2 zeigt im Detail in der Unterteilung *Model* das eigentliche NCES Modul *test_FlipFlop.xml*. Weiters ist unter der Unterteilung *RG* der Erreichbarkeitsgraph ersichtlich (wenn in der Umrahmung 1 das Feld *Geo* aktiviert wurde), *Editor* listet alle Stellen und Transitionen auf und über *Check*, wie schon zuvor erwähnt, können diverse Einstellungen über die Erstellung des Erreichbarkeitsgraphen definiert werden.

Mit der Struktur der einzelenen NCES Module in der Umrahmung 3 kann auf die interne Realisierung der einzelnen Module in der Umrahmung 2 Einsicht genommen werden.

Das Zeitliche Verhalten des NCES Modells ist in der Umrahmung 4 dargestellt. Dieses wurde über der Umrahmung 1 mit den Zuständen 1 bis 7 definiert und erstellt. Die Achsen des Zeitverhaltens der NCES Modelle bestehen aus einer Kombination Transition/Stelle und Zeit/Zustand. Für die Achse Transition/Stelle kann man mit *Options - View* (siehe Umrahmung 1) die gewünschten Transitionen und Stellen der NCES Modelle auswählen. Im Beispiel von Abbildung 4.2 wurden die interessanten Transitionen **t7** und **t8** und die Stellen **p8** und **p9** des Moduls **FLIP FLOP** ausgewählt. In der Umrahmung 4 ist der Zustand **S1** (siehe Umrahmung 7) mit einem Pfeil markiert. Im Zeitverlauf kann man an diesem Zustand erkennen, dass die Transition **t7** aktviert ist und die Stelle **p8** eine (oder mehrere) Markierungen in diesem Fall besitzt.

In der Umrahmung 5 wird der Zustand, der in der Umrahmung 4 markiert wurde, mit dem nachfolgenden Zustand angezeigt. Es wird somit ein Ausschnitt aus dem Erreichbarkeitsgraphen dargestellt. Weiters wird in geschwungenen Klammern angezeigt welche Transition aktiviert sind, wenn man von Zustand **S1** nach Zustand **S2** gelangt. Zusätzlich wird noch die Zeitverzögerung des jeweiligen Zustandes angezeigt. Umgekehrt kann man aus dem gesamten Erreichbarkeitsgraphen (Umrahmung 2 *RG*) einen interessanten Zu-

stand auswählen, der dann in der Umrahmung 5 angezeigt wird.

Die Umrahmung 6 gibt nochmals Auskunft über die Transitions- Schritte von einem Zustand zu dem nachfolgenden Zustand, die in der Umrahmung 5 enthalten sind.

4.3 NCES Modelle im Detail

Mit den nachfolgenden Kapiteln soll versucht werden, die interne Realisierung der formalen Modelle in NCES vom Echtzeitzeitbetriebssystem eCos und der μCrons Laufzeitumgebung zu verdeutlichen. In den folgenden Abbildungen ist die Realisierung einiger formalen Modelle aus Kapitel 3 als NCES Modelle angeführt. Die restlichen wichtigen NCES Modelle sind im Anhang A.2 enthalten und die gesamten NCES Modelle dieser Arbeit sind in der beiliegenden CD wiederzufinden.

4.3.1 NCES Modelle des Echtzeitbetriebssystem eCos

In diesem Abschnitt wird die Realisierung der beiden eCos Scheduler Bitmap und MLQ und der Synchronisations- Mechanismen Semaphore und Mutex in NCES dargestellt. Im Kapitel 3.1.2 wurd bereits die Theorie zu den beiden Schedulern angeführt und im Kapitel 3.3.1 und Kapitel 3.3.2 wurden die jeweiligen formalen Modelle mit Hilfe eines Aktivitätsdiagramms angeführt. Dasselbe gilt für den Semaphore und Mutex, wo die Theorie der beiden im Kapitel 3.1.3 und die formalen Modelle als Aktivitätsdiagramm im Kapitel 3.3.3 wiederzufinden sind.

NCES Modell Bitmap Scheduler

Für die Beschreibung des NCES Modells des Bitmap Schedulers, wird (wie bereits im Kapitel 3.3.1) die Priorität 1 herangezogen. Im Vergleich zu Abbildung 3.7 wird das Verhalten des formalen Modells über die eigentliche Realisierung in NCES erklärt. Abbildung 4.3 stellt nun den Bitmap Scheduler für die Priorität 1 dar. Das äußere Interface ist das selbe wie jenes von Abbildung 3.7a, jedoch findet man im Inneren das NCES Modell.

Für die Realisierung des Bitmap Schedulers wurde als Grundgedanke eine Schrittkette implementiert. Wie man in Abbildung 4.3 erkennen kann, wird an der Stelle **IDLE**, wo eine Marke enthalten ist, begonnen. Sobald ein Ereignis am Ereigniseingang **aktivate** anliegt, wandert die Markierung über die Transition **t1** zur Stelle **Prio_0?**. An dieser Stelle wird über die Bedingungseingänge **p0_wa** und **p0_sus** überprüft, ob die Priorität 0 aktiv oder nicht

aktiv ist. Ist die Priorität 0 nicht aktiv, wird mit der Schrittkette fortgesetzt. Möchte hingegen die Priorität 0 abgearbeitet werden, wandert die Markierung zurück zur Stelle **IDLE** und zur Stelle **p11**. Ab dieser Stelle **p11** wird nach einer Zeit 82 das Ausgangsereignis **next_p0** aktiviert, das den Scheduler an der Priorität 0 aktiviert. Die Zeitimplementierung von 82 entspricht in diesem Fall einem Kontextwechsel (siehe Kapitel 3.6.1 und Tabelle 4.1). Für die folgenden Prioritäten in dieser Schrittkette wird nach demselben Verfahren fortgesetzt.

Gelangt die Markierung bis zur Stelle **execute_TH**, wird der Thread an der Priorität ausgeführt. Die Bekanntgabe des Status der Priorität und die Aktivierung/Deaktivierung des Threads an der Priorität nach außen hin wurde über zwei NCES Flip Flops realisiert (siehe Abbildung 4.3 oben rechts). Die Stellenkombination **want_activ_TH** und **suspend_TH** geben nach außen hin Auskunft darüber, ob der Thread an der Priorität abgearbeitet werden möchte oder nicht. Dies wir über die Ereigniseingänge **th_sus** und **th_wactiv** beeinflusst. Dasselbe gilt für die Stellenkombination **run_TH** und **stop_TH**, die den Thread an der Priorität aktivieren bzw. deaktivieren. Weiter wird auf das Verhalten des Bitmap Schedulers nicht mehr eingegangen, da im Kapitel 3.3.1 der Ablauf bereits erklärt wurde.

Anzumerken sind noch die Bedingungseingänge **p2_wa** und **p2_sus**. Mit diesen beiden Eingängen, die von der Priorität 2 kommen, wird bestimmt, ob eine Zeit bei einem Prioritätenwechsel verbraucht wird oder nicht. Im Falle, dass die Priorität 2 abgearbeitet werden möchte, ist der Bedingungseingang **p2_wa** aktiv und nach einer Kontextwechselzeit von 82 wird das Ausgangsereignis **next_prio** generiert. Dieses Ausgangsereignis in der Priorität 1 ist mit dem Eingangsereignis **activate** der Priorität 2 verbunden. Falls die Priorität 2 nicht aktiv ist, wird über **p2_sus** die Priorität 2 aktiviert, jedoch ohne der Kontextwechselzeit. Bei einem Wechsel auf einer niederen Priorität wird über eine Verzweigung (siehe z. B. ab Transition **t4** in Abbildung 4.3) entweder ein Kontextwechselzeit verbraucht oder auch nicht.

NCES Modell MLQ Scheduler

Die Abbildung 4.4 stellt das NCES Modell eines MLQ Schedulers für die Priorität 1 dar. Als Vergleich dient die Abbildung 3.8 im Kapitel 3.3.2, wo das Verhalten des Schedulers mit dem Aktiviätsdiagramm beschrieben wurde. Im Unterschied zum NCES Modell des Bitmap Schedulers in Abbildung 4.3 besteht das Modell des MLQ Schedulers aus mehreren NCES Modulen. Die ausschlaggebenden NCES Module dieses Schedulers sind **Select_2TH_p1**, **execute_2TH_p1** und **Select_HP**. Diese Module werden in diesem Kapitel graphisch nicht angeführt, sind jedoch im Anhang A.2 enthalten. Weiters beinhal-

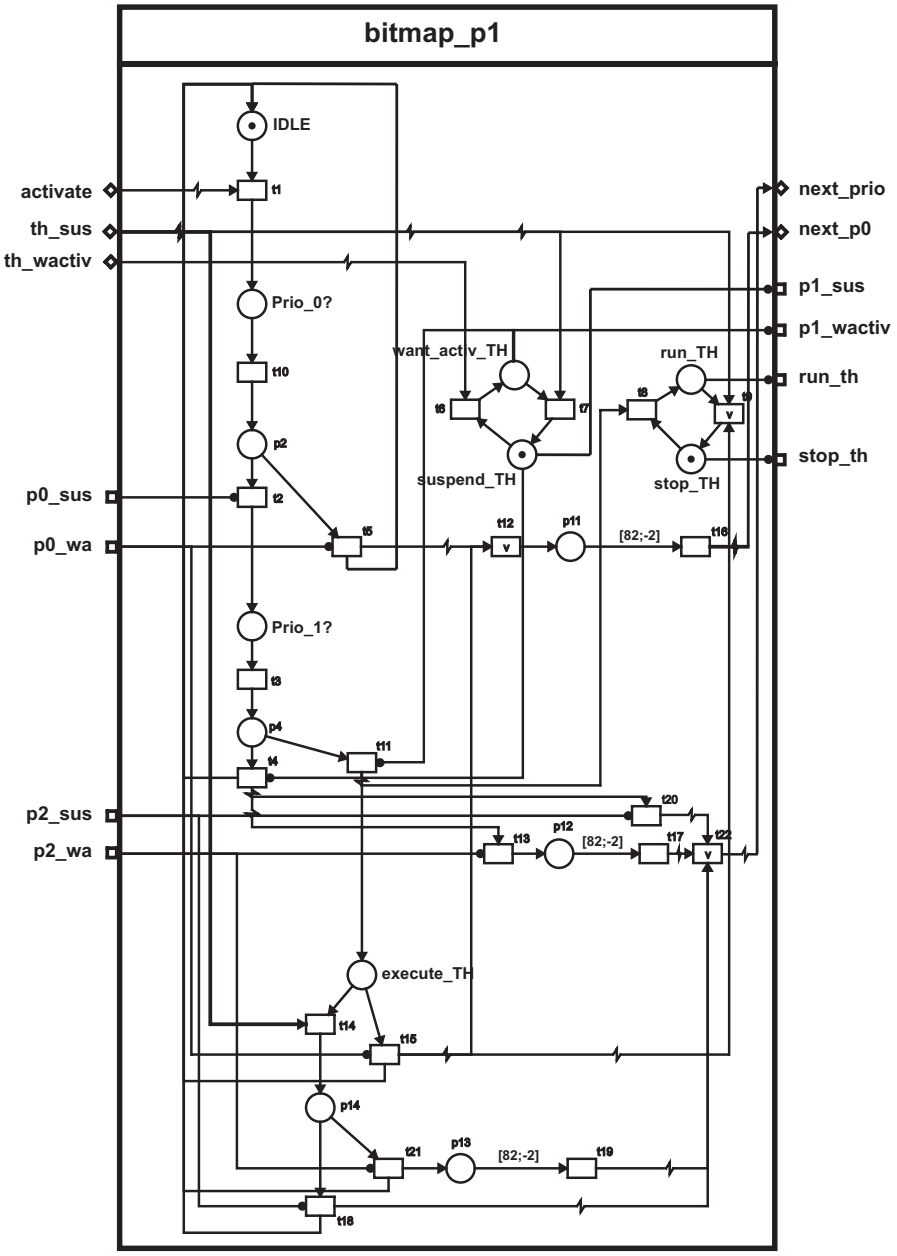


Abbildung 4.3: NCES Modell Bitmap Scheduler Priorität 1

tet das Modul **mlq_p1_2th** die Module **WAUP**, **next_PRIO**, **PRIO_1**, **HP**, **activate_TH1** und **activate_TH2**. Das Modell dieser Module wird auch nicht angeführt, da sie eine simple ODER Verknüpfung darstellen. Sobald ein Eingangsereignis (**eix**) aktiviert wurde, wird das Ausgangsereignis (**eo1**) generiert.

Angefangen vom Modul **Select_2TH_p1** (siehe Abbildung A.13) wird für zwei Threads an der Priorität 1 anhand einer Schrittkette (ähnlich wie beim Bitmap Scheduler im Kapitel zuvor) zuerst entschieden, ob nicht die höhere Priorität 0 abgearbeitet werden möchte. Wenn nicht, wird einer der beiden Threads am MLQ Scheduler ausgeführt. Dies wird mit den beiden Module **execute_2TH_p1** (siehe Anhang A.2 Abbildung A.14) über die Eingangsbedingungen des Moduls **Select_2TH_p1** bestimmt. Möchte die Priorität 0 abgearbeitet werden, wird nach einer Kontextwechselzeit von 82 das Ausgangsereignis **next_prio** über das Modul **next_PRIO** generiert.

Das Modul **execute_2TH_p1** gibt zum einen über die Ausgangsbedingungen **active** und **passive** dem Modul **Select_2TH_p1** bekannt, welcher Thread zuletzt abgearbeitet wurde und definiert zum anderen die Reihenfolge der Abarbeitung der zwei Threads. Dieses Modul repräsentiert sozusagen einen FIFO, der für die Ausführung des Timesclicing verantwortlich ist. Mit dem Modul kann nun definiert werden, welcher Thread an der Priorität des MLQ Schedulers als erster abgearbeitet wird. Mit den Ausgangsbedingungen **wactive_th** (Thread aktiv) und **sus_th** (Thread unterbrochen) wird der Status des Threads an der Priorität 1 nach außen hin bekannt gegeben. Über die Ausgangsbedingungen **run_th** und **stop_th** wird der jeweilige Thread angesteuert. Eine wichtige Implementierung in diesem Modul ist das Timeslicing. Sobald der Thread ausgeführt wird, kann dieser durch das Timeslicing, eine höhere Priorität oder durch sich selbst unterbrochen werden (siehe Anhang A.2 Abbildung A.14 Stelle **execute_TH**). Nach der Ausführung des Threads wird wieder über eine, Schrittkette entschieden, ob der nächste Thread an der Priorität 1 verarbeitet werden möchte (mit dem Ausgangsereignis **next_th**) oder ob auf die nächste Priorität mit Ausgangsereignis **next_prio** verzweigt werden soll. Weiters ist in diesem Modul noch die Kontextwechselzeit implementiert, sobald die Abarbeitung eines weiteren Threads bevorsteht.

Mit dem Modul **Select_HP** wird mit den Eingangsbedindungen **px_wa** und **px_sus** (x steht für die Priorität) entschieden, auf welche höhere Priorität verzweigt werden soll, wenn die aktuelle Priorität von einer höheren Priorität unterbrochen wurde. Da bei einem Wechsel zu einer höheren Priorität Zeit verbraucht wird, wurde die schon bekannte Kontextwechselzeit eingefügt. Das Modell ist wiederum im Anhang A.2 Abbildung A.15 angeführt.

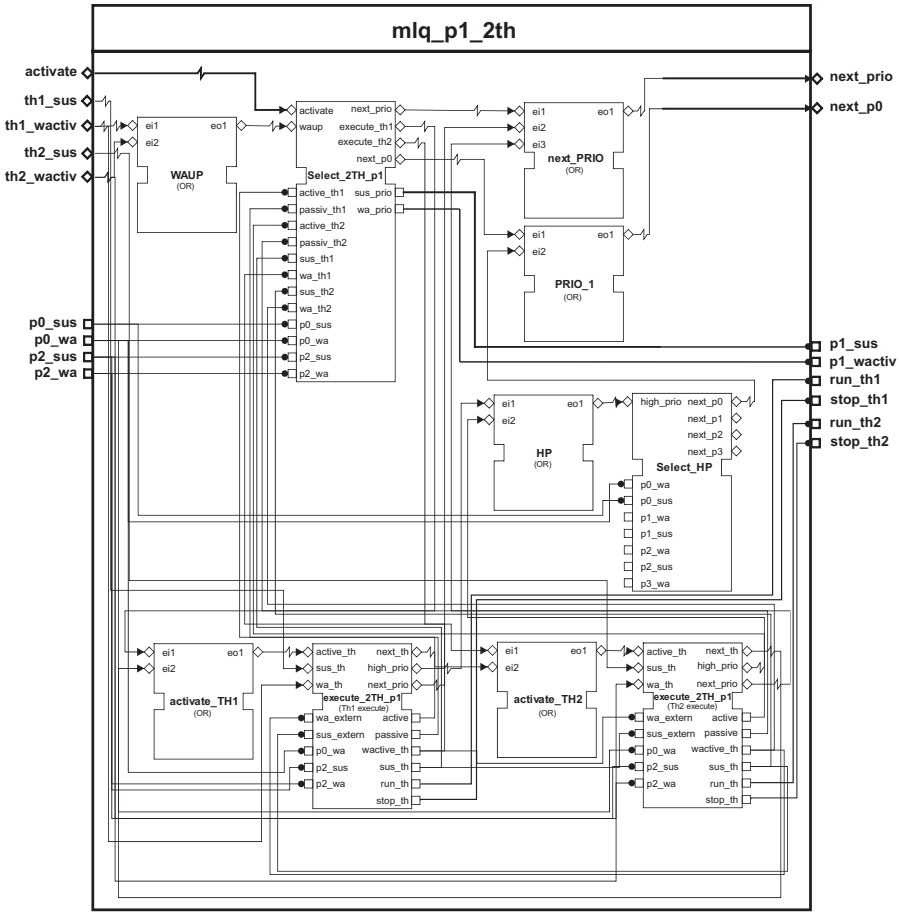


Abbildung 4.4: NCES Modell MLQ Scheduler Priorität 1 für zwei Threads

NCES Modell für die Zeiten Thread Unterbrechung und Wiederaufnahme

Dieses NCES Modell steht im Zusammenhang mit den Modellen des eCos Schedulers. Sobald ein Scheduler mit den Modulen Bitmap und MLQ zusammengestellt wurde, muss noch das Modul **IDLE_Prio5** aus Abbildung 4.5 am Ende der Schedulerkonstruktion eingefügt werden. Ein fertig konstruierter Scheduler beginnt mit seiner Abarbeitung an der Stelle **wait_for_THs** in Abbildung 4.5. Sobald ein Thread an einer Priorität im Scheduler bereit für die Abarbeitung ist, wird über die Eingangsbedingungen **wa_px** und **sus_px** (x steht für die Prioritäten 0 bis 4, am Beispiel dieses Moduls) auf die entspre-

chende Priorität verzweigt. Erst nach einer Verzögerungszeit **Thread Wiederaufnahme** von 13, wird über das Ausgangsereignis **next_px** die entsprechend aktive Priorität aktiviert.

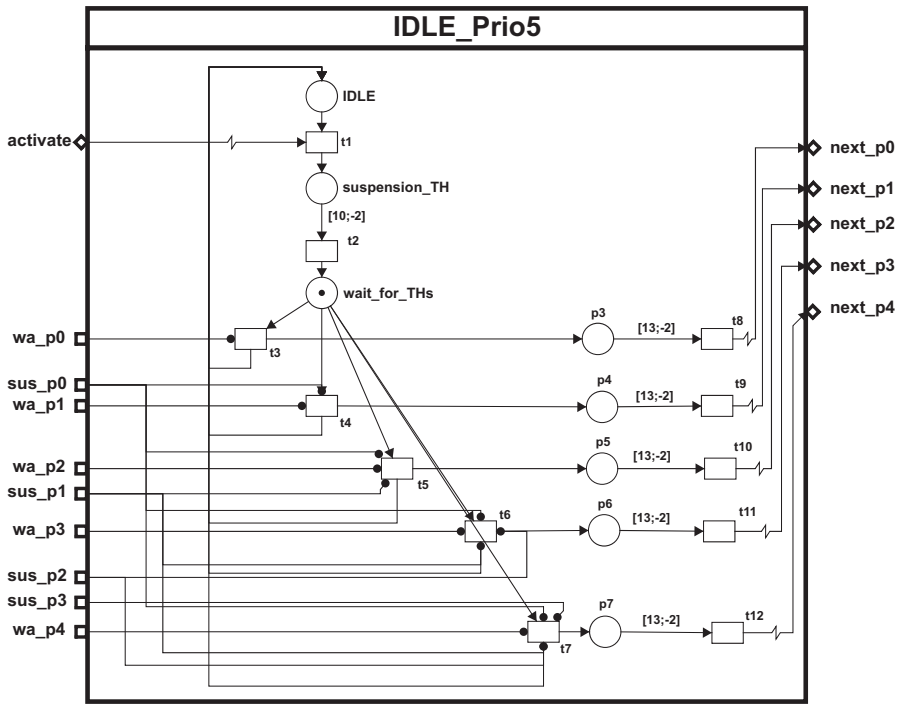


Abbildung 4.5: NCES Modell für die Einführung der Zeiten Thread Unterbrechung und Wiederaufnahme im Scheduler

Nach einer erfolgten Aktiverung einer Priorität über die Ausgangsereignisse **next_px**, befindet sich die Marke in Abbildung 4.5 an der Stelle **IDLE**. Die Markierung wandert erst in der Stelle **suspension_TH** über das Eingangsereignis **activate** und das wird erst dann generiert, sobald kein Thread mehr abgearbeitet werden möchte. Ab diese Stelle tritt die Verzögerungszeit **Thread Unterbrechung** auf, die mit den Wert 10 versehen wurde. Nach Ablauf dieser Zeit gelangt die Markierung wieder zur Stelle **wait_for_THs**, wo auf die Aktivierung eines Threads gewartet wird.

NCES Modelle Semaphore und Mutex

Bereits im Kapitel 3.3.3 wurden die NCES Module und das Verhalten anhand eines Aktivitätsdiagrammes der beiden Synchronisations- Mechanismen erklärt. Nun soll im folgenden das NCES Modell von Semaphore und Mutex angeführt werden. Es soll nochmals angemerkt werden, dass die beiden Synchronisations- Mechanismen so modelliert wurden, dass sie speziell für die μCrons Laufzeitumgebung Verwendung finden.

Wie schon im Kapitel 3.3.3 angeführt, wurde der Semaphore im Zusammenhang mit dem Event Dispatcher verwendet. Die Abbildung 4.6 stellt das NCES Modell für den Semaphore dar. Mit jeweils zwei Transitionen und Stellen wurde ein Flip Flop realisiert, mit dem die Bedingungsausgäge **FALSE** und **TRUE** gesetzt werden. An der Stelle **p1** befindet sich am Anfang die Markierung, die die Ausgangsbedingung **FALSE** setzt.

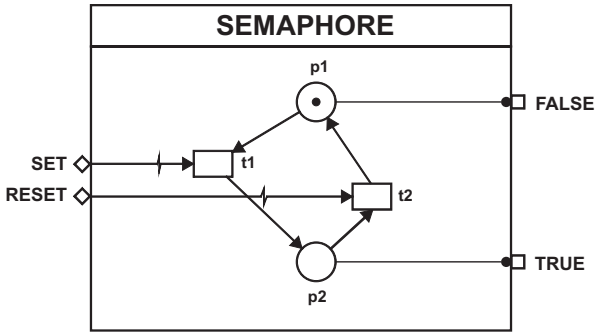


Abbildung 4.6: NCES Modell des Semaphore

Sobald ein Ereignis an am Ereigniseingang **SET** anliegt, wandert die Markierung von der Stelle **p1** über der Transition **t1** zur Stelle **p2**. Die Ausgangsbedingung **FALSE** ist nicht mehr aktiv und die Markierung an der Stelle **p2** bewirkt eine Aktivierung der Ausgangsbedingung **TRUE**. Die Markierung gelangt erst wieder an der Stelle **p1**, wenn am Eingangsereignis **RESET** ein Ereignis anliegt.

In Abbildung 4.7 ist das NCES Modell des Mutex angeführt, das in diesem Beispiel nur eine externe Ereignisquelle verwaltet. Der Mutex hat somit die Aufgabe, das externe Ereignis, das zu einem Thread geführt wird, zu „kapseln“ und dieses in dem Event Dispatcher abzulegen (siehe Kapitel 3.3.3). Für die Ablage eines Ereignisses im Event Dispatcher sind die drei angeführten Stellen/Transitions Kombinationen in Abbildung 4.7 zuständig. Die Stel-

len/Transitions Kombination mit der Stelle **IDLE** bestimmt über die Bedingungseingänge **enable** und **NOTenable**, ob das Modul **Event Dispatcher** und das Modul **RT_Control** aktiviert werden oder nicht. Die restlichen zwei Stellen/Transitions Kombinationen mit **setD**/**resetD** und **setRT**/**resetRT** dienen ausschließlich der Ansteuerung der NCES Module **Event Dispatcher** und **RT_Control**. Die Ansteuerung erfolgt getrennt über die Ausgangsbedingungen **enableD** für das NCES Modul **Event Dispatcher** und **enableRT** für das NCES Modul **RT_Control**. Diese getrennte Aktivierung der beiden Module ist wichtig, da somit die Eigenschaft Priority Inversion des Mutex zur Geltung kommt. Somit kann der Event Dispatcher über **enableD** kurzzeitig aktiviert werden, wenn dieser noch ein Ereignis abarbeiten muss.

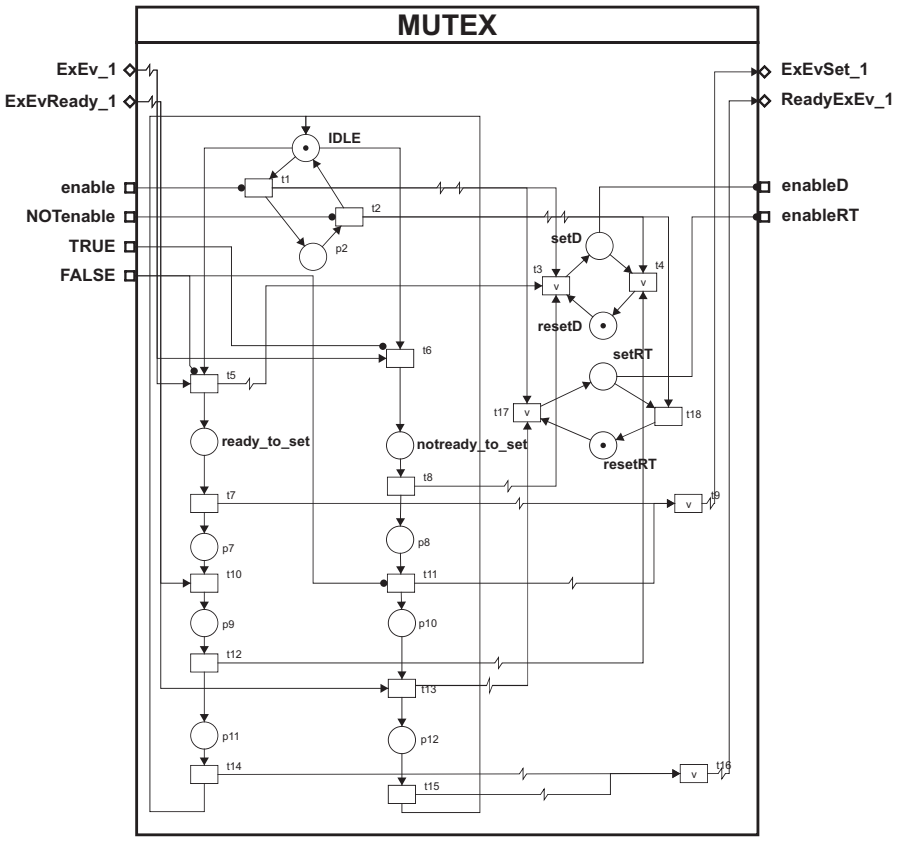


Abbildung 4.7: NCES Modell des Mutex für eine externe Ereignisquelle

Befindet sich die Markierung in Abbildung 4.7 an der Stelle **IDLE**, dann wandert diese, sobald ein externes Ereignis am Thread anliegt, zu den Stel-

len **ready_to_set** oder **notready_to_set**. Liegt nun ein Ereignis am Eingangsereignis **ExEv_1** an, wird mit den Bedingungseingängen **TRUE** und **FALSE** (vom Semaphore) auf die entsprechenden Stellen **ready_to_set** oder **notready_to_set** verzweigt. Der Zweig mit der Transition **t5** und der Stelle **ready_to_set** sagt aus, dass ein Ereignis im Event Dispatcher (dieser ist noch blockiert) über das Ausgangsereignis **ExEvSet_1** abgelegt werden kann. Nach erfolgter Ereignisablage über das Eingangsereignis **ExEvReady_1** und Deaktivierung des Event Dispatchers mit der Transition **t12** (**resetD**) wird zur Stelle **IDLE** zurückgekehrt. Betrachtet man den Zweig mit der Transition **t6** und der Stelle **notready_to_set**, so ist der Event Dispatcher noch nicht bereit, das externe Ereignis zu „empfangen". Das externe Ereignis muss sozusagen „warten" bis der Event Dispatcher das aktuell enthaltene Ereignis abarbeitet. Mit der Transition **t8** wird das Modul des Event Dispatchers aktiviert und das aktuell vorhandene Ereignis im Event Dispatcher kann abgearbeitet werden. Nach dessen Abarbeitung wird über den Semaphore die Eingangsbedingung **FALSE** aktiviert. Mit dieser Eingangsbedingung kann der Mutex nun das „wartende" externe Ereignis in den Event Dispatcher hineinstellen.

4.3.2 NCES Modelle der μCrons Laufzeitumgebung

Im folgenden Abschnitt wird nun das NCES Modell des Event Dispatchers angeführt. Die restlichen Modelle sind im Anhang A.2.2 aufgelistet. Bereits im Kapitel 3.4.1 wurde das Interface und das Verhalten dieses formalen Modells dargestellt und erklärt.

NCES Modell Event Dispatcher

Das NCES Modell des Event Dispatchers in Abbildung 4.8 kann 3 verschiedene Ereignisse unterscheiden und in der Event-Queue können bis zu drei Ereignisse abgelegt werden. Dieser kleine Event Dispatcher wurde deshalb angeführt, da er für die Erklärung des Modells einfacher ist. Weiters ist der Event Dispatcher aus einer Verbindung von mehreren NCES Modulen zusammengesetzt. Für die Ablage eines Ereignisses muss am entsprechenden Eingangsereignis **eix** (x ist als Platzhalter für eine Zahl 1 bis 3 zu lesen) ein Ereignis anliegen, das das NCES Modul **Eventx** aktiviert. Mit dem NCES Modul **InsertControl** wird das Ereignis weiter verarbeitet. Das Modul kontrolliert mit Hilfe des Moduls **checker** im Modul **Level**, ob in der **Event-Queue** Ereignisse abgelegt werden können. Nach dieser Kontrolle und Information der beiden Module **checker** und **Level**, entscheidet das Modul **InsertControl** ob das Ereignis in

der **Event-Queue** abgelegt werden soll oder nicht. Mit dem Ausgangsereignis **insert** vom Modul **InsertControl** wird das Ereignis in der **Event-Queue** abgelegt und mit **register** wird die Anzahl der Ereignisse in der Event-Queue im Modul **Level** erhöht. Anschließend wird das entsprechende Ereignis mit dem Ausgangsereignis **eixready** über das Modul **OUTready_EIx** gesendet. Konnte das Ereignis nicht in der **Event-Queue** abgelegt werden, dann wird nur das Senden des entsprechenden Ausgangsereignisses **eixready** ausgeführt.

Für das Herausnehmen von Ereignissen aus dem Modul des Event Dispatchers (bzw. aus der Event-Queue) dient das Modul **OutControl**. Sobald ein Ereignis am Eingangsereignis **fetch** anliegt wird das Modul **OutControl** aktiviert und in der **Event-Queue** wird nach dem FIFO Prinzip ein Ereignis herausgenommen. Das Modell der **Event-Queue** ist so modelliert, dass jeder Position (in diesem Fall drei) eine Stelle entspricht. Die Ereignisse werden durch die Anzahl an Marken in der Stelle modelliert. Wird in die **Event-Queue** ein neues Ereignis hineingestellt, dann wird dieses an der untersten, freien Stelle abgelegt. So werden in der **Event-Queue** beim Herausnehmen eines Ereignisses über eine Schrittkette die Ereignisse zunächst um eins nach unten weitergeschoben. Das Ereignis an der untersten Position ist das älteste Ereignis und wird in eine Hilfsposition geschoben. Hier erfolgt über das Modul **OutControl** eine Überprüfung, welches Ereignis in dieser Position enthalten ist. Bei der Überprüfung wird von der höchsten Nummer eines Ereignisses in absteigender Folge begonnen, bis ein oder kein Ereignis gefunden wurde. Sobald ein oder kein Ereignis gefunden wurde, wird das Ausgangsereignis **eixOUT** oder **nothingOUT** generiert.

Funktionsblocknetzwerk als NCES Modell

In diesem Kapitel soll erläutert werden, wie ein IEC 61499 Funktionsblocknetzwerk, im Zusammenhang mit der μCrons Laufzeitumgebung, als NCES Modell realisiert wird. Als Ausgangspunkt für die Erstellung eines solchen dient das IEC 61499 Funktionsblocknetzwerk in Abbildung 4.9a. Mit dem Funktionsblock **E_RESTART** wird das Funktionsblocknetzwerk bestehend aus den „einfachen“ Funktionsblöcken **FB1**, **FB2** und **FB3** abgearbeitet. Als erster wird der Funktionsblock **FB1** mit dem Eingangsereignis **ei1** abgearbeitet und im Folgenden **FB2** mit dem Eingangsereignis **ei2** und **FB3** mit dem Eingangsereignis **ei3**.

Das entsprechende NCES Modell für das IEC 61499 Funktionsblocknetzwerk ist in Abbildung 4.9b dargestellt. Das NCES Modell besteht zum einen aus dem NCES Modul **Thread_exe** für die Ereignisablage und zum anderen aus den NCES Modulen **ei1**, **ei2_ei3**, **eo2** und **eo3** vom Typ **SWITCH**

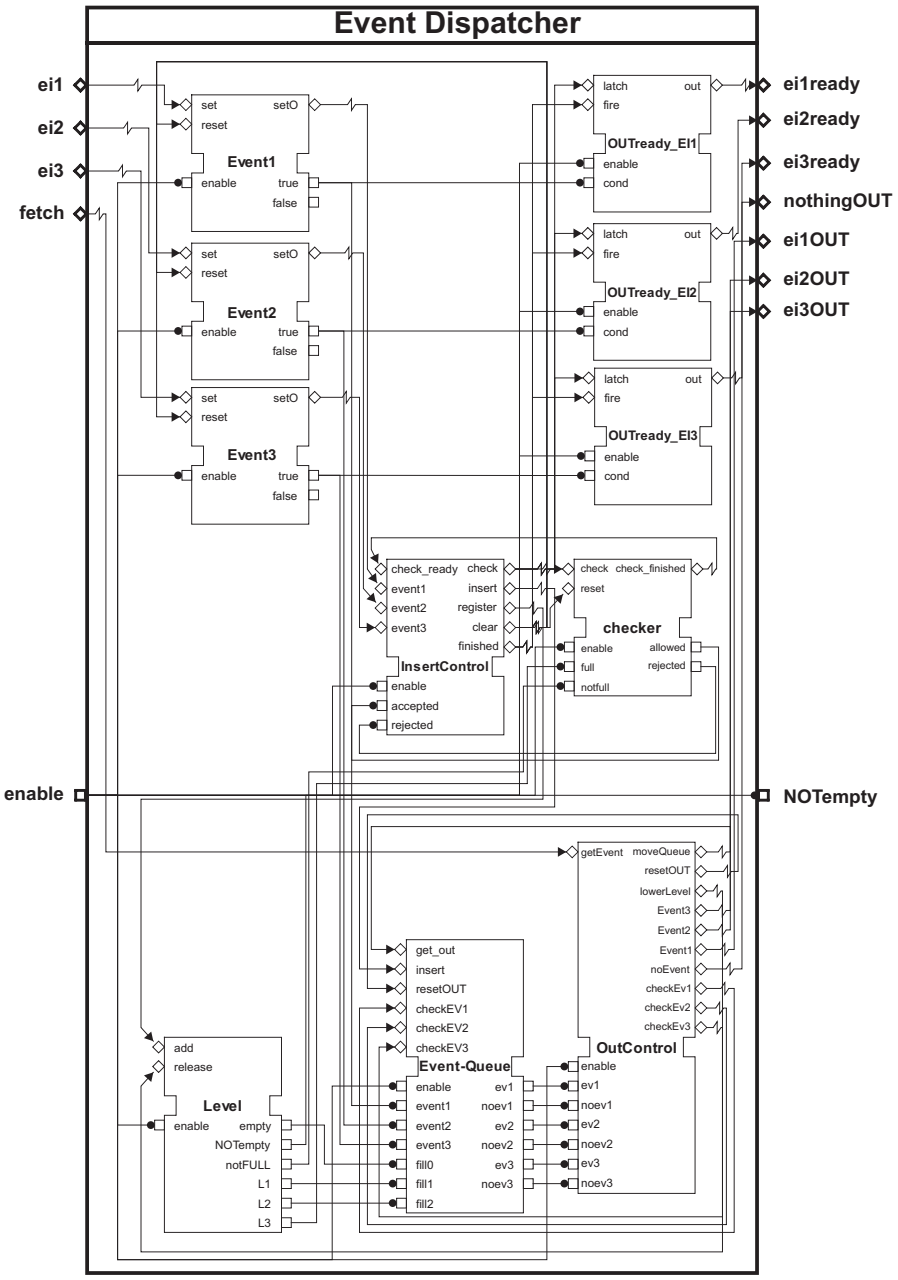
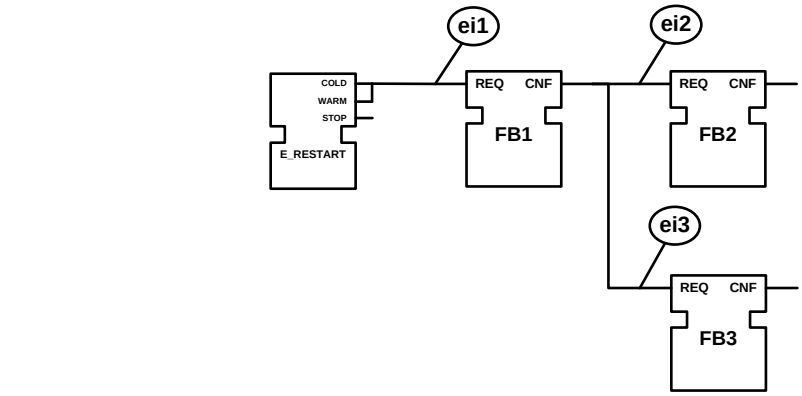
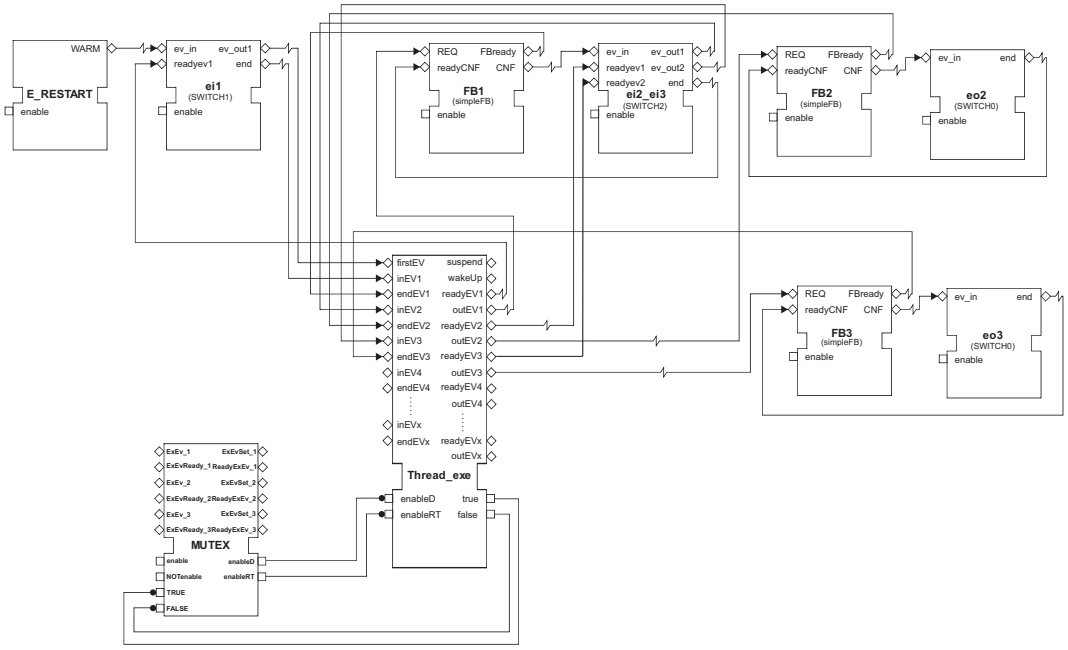


Abbildung 4.8: NCES Modell des Event Dispatchers mit drei Eingangsereignissen



(a)



(b)

Abbildung 4.9: Funktionsblocknetzwerk als a) IEC 61499 und b) NCES Modell

für die Ereignisverzweigung. Die Funktionsblöcke **FB1**, **FB2** und **FB3** sind durch das NCES Modul vom Typ **simpleFB** dargestellt. Für den IEC 61499 E_RESTART Funktionsblock wurde das äquivalente NCES Modul **E_RESTART** realisiert. Diese NCES Modul generiert das Ausgangsereignis **WARM**, sobald die Eingangsbedingung **enable** aktivert wurde. Zusätzlich wurde noch das Modul **MUTEX** eingefügt, das jedoch im Zusammenhang

mit dem IEC 61499 Funktionsblock in Abbildung 4.9a keine Rolle spielt. Es wurde lediglich angeführt um zu zeigen, dass der **MUTEX** im Zusammenhang mit dem Modul **Thread_exe** in Verbindung steht. Für die Verschaltung der einzelnen NCES Modul dient zum Verständnis die Tabelle 4.3.

Interface von		**Interface zu**	
NCES Modul	**Ausgangs-ereignis**	**NCES Modul**	**Eingangs-ereignis**
E_RESTART	*WARM*	**ei1**	*ev_in*
ei1	*ev_out1*	**Thread_exe**	*firstEV*
	end	**Thread_exe**	*inEV1*
FB1	*FBready*	**Thread_exe**	*endEV1*
	CNF	**ei2_ei3**	*ev_in*
ei2_ei3	*ev_out1*	**Thread_exe**	*inEV2*
	ev_out2	**Thread_exe**	*inEV3*
	end	**FB1**	*readyCNF*
FB2	*FBready*	**Thread_exe**	*endEV2*
	CNF	**eo2**	*ev_in*
eo2	*end*	**FB2**	*readyCNF*
FB3	*FBready*	**Thread_exe**	*endEV3*
	CNF	**eo3**	*ev_in*
eo3	*end*	**FB3**	*readyCNF*
Thread_exe	*readyEV1*	**ei1**	*readyev1*
	outEV1	**FB1**	*REQ*
	readyEV2	**ei2_ei3**	*readyev1*
	outEV2	**FB2**	*REQ*
	readyEV3	**ei2_ei3**	*readyev2*
	outEV3	**FB3**	*REQ*
NCES Modul	**Ausgangs-bedingung**	**NCES Modul**	**Eingangs-bedingung**
Thread_exe	*true*	**MUTEX**	*TRUE*
	false	**MUTEX**	*FALSE*
MUTEX	*enableD*	**Thread_exe**	*enableD*
	enableRT	**Thread_exe**	*enableRT*

Tabelle 4.3: Verschaltung des Funktionsblocknetzwerks aus Abb. 4.9b

4.4 Verifikation einer Applikation

In diesem Kapitel wird ein Applikationsbeispiel auf dessen Verhalten verifiziert. Das Applikationsbeispiel wurde mit den formalen Modellen vom Echtzeitbetriebssystem eCos und der μCrons Laufzeitumgebung vom Kapitel 4.3 realisiert. Für das korrekte Verhalten der Applikation werden über das Programm ViVe (siehe Kapitel 4.2.2) Abfragen durchgeführt. Als Spezifikationsabfragen wurden jene vom integrierten Model-Checker verwendet, der die Logikabfragen unterstützt. Die Verwendung von CTL Abfragen konnte nicht genutzt werden, da der Erreichbarkeitsgraph der NCES Applikationen mit dem Model-Checker generiert wurde und nicht mit dem CTL Checker (siehe Kapitel 4.2.2). Grund für die Erstellung des Erreichbarkeitsgraphen mit dem Model-Checker war jener, dass im Zusammenhang mit den formalen Modellen eine Deadlook Freiheit erzielt wurde. Trotzdem kann man mit den Logikabfragen z. B. das Verhalten des MLQ Schedulers an der Priorität 2 in Abbildung 4.10 prüfen.

Die Abbildung 4.10 zeigt symbolisch das zu verifizierende Applikationsbeispiel, bestehend aus einem eCos Scheduler mit vier Prioritäten, External Event Timer, External Event Manager und drei Threads. Der eCos Scheduler besitzt an der Priorität 0, 1 und 3 einem Bitmap Scheduler. Die Priorität 2 des Schedulers in Abbildung 4.10 ist mit einem MLQ Scheduler ausgestattet, der die Threads 1 und 2 ansteuert. Der External Event Timer ist an der Priorität 0 des Schedulers angeführt und hat somit die höchste Priorität. Er steht in Verbindung mit dem Thread 3, da in diesem ein Funktionsblocknetzwerk mit Zeit-Funktionsblöcke enthalten ist. An der Priorität 1 befindet sich der External Event Manager, der im Beispiel in Abbildung 4.10 zu einem gewissen Zeitpunkt ein externes Ereignis (z. B. weil eine Nachricht vom Netzwerk empfangen wurde) dem Thread 3 weiterleitet.

Der Thread 1 und Thread 2 werden über den MLQ Scheduler an der Priorität 2 des eCos Schedulers angesteuert. In diesem Applikationsbeispiel bestehen die formalen Modelle dieser Threads aus keinem Funktionsblocknetzwerk, Laufzeitumgebung und Mutex, wie in Kapitel 3.4.6 angeführt. Sie sind lediglich dazu da, um die Funktionsweise des MLQ Schedulers darzustellen. Die interne Realisierung der Threads 1 und 2 wurde soweit implementiert, dass diese nach einer bestimmten Zeit aktiv werden und mit dem Einbau einer Verzögerungszeit eine Abarbeitung simulieren. Der Thread 1 wird zweimal ausgeführt, der Thread 2 nur einmal.

Das formale Modell von Thread 3 beinhaltet hingegen ein Funktionsblocknetzwerk, Laufzeitumgebung und Mutex. Die Abbildung 4.11 zeigt das Funktionsblocknetzwerk im Thread 3 als IEC 61499. Mit dem Funktionsblock **E_RESTART** werden die Zeit- Funktionsblöcke **E_CYCLE** mit dem Ein-

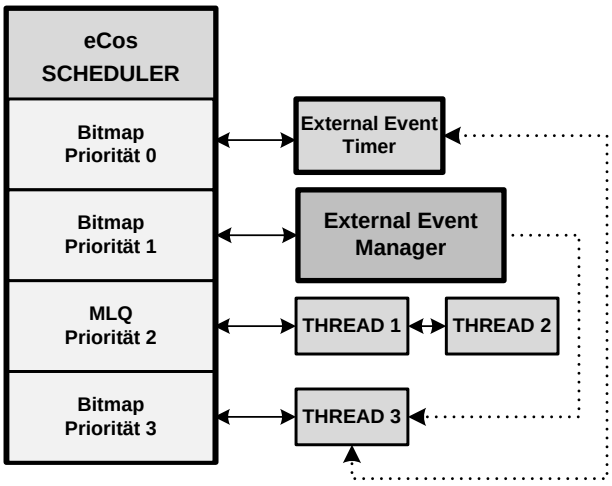


Abbildung 4.10: Verifikation eines Applikationsbeispiels

gangsereignis **ev1** und **E_DELAY** mit dem Eingangsereignis **ev2** über die jeweiligen Eingangsereignisse **START** angestoßen. Der Funktionsblock **E_CYCLE** generiert alle $5ms$ über **EO** ein Ausgangsereignis und wird nach $10ms$ mit dem Funktionsblock **E_DELAY** über das Eingangsereignis **STOP** angehalten.

Mit dem Eingangsereignis **ev3** wird der Funktionsblock **FB1** abgearbeitet. Nach dessen Abarbeitung wird mit **ev4** und **ev5** zuerst der Funktionsblock **FB2** und dann **FB3** abgearbeitet. Nach der Abarbeitung vom Funktionsblock **FB3** wird mit dem Eingangsereignis **ev6** der Funktionsblock **FB4** angestoßen. Nach der Abarbeitung von **FB4** wird zuerst mit **ev7** der Funktionsblock **FB5** und dann mit **ev8** der Funktionsblock **FB6** verarbeitet.

Über den Service Interface Funktionsblock **SIFB** werden die Funktionsblöcke **FB3**, **FB5** und **FB6** erneut abgearbeitet. Der Funktionsblock **SIFB** wird über ein externes Ereignis vom External Event Manager aktiviert.

Das Verhalten des Applikationsbeispiels in Abbildung 4.10, lässt sich aus dem zeitlichen Ablauf des Beispiels in Abbildung 4.13 erkennen. Auf der Abszisse befindet sich die Zeit t und auf der Ordinate ist die Priorität des eCos Schedulers angegeben. Im Modell wurden jene Zeiten verwendet, die in Tabelle 4.1 und 4.2 angeführt sind. So ist z. B. ersichtlich, dass bei der Zeit 0 der Thread 3 bereit ist für die Abarbeitung. Dieser wird erst nach einer Zeit $res(= 13)$ (entspricht der Wiederaufnahme eines Threads) abgearbeitet. Zur Zeit 15 wird der Thread 3 vom Thread 2 unterbrochen, da dieser auf einer höheren Priorität liegt. Nach einer Zeit KW (Kontextwechsel) von 82 wird zum Zeitpunkt 97 die Priorität 2 aktiv. Während der Kontextwechselzeit ist

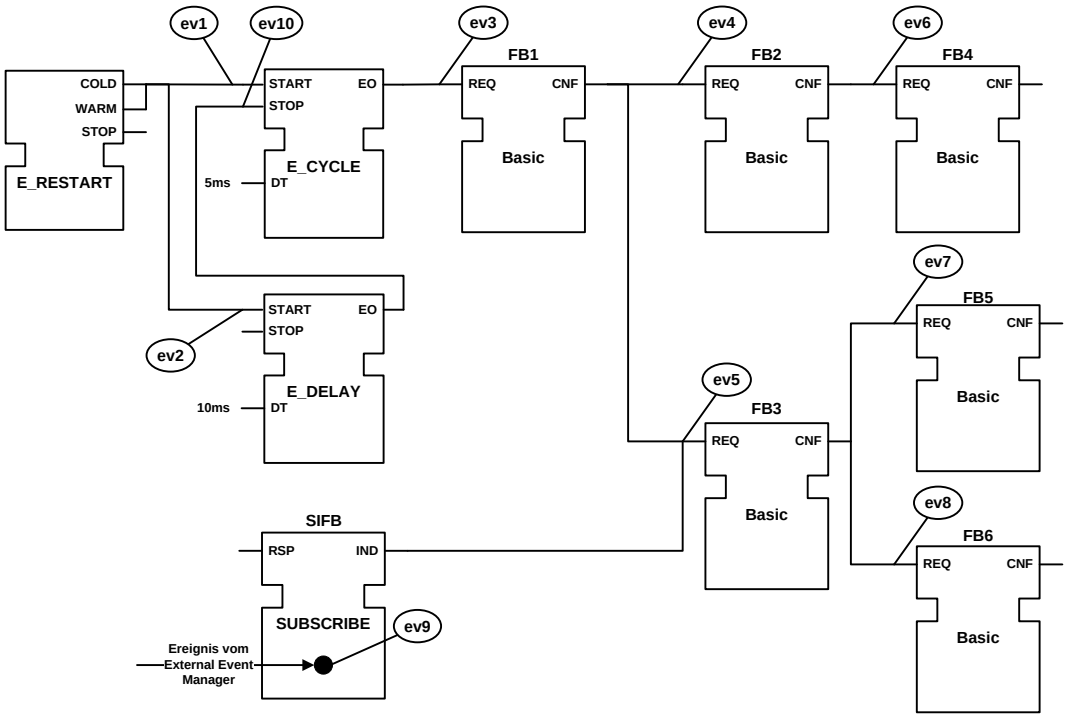


Abbildung 4.11: Interne Realisierung von Thread 3 aus Abb. 4.10

auch der Thread 1 zum Zeitpunkt 19 aktiv geworden. Da der Thread 1 als erster im MLQ Scheduler angelegt wurde, wird auch dieser zum Zeitpunkt 97 verarbeitet.

Für die Erläuterung des Verhaltens des Funktionsblocknetzwerkes in Thread 3 (siehe Abbildung 4.11), dient ein Ausschnitt der zeitlichen Abarbeitung ab dem Zeitpunkt, in dem der **SIFB** Funktionsblock ein Ereignis für die Abarbeitung der Funktionsblöcke **FB3**, **FB5** und **FB6** zur Verfügung stellt. Die Abfolge der Abarbeitung dieser Funktionsblöcke ist in der Abbildung 4.12 ersichtlich. Auf der Abszisse befindet sich wiederum die Zeit t und auf der Ordinate sind die wesentlichen Funktionsblöcke (für die Betrachtung dieses Ausschnitts) von Thread 3 angegeben. Zum Zeitpunkt 6356 (siehe Abbildung 4.12) wird der Funktionsblock **FB3**, durch dem zur Verfügung gestellten Ereignis vom **SIFB** Funktionsblock, wieder abgearbeitet.

Die Funktionsblöcke **FB5** und **FB6** werden zu den Zeitpunkten 7113 und 7284 erneut abgearbeitet. Weiters gibt die Zeitdauer $OUTevX$ (X steht für das entsprechende Ereignis) an, wie lang ein Ereignis aus der Event-Queue herausgenommen wird. Auf den restlichen zeitlichen Verlauf von Abbildung 4.12 und 4.13 soll weiters nicht eingegangen werden.

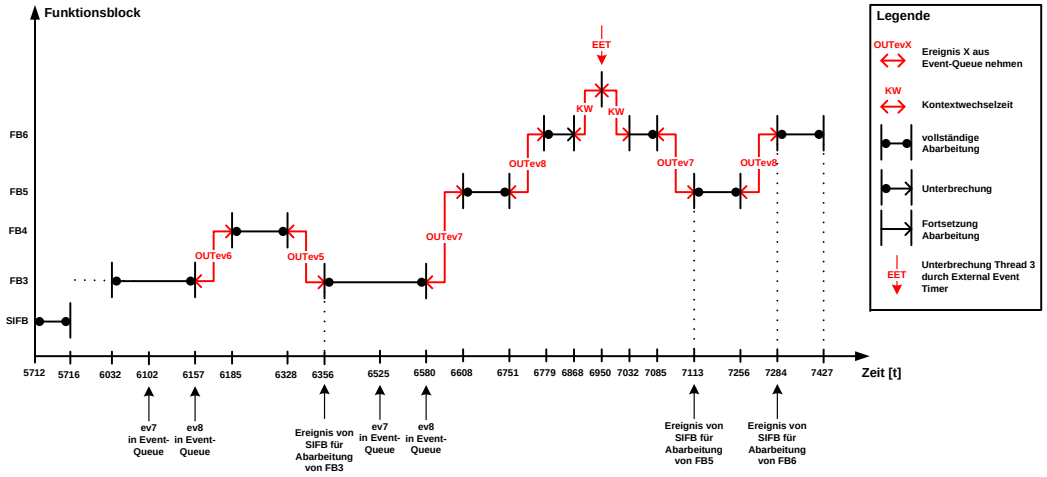


Abbildung 4.12: Ausschnitt des zeitlichen Verhaltens von Thread 3 aus Abb. 4.11

Nun soll mehr Aufmerksamkeit auf die Verifikationsabfragen des Applikationsbeispiels gelegt werden. Betrachtet man den eCos Scheduler in Abbildung 4.10, so kann durch eine erste Abfrage dessen allgemeine Funktionsweise überprüft werden. Der Scheduler hat die Aufgabe, dass External Event Timer, External Event Manager und die drei Threads nie gleichzeitig auftreten können. Mit einer Logikabfrage im Programm ViVe kann dieses Verhalten kontrolliert werden. Jede Priorität des eCos Schedulers als NCES Modell, sei es Bitmap oder MLQ, besitzt eine Ansteuerung für den External Event Timer, External Event Manager und den Threads. Über einer Stellenkombination **run_TH** und **stop_TH** (siehe z.B. Abbildung 4.3) kann das anliegende Element aktiviert oder deaktiviert werden. Für den Nachweis des korrekten Verhaltens des eCos Schedulers in Abbildung 4.10 muss nun eine Spezifikationsformel erstellt werden, wobei geprüft wird, ob nie mehr als eine Priorität aktiv ist. Für die Spezifikationsformel ergibt sich eine UND-ODER Verknüpfung (siehe Spezifikationsformel 4.1) mit den jeweiligen Stellen **run_TH** der unterschiedlichen Prioritäten, wobei im Erreichbarkeitsgraphen keine Zustände markiert sein sollten. Das bedeutet, dass das Verhalten des eCos Schedulers korrekt ist.

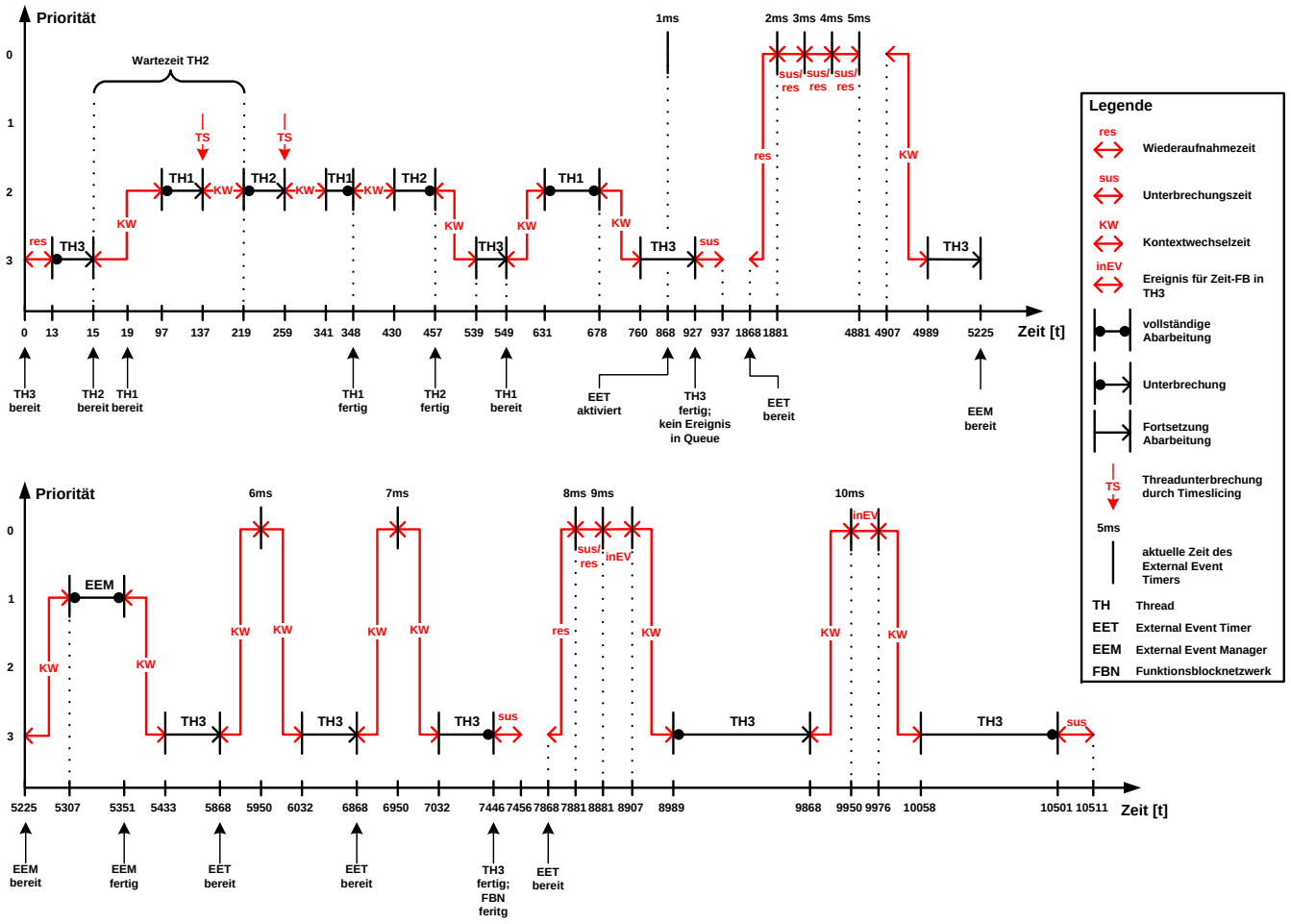


Abbildung 4.13: Zeitliches Verhalten des Applikationsbeispiels von Abb. 4.10

Sobald bei dieser Abfrage im Erreichbarkeitsgraphen ein oder mehrere Zustände markiert sind, dann würde es bedeuten, dass das erstellte NCES Modell des Schedulers sich nicht so verhält wie gewünscht. Die markierten Zustände im Erreichbarkeitsgraphen geben Auskunft darüber, dass die angegebene Spezifikation im NCES Modell bei den jeweiligen Zuständen auftritt.

Die Abbildung 4.14 zeigt diese Spezifikationsabfrage mit dem Programm ViVe. In der Umrahmung 1 kann man die Spezifikationsformel eingeben und anschließend wird überprüft, ob Zustände der Spezifikation im Erreichbarkeitsgraph entsprechen. In der Umrahmung 2 erhält man das Ergebnis der Abfrage. In diesem Fall wurden keine Zustände gefunden (*No state found*).

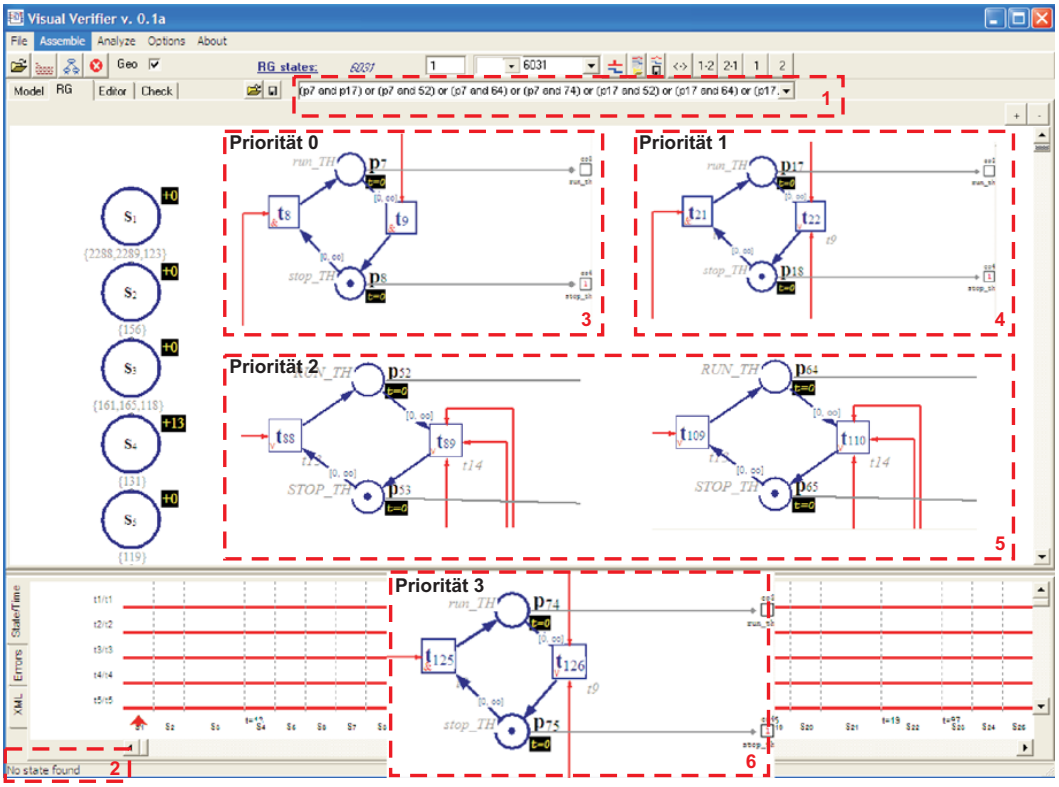


Abbildung 4.14: Abfrage für das Prioritätsverhalten des eCos Schedulers von Abb. 4.10

Die Umrahmungen 3 bis 6 stellen die jeweiligen Ansteuerungen der Prioritäten des Schedulers dar. Hier findet man auch die Stellenkombination **run_TH** und **stop_TH** wieder. Mit den Stellen **p7**, **p17**, **p52**, **p64** und **p74** in den jeweiligen Prioritäten wurde die Spezifikationsformel 4.1 in der Umrahmung 1

definiert und anschließend verifiziert.

$$\begin{aligned} Spezifikationsformel\ 1 = (p7\ and\ p17)\ or\ (p7\ and\ p52)\ or\ (p7\ and\ p64) \\ or\ (p7\ and\ p74)\ or\ (p17\ and\ p52)\ or\ (p17\ and\ 64) \\ or\ (p17\ and\ 74)\ or\ (p52\ and\ p64)\ or\ (p52\ and\ 74) \\ or\ (p64\ and\ 74) \end{aligned} \tag{4.1}$$

Mit einer weiteren Abfrage kann mit dem MLQ Scheduler z. B. überprüft werden, ob die Threads am MLQ Scheduler an der Priorität 2 nicht gleichzeitig abgearbeitet werden. Die folgende Abbildung 4.15 zeigt diese Abfrage für das Beispiel in Abbildung 4.10. Wie schon zuvor beschrieben, besitzt jede Priorität eines Schedulers eine Stellenkombination **run_TH** und **stop_TH** für die Ansteuerung des anliegenden Threads und weiters noch eine Stellenkombination **want_active_th** und **suspend_th** (siehe z.B. Abbildung 4.3), um nach außen hin den Status der Priorität bekannt zu geben. Sobald nun z. B. die Stellen **p50** und **p52** eine Markierung besitzen, wird der Thread 1 am MLQ Scheduler der Priorität 2 abgearbeitet und der Thread 2 muss auf dessen Unterbrechung warten. Für den Nachweis des korrekten Verhaltens des MLQ Scheduler reicht es aus, wenn die beiden Stellen für die Aktivierung (**run_TH**) der Threads UND verknüpft werden. Somit kann mit der Spezifikationsformel 4.2 in der Umrahmung 1 in Abbildung 4.15 kontrolliert werden, ob der Fall auftreten könnte, wo beide Threads gleichzeitig abgearbeitet werden.

$$Spezifikationsformel\ 2 = (p52\ and\ p64) \tag{4.2}$$

Die Umrahmung 2 in Abbildung 4.15 zeigt als Ergebnis, dass keine Zustände im Erreichbarkeitsgraphen gefunden wurden, wo diese Spezifikation auftreten könnte. Daraus resultiert, dass das Verhalten des formalen Modell des MLQ Schedulers korrekt implementiert wurde.

Als weitere Abfrage in Bezug auf dem MLQ Scheduler, ist die Berechnung z. B. der Wartezeit von Thread 2 von der Bekanntgabe, dass er abgearbeitet werden will, bis zu seiner Aktivierung. Aus Abbildung 4.13 ist ersichtlich, dass der Thread **TH2** zum Zeitpunkt 15 bereit für die Abarbeitung ist. Erst zum Zeitpunkt 219 wird dieser jedoch abgearbeitet. Diese Wartezeit kann anhand der Spezifikationsformel 4.3 nachgewiesen werden.

$$Spezifikationsformel\ 3 = (p62\ and\ not\ p64) \tag{4.3}$$

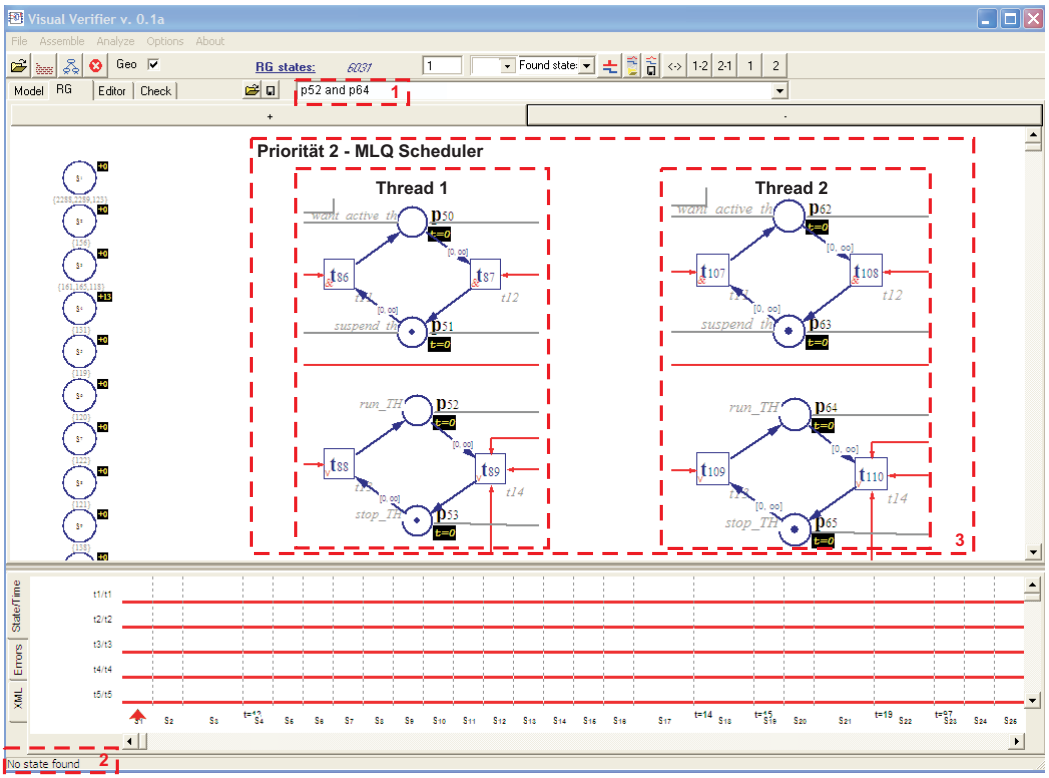


Abbildung 4.15: Abfrage für das Verhalten des MLQ Schedulers von Abb. 4.10

Die Stelle **p62** gibt an, dass der Thread 2 bereit ist für die Abarbeitung und die Stelle **p64** aktiviert den Thread **TH2**, sobald diese eine Markierung enthält. Mit der Negation **not** vor der Stelle **p64** wird ausgesagt, dass Zustände im Erreichbarkeitsgraphen markiert sind, bis die Stelle **p64** eine Markierung besitzt.

Die nächste Abfrage befasst sich mit der Funktionsweise des Mutex und Semaphores im Thread 3. Der Mutex hat die Aufgabe, externe Ereignisse, die in den Thread 3 gelangen, zu „kapseln“. Das externe Ereignis wird in den Event Dispatcher hineingestellt, sobald der Semaphore die Erlaubnis dem Mutex erteilt. Bevor auf die Abbildung 4.16 eingegangen wird, soll zuvor der zeitliche Verlauf des Applikationsbeispiels in Abbildung 4.13 betrachtet werden. Zum Zeitpunkt 5225 in Abbildung 4.13 wird der Thread **TH3** vom External Event Manager **EEM** unterbrochen. Der External Event Manager möchte sofort nach einer Kontextwechselzeit zum Zeitpunkt 5307 ein externes Ereignis in den Event Dispatcher von Thread **TH3** hineinstellen. Im Zusammenhang

mit Abbildung 4.16 sieht man, dass das externe Ereignis erst nach einer gewissen Zeit im Zeitpunkt 5326 (siehe Umrahmung 5) in den Event Dispatcher hineingestellt werden kann, da ein Ereignis im Event Dispatcher enthalten ist und zuvor fertig abgearbeitet werden muss.

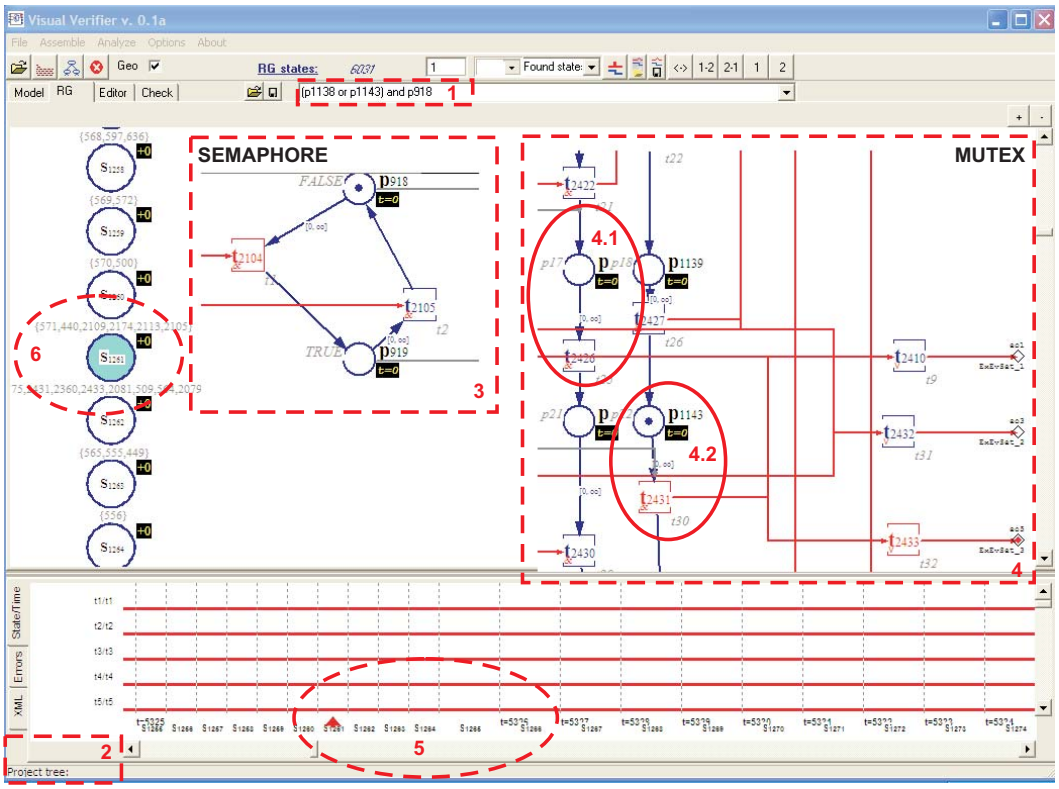


Abbildung 4.16: Abfrage für die Ermittlung des Zeitpunktes für die Ereignisablage vom External Event Manager im Zusammenhang mit dem Mutex und Semaphore in Thread 3

Diesen Zeitpunkt kann man mit der Spezifikationsformel 4.4 in der Umrahmung 1 in Abbildung 4.16 ermitteln. Die Stellen **p1138** und **p1143** entsprechen jene vom Mutex in den Umrahmungen 4.1 und 4.2. Die Stelle **p918(**$\widehat{=}$ **FALSE)** repräsentiert im Semaphore in der Umrahmung 3 nach außen hin die Ausgangsbedingung **FALSE** (siehe Abbildung 4.6).

$$Spezifikationsformel\ 4 = (p1138\ or\ p1143)\ and\ p918 \tag{4.4}$$

Ein externes Ereignis kann erst dann in den Event Dispatcher hineingestellt werden, wenn die Stelle **FALSE** des Semaphores (siehe Umrahmung 3) eine

Markierung besitzt. Diese Stelle gibt dem Mutex bekannt, dass das externe Ereignis nun in den Event Dispatcher abgelegt werden kann. Im Mutex wird nun entweder die Transition **t2426** (siehe Umrahmung 4.1) aktiviert, wenn das Ereignis gleich in den Event Dispatcher hineingestellt werden kann. In diesem Beispiel wird die Transition **t2431** (siehe Umrahmung 4.2) aktiviert, da die Stelle **p1143** mit einer Marke versehen ist. Dies bedeutet, dass in den Event Dispatcher ein Ereignis fertig abgearbeitet wird und auf die Bedingung **FALSE** vom Semaphore gewartete werden muss. Als Ergebnis erhält man in der Umrahmung 2 die Aussage *Project tree:* (Erzeuge Baum (Erreichbarkeitsgraph)). Im Erreichbarkeitsgraphen werden nun jene Zustände markiert, die der Spezifikationsformel entsprechen. Bei dieser Abfrage wird nur der Zustand **S1261** (siehe Umrahmung 6) im Ereichbarkeitsgraphen markiert, da nur das externe Ereignis vom External Event Manager von Interesse war. Mit dem ermittelten Zustand **S1261** kann nun im Zeitverlauf (siehe Umrahmung 5 Pfeil) der Zeitpunkt herausgelesen werden, ab wann das externe Ereignis vom External Event Manager in den Event Dispatcher hineingestellt werden kann.

In weiterer Folge können noch Abfragen in Bezug auf das Funktionsblocknetzwerk im Thread 3 (siehe Abbildung 4.11) durchgeführt werden. Mit zusätzlichen Spezifikationsabfragen könnte z. B. abgefragt werden, wann und wie oft ein Funktionsblock abgearbeitet wird, wann ein bestimmtes Ausgangsereignis auftritt oder wieviel Zeit insgesamt verbraucht wird, damit ein Ereignis in der Event-Queue abgelegt wird.

4.5 Zusammenfassung

In Kapitel 4.1 wurde einführend der Nutzen der formalen Modelle und die Zeiten der Messungen aus [Fer07] und [Bru06], die in den formalen Modellen aufgenommen wurden, angeführt. Weiters wurden in Kapitel 4.2 die Programme zur Erstellung und Verifikation von NCES Modellen anhand eines Beispiels kurz erläutert. Das Hauptaugenmerk galt aber den Kapiteln 4.3 und 4.4, wo zuerst die Realisierung in NCES der Modelle von eCos und dann jene von der μCrons Laufzeitumgebung im Detail erklärt wurden. Im letzten Kapitel wurde mittels eines Applikationsbeispiels das Zusammenspiel der formalen Modelle erklärt. Mit einem zeitlichen Verhalten und einiger Spezifikationsabfragen konnte anhand des Applikationsbeispiels eine Verifikation durchgeführt werden.

5 Zusammenfassung

Diese Arbeit hat sich mit der Realisierung von formalen Modellen eines Betriebssystems und einer Laufzeitumgebung befasst. Unter Verwendung dieser Modelle hat man die Möglichkeit, eine formale Verifikation für eine beliebige Anwendung auszuführen.

Für die Erstellung dieser Modelle wurde nach der Aufgabenstellung der Arbeit im Kapitel 2 ein Einblick zum aktuellen Stand der Technik wiedergegeben. Nach der Analyse von unterschiedlichen Modellierungsarten, Laufzeitumgebungen und Echtzeitbetriebssystem, wurde in nächsten Kapitel 3 speziell im Detail auf das Echtzeitbetriebssystem eCos und die μCorns Laufzeitumgebung eingegangen.

Nach einer genauen Analyse der Eigenschaften und Verhalten vom Echtzeitbetriebssystem eCos und der μCrons Laufzeitumgebung konnten die entsprechenden formalen Modelle erstellt werden. Dabei wurden die einzelnen Module als Aktivitätsdiagramm beschrieben und später als Implementierung in NCES dargestellt. Mit den formalen Modellen und der Implementierung von Abarbeitungszeit in den Modellen der beiden Elementen konnte dann das Zusammenspiel Betriebssystem und Laufzeitumgebung veranschaulicht werden.

Abschließend wurde mit den NCES Modellen im Kapitel 4 eine Anwendung beispielhaft realisiert. Unter Verwendung der Messungen für eCos und μCrons konnte mit diesen Zeiten in den formalen Modellen eine entsprechende Abarbeitungszeit implementiert werden, um ein reales Verhalten zu gewährleisten. Anhand der realisierten Anwendung konnte nun das Verhalten der NCES Modelle auf deren Korrektheit mittels Spezifikationsabfragen überprüft werden. Nicht nur das korrekte Verhalten der einzelnen Elemente konnte analysiert werden, sondern auch der zeitliche Ablauf der zusammenwirkenden Modelle.

6 Ausblick

Mit den erstellten NCES Modellen in dieser Arbeit können praktische Applikationen im Zusammenhang mit dem Echtzeitbetriebssystem eCos und der μCrons Laufzeitumgebung einer Verifikation unterzogen werden. Trotzdem sind noch Erweiterungen im Hinblick auf die NCES Modelle und deren Entwicklungsumgebung denkbar. Im folgenden werden die Erweiterungen in zwei Bereiche unterteilt.

Erweiterung der NCES Modelle

Das NCES Modell Event Dispatcher im Kapitel 3.4.1 kann in Hinblick auf die enthaltene Event Queue erweitert werden. In dieser Arbeit wurde die Event Queue für 30 verschiedene Eingangsereignisse und mit einer Ablage von 10 Ereignissen implementiert. Mit dieser Anzahl an Eingangsereignissen können kleinere Funktionsblocknetzwerke abgearbeitet werden. Für größere Funktionsblocknetzwerke ist jedoch eine gößere Event Queue erforderlich.
Im Bezug auf die Realisierung eines Funktionsblocknetzwerkes in NCES kann noch auf die Modellierung von weiteren Funktionsblöcken, die der Standard IEC 61499 zur Verfügung stellt, hingewiesen werden.
Betrachtet man die Abbildung 3.4 im Kapitel 3.2.2, so kann die Modellierung des External Event Managers realisiert werden. Im Kapitel 4.4 wurde zu Testzwecken ein Modell für den External Event Manager entwickelt, das aber nur nach einer gewissen Zeit ein Ereignis im Thread hineinstellt. Je nach Art der Interaktion mit der Umgebung der Applikation sind hier aufwändigere Modelle nötig.
Eine zusätzliche Erweiterung, die in die Modelle der μCrons Laufzeitumgebung einfließen könnte, ist die Eigenschaft der Rekonfigurierbarkeit.
Im Bezug auf das Echtzeitbetriebssystem eCos kann das Modell des Mutex erweitert werden. Derzeit wurde das NCES Modell des Mutex für drei externe Eingangsereignisse implementiert. Eine Erweiterung für mehrere externe Eingangsereignisse könnte realisiert werden. Weiters fehlt noch die Implementierung für ein gleichzeitiges Auftreten von zwei bzw. mehreren externen Ereignissen im Modell des Mutex. Hier gilt es, eine Abarbeitungsfolge der externen Ereignisse im NCES Modell zu erzeugen.

Das Echtzeitbetriebssystem eCos besitzt mehrere Elemente, die gegebenenfalls modelliert werden können. Im Zusammenhang mit dieser Arbeit wurden nur die wesentlichen Elemente von eCos modelliert.

Allgemeine Erweiterung im Hinblick auf die Umgebung der NCES Modelle

Für die Verifikation der Modelle diente das Programm ViVe (siehe Kapitel 4.2.2), das für Spezifikationsabfragen CTL Formeln und Prädikatenlogik der ersten Stufe ($\wedge$, $\vee$, $\neg$) verwendet. Da der Erreichbarkeitsgraph der erstellten Modelle anders ist als mit dem integrierten Model-Checker SESA, können somit die angeführten CTL Formeln aus dem Kapitel 2.2 mit diesem Programm nicht ausgeführt werden. Als Erweiterung für die Spezifikation könnte man das Programm soweit erweitern, dass es Abfragen von CTL Formeln im Zusammenhang mit den NCES Modellen gewährleistet.
Als weitere Erweiterung aus der Sicht der Umgebung der Modelle, könnte ein Algorithmus entwickelt werden, der die vorhandenen NCES Modelle verwendet, um daraus ein gewünschtes Endmodell automatisch zu erstellt. Dies könnte z. B. im Zusammenhang mit der Erstellung eines Schedulers geschehen oder eines Funktionsblocknetzwerkes. Mit diesem Algorithmus hätte man die Möglichkeit, schnell beliebige Applikationen zu erstellen und zu verifizieren.
Ein abschließender wichtiger Punkt sind noch die ausstehende Messungen des Echtzeitbetriebssystem eCos im Zusammenhang mit der μCrons Laufzeitumgebung. Diese Messungen könnten in den formalen Modellen der beiden Elemente einfließen, um ein realeres Verhalten wiederzuspiegeln.

A Anhang

Im Anhang sind die Elemente für die Erstellung von Aktivitätsdiagramme aufgelistet, die in den Abbildungen von Kapitel 3.3 und 3.4 verwendet wurden. Weiters werden die wichtigsten NCES Modelle, die im Kapitel 4.3 nicht angeführt wurden, dargestellt. Die restlichen NCES Modelle sind in der beiliegenden CD angeführt.

A.1 Elemente im Aktivitätsdiagramm

Im Folgenden soll kurz auf die einzelnen Elemente eines Aktivitätsdiagrammes eingegangen werden, die aus [UML] entnommen wurden.

Abbildung A.1: Aktionszustand

Der Aktionszustand stellt eine Art von Zustand dar, der eine interne Aktion und mindestens einen ausgehenden Übergang besitzt, der die expliziten Ereignisse einbezieht, die die interne Aktion vervollständigen.

Abbildung A.2: Objektknoten

Befindet sich ein Objektknoten (Datenspeicher) zwischen zwei Aktionen, dann transportiert er von der ausgehenden Aktion Daten in den Speicher und von dort in die nächste Aktion.

Abbildung A.3: Anfangszustand

Der Anfangszustand stellt den Zustand eines Objekts dar, bevor in dem Aktivitätsdiagramm vorhandene Ereignisse auf das Objekt eingewirkt haben.

Abbildung A.4: Endzustand

Der Endzustand stellt den Abschluss von einer Aktivität in dem umschließenden Zustand oder dem Aktionsstatus dar.

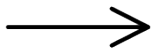


Abbildung A.5: Kontrollfluss

Der Kontrollfluss stellt als Übergang in einem Aktivitätsdiagramm eine Beziehung zwischen zwei Zuständen oder Aktionszuständen dar und zeigt an, dass ein Objekt, das sich im ersten Zustand befindet, in den zweiten Zustand übergeht und festgelegte Aktionen durchführt.

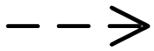
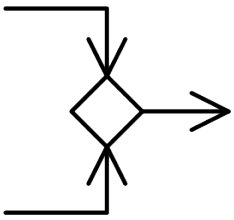


Abbildung A.6: Objektfluss

Der Objektfluss zeigt an, dass ein Objekt die Eingabe für eine Aktion bzw. deren Ausgabe ist.

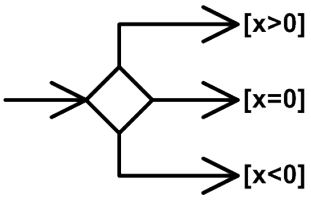
Abbildung A.7: Entscheidung

Eine Entscheidung muss einen oder mehr eingehende und zwei oder mehr ausgehende Übergänge besitzen, von denen jeder mit einer eigenen Bedingung beschriftet sein muss.



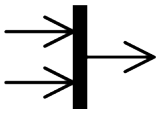
Bei dieser Entscheidung handelt es sich um eine ODER Verknüpfung.

Abbildung A.8: Zusammenführung



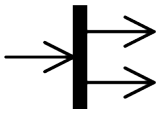
Bei dieser Entscheidung handelt es sich um eine Verzweigung. Mit den Bedingungen $[x > 0]$, $[x < 0]$ und $[x = 0]$ werden unterschiedliche Übergänge erreicht.

Abbildung A.9: Verzweigung



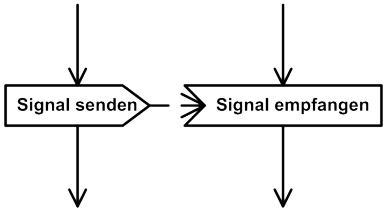
Bei der Synchronisation handelt es sich um eine UND Verknüpfung, Ströme werden zusammengeführt.

Abbildung A.10: Synchronisation



Beim Splitting wird der Kontrollfluss in mehrere Ströme aufgeteilt.

Abbildung A.11: Splitting



Eine Signalsendung kann verwendet werden, um eine Aktion weiterzuleiten. Mit der Siganbestätigung kann die Aktion aufgenommen und verarbeitet werden.

Abbildung A.12: Signale

A.2 NCES Modelle

In den Folgenden Kapiteln sind die NCES Modelle des Echtzeitbetriebssystem eCos aus Kapitel 4.3.1 und der μCrons Laufzeitumgebung aus Kapitel 4.3.2 angeführt.

A.2.1 eCos Echtzeitbetriebssystem

Die Auflistung der folgenden NCES Modelle betreffen dem MLQ Scheduler erster Priorität. Die Realisierung der restlichen Prioritäten für Bitmap und MLQ Scheduler sind in der beiliegenden CD zu finden.

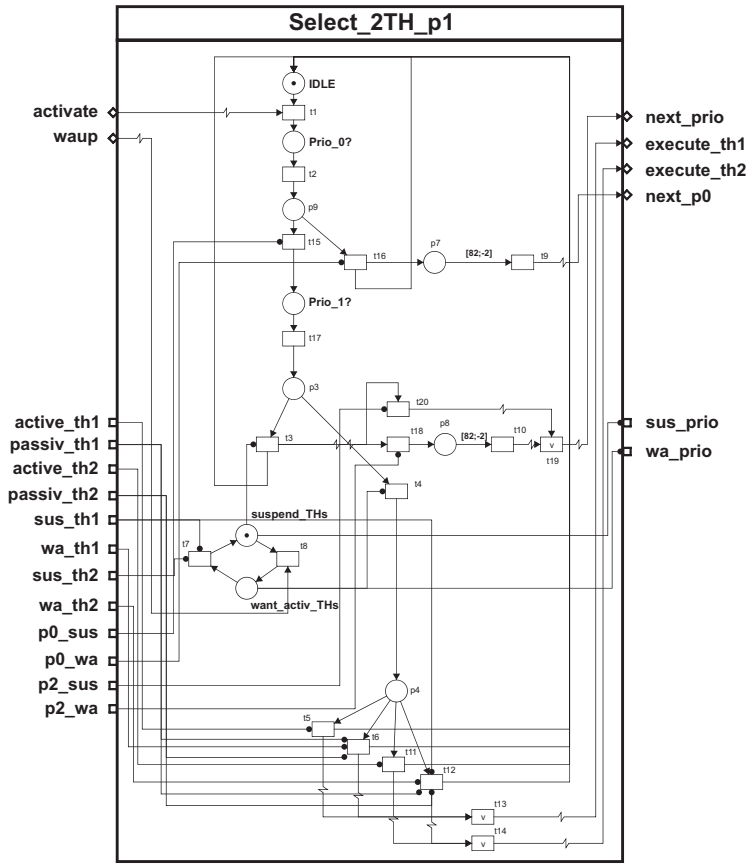


Abbildung A.13: NCES Modell Select_2TH_p1 für die Prioritäts/Thread Entscheidung im MLQ Scheduler

Das NCES Modell **Select_2TH_p1** in Abbildung A.13 entscheidet über einer Schrittkette für die Priorität 1 welcher von den zwei Threads, die am MLQ Scheduler angeführt sind, abgearbeitet wird. Weiters wird bestimmt welche Priorität abgearbeitet wird und zustäzlich wird nach außen hin der Status der Priorität 1 bekanntgegeben.

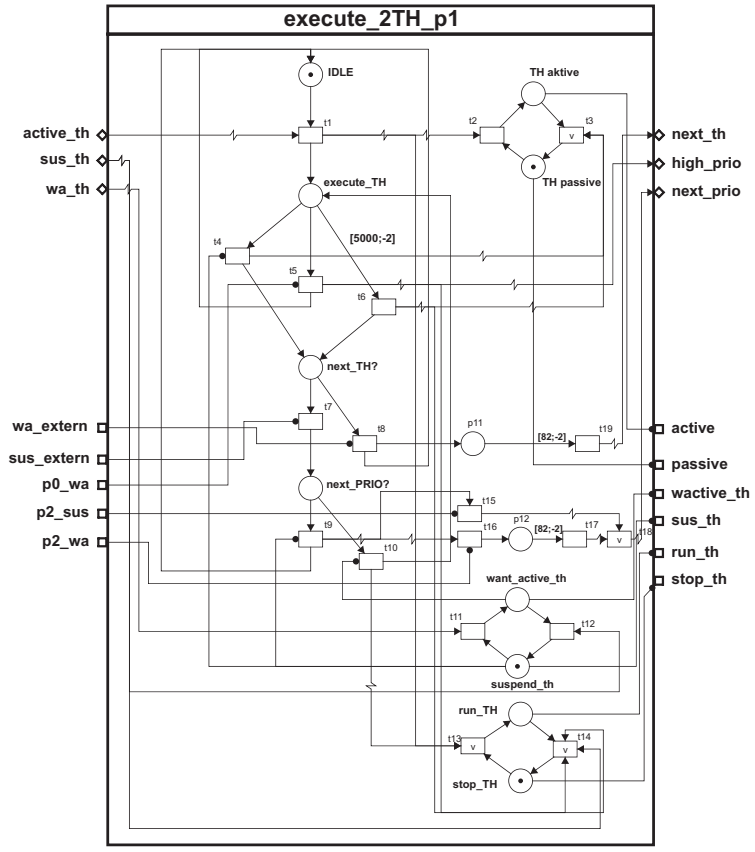


Abbildung A.14: NCES Modell Execute_2TH_p1 für die Ausführung eines Threads im MLQ Scheduler

Mit dem NCES Modell **Execute_2TH_p1** in Abbildung A.14 wird der Thread am MLQ Scheduler erster Priorität an der Stelle **execute_TH** aktiviert und abgearbeitet. Hier kann der Thread von einem höherprioren Thread, von sich selbst oder durch das Timeslicing (in diesem Fall nach einer Zeit **[5000;-2]**) unterbrochen werden. Weiters wird bei einer Unterbrechung entweder auf den nächsten Thread oder Priorität verzweigt.

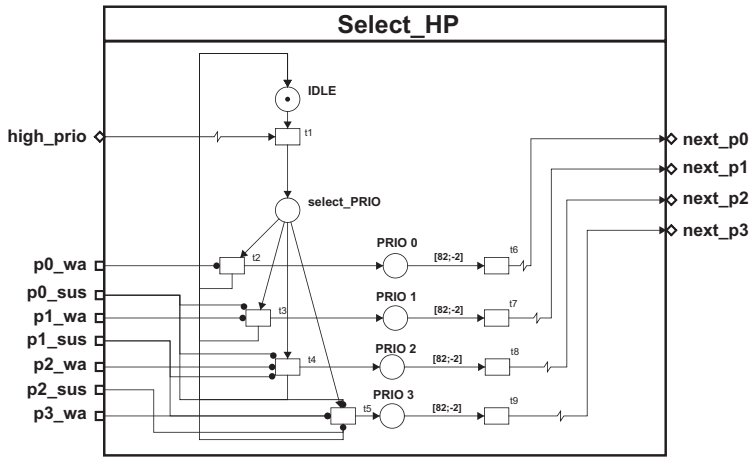


Abbildung A.15: NCES Modell Select_HP für die Verzweigung zu einer höheren Priorität im MLQ Scheduler

Sobald ein höherpriorer Thread abgearbeitet werden möchte, wird über das NCES Modell **Select_HP** in Abbildung A.15 auf die zugehörige Priorität verwiesen. In diesem Modell ist weiters noch die Kontextwechselzeit implementiert.

A.2.2 μCrons Laufzeitumgebung

Die NCES Modelle der μCrons Laufzeitumgebung die als nächstes angeführt sind, stellen die wichtigsten Elemente dar. Die Realisierung der restlichen NCES Module z. B. jene die im Modell des Event Dispatcher enthalten sind, sind in der beiliegenden CD zu finden.

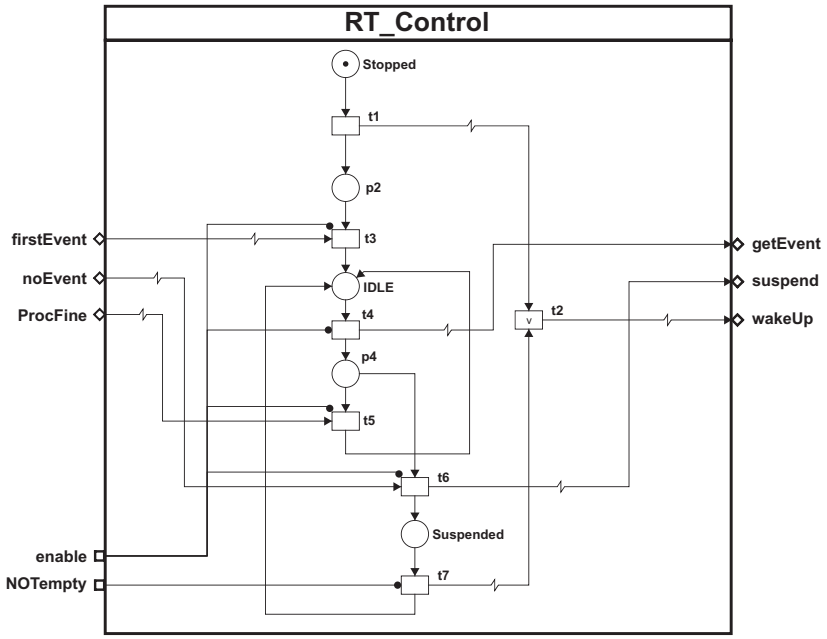


Abbildung A.16: NCES Modell RT_Control für die Kontrolle für einen Thread mit Funktionsblocknetzwerk

Mit dem NCES Modell **RT_Control** in Abbildung A.16 wird bestimmt ob der Thread abgearbeitet werden soll oder nicht. Die Implementierung dieses Modells wurde wieder über einer Schrittkette realisiert, jedoch wird beim eintreffen der Markierung in der Stelle **IDLE** eine Schleife ausgeführt, die im Zusammenhang mit dem Modul Event Dispatcher eine Kontrolle über mögliche Ereignisse die in der Event-Queue durchführt.

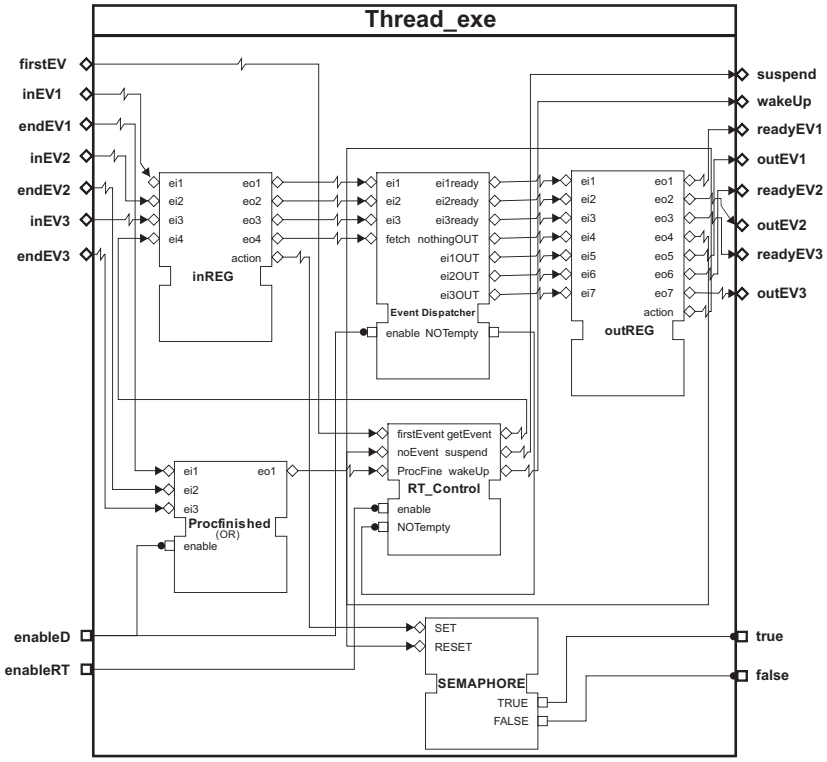


Abbildung A.17: NCES Modell Laufzeitumgebung

Das NCES Modell **Thread_exe** ist ein kombiniertes Modell bestehend aus den NCES Modellen **Event_Dispatcher**, **RT_Control**, **SEMAPHORE**, **inREG**, **outREG** und **Procfinished** vom Typ ODER.

Literaturverzeichnis

[Abe90] D. Abel. *Petri-Netze für Ingenieure*. Springer Verlag, Berlin, 1990.

[Abe98] D. Abel. *Theorie ereignisdiskreter Systeme, Tutorium des GMA-Fachausschusses 1.8 „Methoden der Steuerungstechnik“*. R. Oldenburg Verlag, München Wien, 1998.

[Bau96] B. Baumgarten. *Petrinetze, Grundlagen und Anwendungen*. Spektrum Akademischer Verlag GmbH, Heidelberg, 2. edition, 1996.

[Bey02] Dirk Beyer. *Formale Verifikation von Realzeit-Systemen mittels Cottbus Timed*. Phd report, TU Cottbus, 2002.

[Bru06] Jeroen Brunnenkreef. Design and implementation of a demonstrator/testbed for ipmcs with respect to physical and logical reconfiguration of mechatronical parts. Technical report, External training at Profactor GmbH, Steyr, Austria, 2006.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time logic. In *Proc. Workshop on Logic of Programs*, volume 131, pages 52–71. Springer-Verlag, 1981.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The {MIT} Press, Cambridge, Massachusetts, 1999.

[CLA06] Goran Cengic, Oscar Ljungkrantz, and Knut Akesson. Formal Modeling of Funktion Block Applications Running in IEC 61499 Execution Runtime. In *Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA'06)*, pages 1269–1276, 2006.

[CS01] Edmund M. Clarke and B.-H. Schlingloff. Model Checking. In *ed. in Handbook of Automated Reasoning*, pages 1369–1468. Elsevier Science Publishers B. V., 2001.

[CWea] Edmund M. Clarke, J. M. Wing, and et al. Formal methods: State of the art and future directions. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh.

[DT05] George S. Doukas and Kleanthis C. Thramboulidis. A Real-Time Linux Execution Evironment for Funktion-Block Based Distributed Control Applications. In *Proceedings of the 3rd IEEE Int. Conferencs on Industrial Informatics (INDIN'05)*, pages 56–61, 2005.

[FB04] B. Favre-Bulle. *Automatisierung komplexer Industrieprozesse, Systeme, Verfahren und Informationsmanagement.* Springer Verlag, Wien New York, 2004.

[FBZ05] B. Favre-Bulle and G. Zeichen. Zukunft der Forschung in den Produktionswissenschaften. Technical report, Technische Universität Wien, 2005.

[Fer07] Tarik Ferhatbegovic. Echtzeitbetriebssysteme - Vergleich. Technical report, Institut für Automatisierungs- und Regelungstechnik (IFAT), TU Wien, 2007.

[HUB] Modelling and Verification of Execution control of the Function Blocks following to the standard IEC 61499 by means of Net Condition/Event Systems (NCES). Technical report, Institut für Automatisierungstechnik (IFAT), Humboldt-Universität Berlin.

[HW] N. Haage and B. Wagner. A new Function Block Modeling Language Based on Petri Nets for Automatic Code Generation. In *IEEE Transactions on Industrial Informatics*, volume 1 of *4*, pages 226–237.

[IEC04a] IEC 61499-2. *Function blocks – Part 2: Software tool requirements.* International Electrotechnical Commission, Geneva, 2004.

[IEC04b] IEC 61499-3. Function blocks for industrial-process measurement and control systems – Part 3: Tutorial information. Technical report, International Electrotechnical Commission, Geneva, 2004.

[IEC05a] IEC 61499-1. *Function blocks – Part 1: Architecture.* International Electrotechnical Commission, Geneva, 2005.

[IEC05b] IEC 61499-4. *Function Blocks – Part 4: Rules for compliance profiles.* International Electrotechnical Commission, Geneva, 2005.

[ISA] http://www.isagraf.com/, ISaGRAF Website.

[Kro97] Thomas Kropf. *Formal Methods of Hardware Verification.* Springer Verlag, Berlin, 1997.

[Kro05] M. Kropik. Distributed Automation in Automative Manufacturing - Current Status and Strategies. Technical report, 18th International cooperation symposium industry-resarch, Vienna, Austria, 2005.

[Lew98] R. W. Lewis. *Programming industrial control systems using IEC 61131-3.* The Institution of Electrical Engineers, London, United Kingdom, revised edition, 1998.

[Lew01] R. W. Lewis. *Modelling control systems using IEC 61499.* The Institution of Electrical Engineers, London, United Kingdom, 1st edition, 2001.

[LGLT] Jose L. Martinez Lastra, Luis Godinho, Andrei Lobov, and Reijo Tuokko. An IEC 61499 Application Generator for Scan-Based Industrial Controllers. Technical report, Institute of Production Engineering, Tampere University of Technology, Tampere, Finland.

[Mas03] Anthony J. Massa. *Embedded Software Development with eCos.* Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.

[MYC] http://www.microns.org/, Microns Website.

[Pel01] Doron A. Peled. *Software Reliability Methods.* Springer-Verlag, New York, 2001.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[SFL02] C. Schnakenbourg, J.-M. Faure, and J.-J. Lesage. Towards IEC 61499 function blocks diagrams verification. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 3, 2002.

[SR02] P.-H. Starke and S. Roch. *Analysing Signal Net Systems.* Humbolt-Universität, Berlin, 2002.

[Sta05] M. Stanica. *Behavioral Modeling of IEC 61499 Control Applications.* Phd report, Universite de Rennes, 2005.

[Tan03] Andrew S. Tanenbaum. *Moderne Betriebssysteme.* Pearson Studium, München, 2003.

[UML] http://www.oose.de/downloads/uml-2-Notationsuebersicht-oose.de.pdf, UML2.

[VH99] V. Vyatkin and H.-M. Hanisch. A modeling approach for verification of IEC 61499 function blocks using net condition/event systems. In *Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA'99)*, pages 261–270, 1999.

[Vyaa] Valeriy Vyatkin. http://www.fb61499.com/valid.html, http://www.ece.auckland.ac.nz/%7Evyatkin/tools/modelchekers.html Website.

[Vyab] Valeriy Vyatkin. Modelling and Validation of Industrial Automation Systems, (Web Resources).

[Vya06] V. Vyatkin. Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN'06)*, pages 874–879, 2006.

[WEBa] www.qnx.com, QNX Website.

[WEBb] www.rtai.org, RTAI Website.

[WEBc] www.ucos-ii.org, MicroC/OS-II Website.

[WW00] H. Wurmus and B. Wagner. IEC 61499 konforme Beschreibung verteilter Steuerungen mit Petri-Netzen. Technical report, Fachtagung Verteilte Automatisierung, 2000.

[ZDH04] W. Zhang, C. Diedrich, and W. A. Halang. Module and Integration Verification for Function Block-based Safety-related System Development. In *Proceedings 2nd Intl. Conf. on Industrial Informatics (INDIN'04)*, pages 210–215, Berlin, Germany, 2004.

[Zoi02] Alois Zoitl. Development of an IEC 61499 based embedded control platform and integration in a distributed automation system. Diplomarbeit, TU Wien, September 2002.