(19) Deutsches Patent- und Markenamt

(10) **DE 10 2009 027 627 B3** 2011.03.17

(12) # Patentschrift

(21) Aktenzeichen: **10 2009 027 627.0**
(22) Anmeldetag: **10.07.2009**
(43) Offenlegungstag: **–**
(45) Veröffentlichungstag der Patenterteilung: **17.03.2011**

(51) Int Cl.[8]: ***G05B 17/02*** (2006.01)

Innerhalb von drei Monaten nach Veröffentlichung der Patenterteilung kann nach § 59 Patentgesetz gegen das Patent Einspruch erhoben werden. Der Einspruch ist schriftlich zu erklären und zu begründen. Innerhalb der Einspruchsfrist ist eine Einspruchsgebühr in Höhe von 200 Euro zu entrichten(§ 6 Patentkostengesetz in Verbindung mit der Anlage zu § 2 Abs. 1 Patentkostengesetz).

(73) Patentinhaber:
Wolfgang Pree GmbH, Salzburg, AT

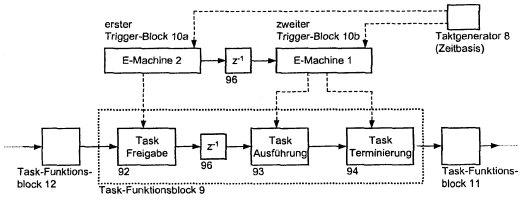
(74) Vertreter:
Westphal, Mussgnug & Partner, 80331 München

(72) Erfinder:
Pree, Wolfgang, Salzburg, AT; Templ, Josef, Leonding, AT; Naderlinger, Andreas, Salzburg, AT

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:
siehe Folgeseiten

(54) Bezeichnung: **Simulation von Echtzeit-Software-Komponenten auf Basis der Logischen Ausführungszeit**

(57) Zusammenfassung: Ein Beispiel der Erfindung bezieht sich auf ein System zum Simulieren eines Echtzeitsystems mittels einer blockorientierten Simulation mit statischer Ausführungsreihenfolge der Blöcke. Das System besteht aus: einem Taktgenerator, welcher die Zeitbasis für die Simulation bereitstellt; einem ersten Task-Funktionsblock, der einen Task-Freigabeblock, einen Task-Ausführungsblock, einen Task-Terminierungsblock und einen Delay-Block enthält; einem ersten Trigger-Block der konfiguriert ist, den Task-Freigabeblock zu einem ersten, von dem Taktgenerator bereitgestellten, Zeitpunkt zu triggern; einem zweiten Trigger-Block der konfiguriert ist, den Task-Terminierungsblock zu einem dritten, von dem Taktgenerator bereitgestellten, Zeitpunkt zu triggern; der Task-Ausführungsblock ist konfiguriert, abhängig von einem Eingangswert in Übereinstimmung mit der gewünschten Transferfunktion, einen Ausgangswert zu berechnen. Der Task-Freigabeblock ist konfiguriert, Eingangsdaten von einer Datenquelle zu empfangen und den Eingangswert des Task-Ausführungsblocks gemäß empfangener Daten zu setzen. Der Task-Terminierungsblock ist konfiguriert, den Ausgangswert eines Task-Ausführungsblocks als Ausgangsdaten für eine Datensenke bereitzustellen. Der erste Trigger-Block oder der zweite Trigger-Block ist konfiguriert, den Task-Ausführungsblock zu einem zweiten, von dem Taktgenerator bereitgestellten, Zeitpunkt zu triggern. Der Delay-Block ist zwischen Task-Freigabeblock und Task-Terminierungsblock ...



(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:

T. Henzinger, B. Horowitz, and C. Kirsch, Giotto: A time-triggered language for embedded programming, Proceedings of the IEEE, Vol. 91, Issue 1, Jänner 2003, S. 84-99

T.A. Henzinger and C.M. Kirsch: The embedded machine: predictable, portable real-time code, Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, S. 315-326

J. Templ: Timing Definition Language (TDL) 1.5 Specification, Technical report, University of Salzburg, 2009, Verfügbar unter http://www.softwareresearch.net

G. Stieglbauer: Model-based Development of Embedded Control Software with TDL and Simulink, Doktorarbeit, Universität Salzburg, 2007

Hyperperiod Bus Scheduling and Optimizations for TDL Components by E. Farcas, W. Pree in Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Patras, Greece. IEEE, Sep. 2007

Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language (TDL) by A. Naderlinger, J. Pletzer, W. Pree, J. Templ, 4th International ICSE workshop on Software Engineering for Automotive Systems, Minneapolis, 26 May 2007

Simulation of LET Models in Simulink and Ptolemy by P. Derler, A. Naderlinger, W. Pree, S. Resmerita, J. Templ, Monterey Workshop 2008, Budapest, Sept. 24-26, 2008

Beschreibung

[0001] Die vorliegende Erfindung bezieht sich auf die Simulation von Systemen, die strenge Echtzeitanforderungen erfüllen müssen (Echtzeit-Systeme), insbesondere auf die Simulation von Echtzeitsystemen, die auf Echtzeit-Software-Komponenten aufbauen, die das Konzept der Logischen Ausführungszeit (= Logical Execution Time = LET) verwenden.

[0002] Echtzeitsysteme werden oft bei Regelungssystemen eingesetzt, um Regelungsgesetze zur Steuerung technischer Prozesse zu implementieren. In vielen Anwendungen wird bei diesen Echtzeitsystemen verteilte Hardware verwendet, wodurch die Software für die diversen (Berechnungs-)Aufgaben (= Tasks), die für die Steuerungszwecke nötig sind, auf getrennten Prozessoren ausgeführt wird. Üblicherweise besteht ein System zur Ausführung verteilter Software aus einer Vielzahl von Knoten und zumindest einem Kommunikationskanal, wobei das System so konfiguriert ist, dass die Knoten Daten über den Kommunikationskanal übertragen können. Beispiele für solche Systeme sind auch so genannte eingebettete Systeme (= Embedded Systems), in denen die Knoten als Electronic Control Units (abgekürzt ECUS) bezeichnet werden. Eine ECU kann Software-Tasks ausführen und ist möglicherweise in das Gerät, das es steuert, integriert.

[0003] Zu eingebetteten Systemen gehören unter anderem Automobilsysteme, Automatisierungssysteme, Eisenbahnsysteme und Systeme in Flugzeugen. Ein Automobilsystem besteht zum Beispiel aus mehreren Geräten zur Betätigung von Bremsen, mehreren Geräten zur Messung der Rad-Umdrehungsgeschwindigkeiten, einem Gerät zum Messen der Fahrzeuggeschwindigkeit etc., die über einen Kommunikationskanal kommunizieren und die zum Beispiel so konfiguriert sind, dass sie die Funktion eines Antiblockiersystems (ABS) erfüllen. Da der Betrieb eines Antiblockiersystems sicherheitskritisch für das Fahrzeug und seine Insassen ist, müssen die Sensoren periodisch gelesen werden, ebenso wie die Task-Berechnungen und Aktor-Aktualisierungen periodisch vorgenommen werden müssen, zum Beispiel alle fünf Millisekunden. In der Praxis muss ein derartiges System strikte Echtzeitanforderungen erfüllen, was bedeutet, dass die Korrektheit der Funktion nicht nur von der logischen Korrektheit der Funktion sondern auch von der Zeit, zu der sie ausgeführt wird, abhängt. Eine Funktion, die später als zu einem Zeitpunkt, der innerhalb des Systems als der spätest mögliche definiert wurde (= Deadline), ist per definitionem falsch und hat üblicherweise keinen Wert. Das bedeutet, dass das Regelungssystem die Erfüllung vordefinierter Zeitanforderungen garantieren muss.

[0004] Konventionelle Software, die für Echtzeitsysteme entworfen wurde, ist typischerweise so konfiguriert, dass die Software in mehrere Tasks aufgeteilt ist, die das System auszuführen hat. Diese Tasks können von einer ECU (also einem Knoten), oder von verschiedenen Knoten ausgeführt werden, wobei jeder einzelne Knoten eine oder mehrere Tasks ausführen kann Tasks erhalten eventuell die Ausgabesignale von Sensoren als ihre Eingabe und stellen eventuell Ausgabesignale an Aktoren bereit. Verschiedene Tasks kommunizieren eventuell untereinander durch Datenaustausch. Daher hängen ein Ausführungsplan (= Schedule) und die Ausführung von Tasks eventuell von externen Ereignissen (= Events) ab, die vom System durch einen oder mehrere Sensoren erfasst werden. Daher kann sich auch ein Betriebszustand (= Mode of Operation) irgend eines Systems auf irgend einem Knoten im Lauf der Zeit ändern, so wie sich auch die Anforderungen an den Kommunikationskanal betreffend Bandbreite im Lauf der Zeit ändern können. In einem harten Echtzeitsystem muss jedoch sichergestellt sein, dass eine gegebene Bandbreite eines Kommunikationskanals ausreicht, um ein fehlerfreies Funktionieren eines harten Echtzeitsystems in jeder Kombination von Betriebszuständen auf allen Knoten zu garantieren.

[0005] Es ist auf diesem Gebiet bekannt, dass die Entwicklung absolut korrekter eingebetteter Systeme mit harten Echtzeitanforderungen schwierig, fehleranfällig und daher teuer ist. Das gilt sowohl für Einzel-Knoten-Systeme als auch für verteilte eingebettete Systeme.

[0006] Verschiedene Anstrengungen wurden bereits unternommen, um den Entwurf und die Entwicklung von potentiell verteilten eingebetteten Systemen zu verbessern. Das ”Giotto” genannte Projekt an der Universität von Kalifornien in Berkeley, USA, lieferte zum Beispiel eine Programmiermethode für eingebettete Regelungssysteme. Diese Methode beinhaltet ein Konzept zur Definition einer logischen Ausführungszeit von Tasks unter harten Echtzeitbedingungen. Das Konzept wird als ”LET” (= Logical Execution Time) bezeichnet und detaillierter in folgender Publikation von T. A. Henzinger et al. beschrieben: T. A. Henzinger et al., ”Giotto: A time-triggered language for embedded programming”, Proceedings of the IEEE, Vol. 91, Ausgabe (= Issue) 1, Jänner 2003, Seiten 94–99. Giotto bietet eine Programmierabstraktion für harte Echtzeitanwendungen, die ein periodisches, multi-modales Verhalten haben, wie es im Automobil-, Flugzeug-, Raumfahrts- und Fertigungsbereich vorkommt. Traditionell erfolgt der Entwurf von Reglern auf einer mathematischen Abstraktionsstufe, auf der ein Regelungstechniker (= Control Engineer) Differentialgleichungen und Zustandswechsel-Logik mit Werkzeugen wie MATLAB/Simulink, LabView oder MatrixX editiert. Zu den typischen Tätigkeiten eines Regelungstechnikers gehören die Modellierung des Verhaltens einer Anlage (= Plant) und von deren Störungen (= Disturbances), die Ableitung und Optimie-

rung der Regelungsgesetze, sowie die Validierung der Funktionalität und Performanz des Modells durch Analyse und Simulation. Wenn der validierte Entwurf in Software umgesetzt werden muss, wird es [das Modell] an einen Software-Techniker weitergegeben, der den Code für die jeweilige Plattform schreibt (mit ”Plattform” ist hier die Hardware-Konfiguration, also entweder ein einzelner Knoten oder ein verteiltes System, zusammen mit einem Echtzeit-Betriebssystem gemeint).

[0007] Zu den typischen Tätigkeiten eines Software-Technikers gehören die Zerlegung der benötigten Berechnungen in periodische Tasks, die Zuweisung dieser Tasks auf CPUs (Central Processing Units) und die Festlegung von Task-Prioritäten, um die erwünschten harten Echtzeit-Einschränkungen erfüllen zu können, und zwar bei einem gegebenen Abarbeitungsplan (= Scheduling Mechanism) und gegebener Hardware-Performanz, sowie das Erreichen eines Grades an Fehlertoleranz durch Replizierung und Fehlerkorrektur.

[0008] Giotto bietet eine mittlere Abstraktionsstufe, die eine effektivere Kommunikation zwischen Software-Techniker und Regelungstechniker erlaubt. Spezifischer ausgedrückt, wird eine Software-Architektur der Implementierung definiert, indem die Funktionalität und das Zeitverhalten spezifiziert werden. Funktionalität und das Zeitverhalten sind ausreichend und notwendig, um sicherzustellen, dass die Implementierung mit dem mathematischen Modell des regeltechnischen Entwurfs konsistent ist. ”Correct-by-construction development” bedeutet, dass die Implementierung eines eingebetteten Systems, das exakt seiner Spezifikation entspricht, automatisch generiert wird. Das erlaubt die Abstraktion von der Implementierung der Software-Architektur auf einer spezifischen Plattform, und befreit den Software-Techniker davon, sich Gedanken über Dinge wie die Hardware-Performanz und das Scheduling-Verfahren machen zu müssen, wenn er mit dem Regelungstechniker kommuniziert. Nachdem ein Giotto-Programm erstellt wurde, ist nach wie vor die zweite Aufgabe des Software-Technikers, das Programm auf einer bestimmten Plattform zu implementieren. In Giotto ist jedoch diese zweite Aufgabe, die keiner Interaktion mit dem Regelungstechniker bedarf, klar von der ersten Aufgabe getrennt, und kann durch zunehmend mächtigere Generatoren (= Compiler) automatisiert werden. Die Trennung zwischen logischen Korrektheitsbelangen (Funktionalität und Zeitverhalten) von der physischen Umsetzungsbelangen (Abbildung und Scheduling) hat den zusätzlichen Vorteil, dass die Definition des Zeitverhaltens komplett plattformunabhängig ist und für verschiedene, sogar heterogene Plattformen generiert werden kann.

[0009] Wolfgang Pree und sein Team haben in einem ad personam Forschungsprojekt an der Paris Lodron Universität Salzburg, Österreich, eine Sprache entwickelt, die teilweise auf den Giotto-Sprachkonzepten zur Spezifikation des Zeitverhaltens von verteilter Software basiert. Auf diese Sprache beziehen wir uns als ”TDL” (Timing Definition Language; Zeit-Definitions-Sprache) welche im Bericht von Josef Templ (siehe J. Templ, Timing Definition Language (TDL) 1.5 Specification, Technical Report, Universität Salzburg, 2009) definiert ist. Die Publikation E. Farcas and W. Pree: ”Hyperperiod Bus Scheduling and Optimizations for TDL Components”, in: Proceedings of the 12th IEEE Conference an Emerging Technologies and Factory Automation (ETFA), Patras, Greece. IEEE, S. 1262–1269, 2007, beschreibt ein TDL-Komponentenmodell und eine so genannte ”Tool-Chain” als Lösung zur Generierung von zeitkritischen TDL-Software-Komponenten, welche unabhängig voneinander entwickelt und in eine verteilte Plattform integriert werden können werden können, ohne das von außen beobachtbare Zeitverhalten zu verändern oder die Echtzeit-Software-Komponenten adaptieren zu müssen. Modellgetriebene Entwicklung ist ein Oberbegriff für Techniken, die automatisiert, aus formalen Modellen lauffähige Software erzeugen. Die Publikation A. Naderlinger, J. Pletzer, W. Pree, J. Templ: ”Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language (TDL)”, in: Proceedings of the 4th International Workshop an Software Engineering for Automotive Systems, S. 6ff., 2007, beschreibt eine plattformneutrale Systementwicklung mit Hilfe einer Systemmodellierung mittels TDL.

[0010] Echtzeit-Software-Komponenten, die auf dem Konzept der Logischen Ausführungszeit (= Logical Execution Time = LET) beruhen, zeigen ein äquivalentes beobachtbares Verhalten, unabhängig von der Ausführungsplattform beziehungsweise von der Simulationsumgebung. Daher stellt das LET-Konzept eine perfekte Übereinstimmung zwischen Simulation und Ausführung auf einer (möglicherweise verteilten) Hardware sicher, ohne dass plattformspezifische Details bereits im Anwendungsmodell berücksichtigt werden müssen.

[0011] Speziell für komplexe multi-modale (= multi-mode) Systeme mit unterschiedlichen [Task-]Ausführungs-Perioden (= multi-rate) kann eine virtuelle Maschine (VM) ein geeigneter Ansatz sein, um das korrekte Zeitverhalten sicher zu stellen. Simulationsumgebungen, wie z. B. Matlab/Simulink (siehe z. B. P. Derler, A. Naderlinger, W. Pree, R. Resmerita, J. Templ: ”Simulation of LET Models in Simulink and Ptolemy”, in: Monterey Workshop 2008, Budapest, Sept. 24–26, 2008), verfügen typischerweise über einen Auslösemechanismus (= Trigger Mechanism), der die Implementierung einer solchen VM erlaubt. Diese Offenbarung diskutiert die Probleme von Datenabhängigkeiten, die bei der Simulation von LET-basierten Komponenten auftreten können und die

die Anwendbarkeit bestehender Ansätze in der Praxis beträchtlich einschränken können. Die identifizierten Einschränkungen betreffen Komponenten, die zyklischen Datenfluss aufweisen, Regelungsschleifen, die eine Anlage (= Plant) ohne Verzögerung enthalten, und die Kombination von LET-basierten und konventionellen Komponenten.

[0012] Es ist ein Ziel der voriegenden Erfindung, ein System und ein Verfahren für die Simulation LET-basierter Komponenten bereit zu stellen, das nicht die oben erwähnten Einschränkungen hat.

[0013] Das oben angeführte Ziel wird durch das System von Anspruch 1 und das Verfahren von Anspruch 6 erreicht. Verschiedene beispielhafte Ausprägungen der Erfindung werden in voneinander abhängigen Ansprüchen behandelt.

[0014] Ein Beispiel für die Erfindung bezieht sich auf ein System zur Simulation eines Echtzeitsystems, bei dem eine Blockorientierte Simulation mit einer statischen Block-Aktualisierungsreihenfolge benutzt wird. Das System besteht aus: einer Uhr (= clock, Taktgenerator), die die Zeitbasis für die Simulation bildet; einem ersten Task-Funktionsblock, der einen Task-Freigabeblock, einen Task-Ausführungsblock, einen Task-Beendigungsblock (= Task-Terminierungsblock) und einen Verzögerungsblock enthält; einem ersten Auslöser (= Trigger)-Block, der so konfiguriert ist, dass er den Task-Beendigungsblock zum Zeitpunkt 3 laut Uhr anstößt. Der Task-Ausführungsblock ist so konfiguriert, dass er einen Ausgabewert abhängig vom Eingabewert entsprechend einer erwünschten Übergangsfunktion (= Transfer Function) berechnet. Der Task-Freigabeblock ist so konfiguriert, dass er Eingabedaten von einer Datenquelle erhält, und den Eingabewert des Task-Ausführungsblocks abhängig von den empfangenen Daten setzt. Der Task-Beendigungsblock ist so konfiguriert, dass er den Ausgabewert des Task-Ausführungsblocks an eine Datensenke weitergibt. Der erste Trigger-Funktionsblock oder der zweite Trigger-Funktionsblock ist so konfiguriert, dass er den Task-Ausführungsblock zum Zeitpunkt 2 laut Uhr anstösst. Der Verzögerungsblock ist zwischen dem Task-Freigabeblock und dem Task-Beendigungsblock angeordnet.

[0015] In einem Beispiel der Erfindung sind Zeitpunkt 1 und Zeitpunkt 3 aufeinanderfolgende Zeitpunkte. Die Zeit, die zwischen Zeitpunkt 1 und Zeitpunkt 3 vergeht, ist gleich der logischen Ausführungszeit (= Logical Execution Time = LET) des Task-Funktionsblocks, und der Zeitpunkt 2 liegt entweder im Intervall zwischen Zeitpunkt 1 und Zeitpunkt 3 oder ist gleich dem Zeitpunkt 1.

[0016] Die vorigen als auch die anderen vorteilhaften Eigenschaften der Erfindung werden durch die nachfolgende detaillierte Beschreibung beispielhafter Ausprägungen der Erfindung zusammen mit einem Verweis auf die zugehörigen Zeichnungen offensichtlicher. Es wird darauf hingewiesen, dass nicht alle möglichen Ausprägungen der vorliegenden Erfindung notwendigerweise jeden, oder auch nur irgendeinen der identifizierten Vorteile zeigen.

[0017] Abb. 1 zeigt eine schematische Darstellung, die ein System zur Ausführung verteilter Software illustriert

[0018] Abb. 2 ist eine schematische Darstellung, die Beispiel-Module zeigt

[0019] Abb. 3 ist eine schematische Darstellung, die das Konzept der logischen Ausführungszeit (= Logical Execution Time, LET) illustriert

[0020] Abb. 4 ist eine schematische Darstellung, die die parallele Ausführung verschiedener Tasks auf einem Knoten illustriert

[0021] Abb. 5 ist eine schematische Darstellung, die das LET-Verhalten einer Task mit Hilfe von Standard-Simulationsblöcken illustriert

[0022] Abb. 6 ist eine schematische Darstellung, die das Konzept von einer E-Maschine pro Modul in einer Simulationsumgebung illustriert

[0023] Abb. 7 ist eine schematische Darstellung, die ein Modell vor (a) und nach (b) der Migration eines Reglers c nach TDL illustriert

[0024] Abb. 8 ist eine schematische Darstellung, die die Trennung von Triggern in Schritte illustriert

[0025] Abb. 9 ist eine schematische Darstellung, die ein E-Maschinen-Paar illustriert

[0026] Abb. 10 ist eine weitere schematische Darstellung, die ein E-Maschinen-Paar allgemeiner illustriert Ähnliche Komponenten sind in den Zeichnungen mit dem gleichen Referenzsymbol versehen.

[0027] Modellierungs- und Simulationsumgebungen ermöglichen es dem Entwickler, eine Anwendung schrittweise zu entwerfen und ihr Verhalten bereits früh im Entwicklungsprozess kontinuierlich zu testen, zu analysieren und zu optimieren. Automatische Code-Generatoren transformieren Modelle typischerweise nach C-Code, der später compiliert und auf einer Zielplattform ausgeführt wird. Aber obwohl der generierte Code perfekt mit der modellierten Funktionalität übereinstimmen mag, ist es wahrscheinlich, dass sich die Anwendung, wenn sie auf einer Hardware-Plattform ausgeführt wird, leicht anders verhält, oder sogar völlig unerwartetes Verhalten zeigt.

[0028] Auf einer Hardware-Plattform kann die Ausführung einer Regelungs-Task-Funktionalität oder einer Netzwerk-Kommunikation beträchtliche Zeit benötigen, wohingegen die Ausführung in einer Simulationsumgebung grundsätzlich ohne Zeit in Anspruch zu nehmen fertig gestellt wird. Insbesondere für verteilte Systeme ist es üblich, dieses Auseinanderklaffen (= Mismatch) durch die Einführung zusätzlicher, zufälliger plattformspezifischer Verzögerungen in das intrinsisch plattformneutrale Modell aufzuweichen. Mit anderen Worten formuliert, sind die Modell-Simulation und die Ausführung des entsprechenden Codes nur im besten Fall ungefähr gleich und, da der generierte Code typischerweise manuell optimiert (= fine-tuned) wird, es geht die Beziehung zum Modell verloren.

[0029] Das Konzept der logischen Ausführungszeit (= Logical Execution Time = LET), das im Einführungsteil oben erwähnt wurde, abstrahiert von der jeweiligen Plattform und Kommunikationstopologie. Das erlaubt die Änderung der zugrundeliegenden Plattform und sogar die Verteilung der Komponenten auf unterschiedliche Knoten, ohne das gesamte Systemverhalten zu beeinflussen, wenn genügend Rechen- und Kommunikations-Ressourcen vorhanden sind. Das verbessert den Status von Simulationsumgebungen beträchtlich, da Simulationsergebnisse perfekt mit dem Verhalten auf einer Zielplattform übereinstimmen. Frühere Arbeiten haben gezeigt, dass die Simulation LET-basierter Anwendungen im Prinzip mit dem verbreiteten Simulationssystem MATLAB/Simulink machbar ist (siehe G. Stieglbauer: ”Model-based Development of Embedded Control Software with TDL and Simulink”, Doktorarbeit, Universität Salzburg, 2007).

[0030] Die weitere Diskussion fokussiert sich auf drei Aspekte, die in der Praxis unmittelbar auftauchen und die eine Verfeinerung der bisherigen Simulationsansätze benötigen: (1) die Simulation mehrerer LET-basierter Komponenten mit zyklischen Datenabhängigkeiten; (2) die Simulation von Regelungsschleifen mit Anlagen ohne Verzögerung; und (3) die schrittweise Migration von Simulationsmodellen in Richtung LET.

[0031] Bevor wir die Details der Simulation betrachten, führen wir nachfolgend in die oben erwähnte Timing Definition Language (”TDL”) ein. TDL ist eine weiterentwickelte Implementierung des LET-Konzepts der Giotto-Sprache.

[0032] Die Timing Definition Language (TDL) ist eine höhere (= highlevel) Software-Beschreibungssprache, die die explizite Spezifikation des Zeitverhaltens von harten Echtzeitkomponenten in einer plattformunabhängigen Art erlaubt. TDL basiert auf der logischen Ausführungszeit-(= Logical Execution Time = LET) Abstraktion, die von der Giotto-Sprache bekannt ist. LET bedeutet, dass das beobachtbare zeitliche Verhalten einer Task unabhängig von ihrer physischen Ausführung ist. Die einzige notwendige Annahme ist, dass die physische Task-Ausführung schnell genug ist, um irgendwo zwischen dem logischen Start- und Endepunkt zu passen. Für die jeweilige Plattform muss das durch adäquates Scheduling sicher gestellt werden, wobei für jede Task die ungünstigste Ausführungszeit (= Worst Case Execution Time) gegeben ist. Die Task-Eingaben werden zum Freigabe-(= Release-)Zeitpunkt gelesen und die neu berechneten Ausgaben sind zum Beendigungs-(= Terminate-)Zeitpunkt verfügbar. Zwischen diesen beiden logischen Zeitpunkten haben die Ausgaben den Wert der vorigen [Task-]Ausführung. Obwohl LET eine zusätzliche Verzögerung der Antwort (= response time overhead) einführt, schafft es [LET] die Basis für deterministisches Verhalten, Plattform-Abstraktion und sauber definierte Interaktionssemantik zwischen parallelen Aktivitäten. Um komplexe Anwendungen zu ermöglichen, führt TDL ein Komponentenmodell ein. Anwendungen können in einzelne Komponenten zerlegt werden, von denen jede durch ein TDL-Modul repräsentiert wird.

[0033] Nachfolgend werden das LET-Konzept und die grundlegenden technischen Begriffe ”Modul”, ”Modus” (= Mode), ”Task”, ”Port” und ”Guard” erklärt, um die weitere technische Diskussion zu erleichtern.

[0034] Ein Modul führt Berechnungen in Form von Tasks durch und kommuniziert mit der Umgebung mittels Sensoren und Aktoren. Module können einen oder mehrere andere Module importieren und auf ihre öffentlichen Entitäten zugreifen. Zur Laufzeit laufen alle Module einer Anwendung parallel. Deswegen ist eine TDL-Anwendung die parallele Komposition einer Menge von TDL-Modulen, die zu einer gemeinsamen Zeitbasis synchronisiert sind. Ein Modul kann einen oder mehrere Modi haben, ist aber zu einem Zeitpunkt in exakt einem Modus.

[0035] Ein Modus ist ein jeweiliger Betriebszustand eines Moduls. Modi haben eine Periode und bestehen aus einer Menge von Aktivitäten. Eine Aktivität kann ein Task-Aufruf, eine Aktor-Aktualisierung oder ein Modus-Wechsel sein. Die LET einer Task ist immer größer als Null, wohingegen Aktor-Aktualisierungen und Modus-Wechsel in logisch Null-Zeit (= Logigal Zero Time = LZT) ausgeführt werden, was in der Quintessenz bedeutet, dass sie ”schnell genug und daher vernachlässigbar” sind.

[0036] Eine Task repräsentiert eine Berechnungseinheit einer Anwendung. Eine Task deklariert eine externe Funktion, die in einer beliebigen imperativen Sprache wie zum Beispiel C implementiert werden kann. Eine Task deklariert außerdem Eingabe-, Zustands- und Ausgabestellen (= Ports).

[0037] Ports sind typisierte Variablen, die für Datenkommunikation benutzt werden. TDL unterscheidet zwischen Task-Ports, Sensoren und Aktoren. Ein Wächter (= Guard) ist eine externe Boole'sche Funktion. Guards können benutzt werden, um Modus-Aktivitäten bedingt auszuführen.

[0038] Abb. 1 zeigt ein System **1**, das aus drei Knoten (Bezeichnung **3**) mit der Bezeichnung ”node 1”, ”node 2” und ”node 3” bestht und die durch einen mit ”bus” bezeichneten Kommunikationskanal **5** verbunden sind. Der Bus wird für die Datenkommunikation zwischen den mit 3 bezeichneten Knoten verwendet. Die Knoten sind elektronische Geräte, die in manchen Anwendungsbereichen wie zum Beispiel der Automobilindustrie als ”Electronic Control Units” (= ECUs) bezeichnet werden. Jeder Knoten kann eine dedizierte Hardware haben, die den Knoten mit dem Kommunikationskanal verbindet und die dann allgemein als ”Controller” bezeichnet wird. In dem in Abb. 1 dargestellten Beispiel ist der Kommunikationskanal ein Bus mit ”Rundfunk-Semantik” (= Broadcast Semantics), was bedeutet, dass Daten, die von einem Knoten an den Kommunikationskanal übertragen werden, von jedem anderen Knoten empfangen werden können. Die vorliegende Erfindung ist jedoch nicht auf derartige Kommunikationskanäle beschränkt, sondern umfasst auch Kanäle mit anderer geeigneter Topologie und Semantik.

[0039] System **1** ist so konfiguriert, dass es Software bestehend aus den Modulen M1, M2, M3 und M4 ausführt. Module sind ein Beispiel für eine Methode zur Strukturierung komplexer Software, und ein Modul ist allgemein ein Stück Software mit einer Programmierschnittstelle (= Application Programming Interface = API). Software, die aus mehreren Modulen besteht, erlaubt die transparente Verteilung der Software über mehrere Knoten, um diese auszuführen. Im Beispiel, das in Abb. 1 illustriert wird, führt ”node 1” die Module M1 und M2 aus, ”node 2” den Modul M3 und ”node 3” den Modul M4.

[0040] Abb. 2 zeigt ein spezifischeres Beispiel für Software, die aus zwei Modulen besteht. Die Beispiel-Software in Abb. 2 besteht aus einem Modul **7** (bezeichnet als ”Module Service”) und einem Modul **8** (bezeichnet als ”Module Client”). Jeder Modul kann eine Menge von Sensoren **9**, eine Menge von Aktoren **10** und eine Menge von Modi **11** haben. Die Sensoren **9** von Modul **7** haben die Bezeichnung ”S1” und ”S2”. Der Sensor **9** von Modul **8** hat die Bezeichnung ”S”. Die Aktoren **10** von Modul **7** haben die Bezeichnung ”A1”, ”A2” und ”A3”, und die Aktoren **10** von Modul **8** haben die Bezeichnung ”A1” und ”A2”. Modul **7** hat zwei Modi **11**, bezeichnet als ”mode 1” und ”mode 2”. Modul **8** hat drei Modi **11**, bezeichnet als ”mode 1”, ”mode 2” und ”mode 3”.

[0041] Jeder Modul **7**, **8** kann nur in einem Modus zu einem Zeitpunkt sein. Modus **1** des Moduls **7** hat zwei Tasks, die mit ”task 1” und mit ”task 2” bezeichnet sind, wobei ”task 1” periodisch mit einer Periode von zehn Millisekunden ausgeführt wird (durch das ”[10 ms]” angedeutet) und eine Task 2 wird periodisch mit einer Periode von zwanzig Millisekunden ausgeführt (durch das ”[2.0 ms]” angedeutet).

[0042] Im vorliegenden Beispiel können die Task-Aufrufe LET-Semantik haben, wie sie beim Giotto-Programmiermodell eingeführt wurde (siehe dazu den Beitrag von T. A. Henzinger et al., wie er im Einführungsteil erwähnt wurde). Abb. 3 zeigt schematisch die Task-Ausführung nach LET-Semantik. Die Eingaben der Task werden zu Beginn der LET-Periode gelesen. Das Lesen der Eingaben erfolgt praktisch in nahezu keiner Zeit (= zero time), was als ”logisch Null Zeit” (= ”logical zero time” = LZT) bezeichnet wird. Der Beginn der LET-Periode ist mit der Bezeichnung ”release” in Abb. 3 gekennzeichnet. Neu berechnete Ausgaben der Task sind exakt am Ende der LET-Periode verfügbar, was mit der Bezeichnung ”terminate” in Abb. 3 gekennzeichnet ist. Die tatsächliche (= physical) Ausführung der Task auf einem Knoten beginnt zum Zeitpunkt des Pfeils mit der Bezeichnung ”start” in Abb. 3 und endet zum Zeitpunkt des Pfeils mit der Bezeichnung ”stop”, wobei die tatsächliche Ausführung der Task zum Zeitpunkt des Pfeils mit der Bezeichnung ”suspend” unterbrochen wird, und zum Zeitpunkt des Pfeils mit der Bezeichnung ”resume” wieder aufgenommen wird.

[0043] Die Zeit, wann die tatsächliche Ausführung innerhalb der LET-Periode erfolgt, wird durch die LET nicht festgelegt. Es ist aber eine Anforderung, dass die tatsächliche Ausführung der Task vor dem Ende der LET-Periode endet. Mit anderen Worten formuliert, kann der Beginn der tatsächlichen Ausführung der Task zu jedem Zeitpunkt am Beginn der LET oder danach erfolgen, und das Ende der tatsächlichen Ausführung der Task muss spätestens, auch im ungünstigsten Fall, vor oder zum Ende der LET-Periode erfolgen. Gemäß der LET-Semantik stehen nach aussen die Ergebnisse der Berechnung der Task erst am Ende der LET-Periode oder danach zur Verfügung, statt am Ende der tatsächlichen Ausführung der Task. Das bedeutet, dass vor dem Ende der LET-Periode nur die ”alten” Werte der vorhergehenden Ausführung der Task verfügbar sind.

[0044] Bezugnehmend auf Abb. 2 bedeutet das, dass Task 1 in Modus **1** von Modul **7** mit einer Periode von 10 Millisekunden wiederholt ausgeführt wird, wobei der Sensor exakt zu Beginn der 10 Millisekunden-Periode in LZT gelesen wird und die Ergebnisse der Berechnung von Task 1 für den Aktor A1 exakt am Ende der 10 Millisekunden-Periode bereit gestellt werden.

[0045] Abb. 2 illustriert auch die Kommunikation zwischen Tasks über Modul-Grenzen hinweg. Eine Ausgabe von Task 2 von Modus **1** in Modul **7** wird als ”task2.o” bezeichnet und wird als Eingabe von Task 1 von Modus **1** in Modul **8** bereit gestellt.

[0046] Die Komposition von Software aus einer Menge von Modulen und die Definition von Tasks dieser Module gemäß LET-Semantik erlaubt eine transparente Verteilung der Software auf einen oder mehrere Knoten, wobei das zeitliche Verhalten der Software garantiert wird. Insbesondere wird das Hinzufügen eines Moduls das beobachtbare zeitliche Verhalten der anderen Module nie verändern, so lange die internen Scheduling-Verfahren der jeweiligen Knoten die Übereinstimmung mit der LET-Semantik garantieren, unter der Annahme, dass die ungünstigsten Ausführungszeiten (= Worst Case Execution Times = WCETs) für alle Tasks bekannt sind.

[0047] Abb. 4 zeigt die Ausführung von Modul M1 und M2 durch node 1 aus Abb. 1. Modul M1 hat eine Task ”task 1” mit LET1, und Modul M2 hat eine Task ”task 2” mit LET2. Task 2 erhält die Ausgabe von Task 1 als Eingabe, und die LET1 von Task 1 ist zweimal so lange wie die LET2 der Task 2. Die Rechtecke in Abb. 4 zeigen schematisch die tatsächlichen Ausführungszeiten von Task 1 und von Task 2. Die Ausgaben von Task 1 werden an Task 2 am Ende der logischen Ausführungszeit LET1 der Task 1 (siehe Pfeil) bereit gestellt. Das kann dadurch erreicht werden, dass die Werte aus einem Speicherbereich, der Task 1 zugeordnet ist, in einen Speicherbereich kopiert werden, der Task 2 zugeordnet ist. Dieses Kopieren nimmt auf einem einzelnen Knoten nahezu keine Zeit in Anspruch (= logisch Null Zeit = logical zero time = LZT).

[0048] Sowohl der 3. als auch der 4. Aufruf von Task 2 in Abb. 4 werden die Ausgabe des ersten Aufrufs von Task 1 erhalten. Das bedeutet, dass die 4. Ausführung von Task 2 nicht die Ausgabe des zweiten Aufrufs von Task 1 nutzen wird, obwohl die tatsächliche Ausführung des zweiten Aufrufs von Task 1 bereits fertig ist sobald die tatsächliche Ausführung des vierten Aufrufs von Task 2 beginnt.

[0049] Giotto- und TDL-Programme werden beide von einer virtuellen Maschine (VM) ausgeführt, die dafür verantwortlich ist, ein korrektes Zeitverhalten sicherzustellen. Für TDL-Programme wird diese VM als E-Machine (Embedded Machine = eingebettete Maschine) bezeichnet. Die E-Machine ist eine VM, die den Grundstein für plattformunabhängige Echtzeit-Software legt. Ihre eigene Implementierung hängt von der Plattform (z. B. Betriebssystem) ab. Die E-Machine stellt das korrekte Zeitverhalten der Echtzeitanwendungen sicher. Hierfür wird das in einer TDL-Komponente beschriebene Zeitverhalten in einen Zwischen-Code namens E-Code (Embedded Code = eingebetteter Code) kompiliert. Der E-Code beschreibt die Reaktivität der Anwendung, d. h. die Zeitpunkte wo Tasks freigegeben (= release) bzw. terminiert (= terminate) werden oder mit der Umgebung interagiert wird.

[0050] Es handelt sich dabei um eine portable Beschreibung, da sich der E-Code auf logische Zeitpunkte stützt und plattformunabhängig ist. E-Code ist als Sequenz von Instruktionen repräsentiert, der von einer E-Machine interpretiert wird (vgl. T. A. Henzinger, C. M. Kirsch, ”The embedded machine: predictable, portable real-time code”, Proceedings of the ACM SIGPLAN 2002 Conference an Programming language design and implementation, pp. 315–326).

[0051] Die E-Code Instruktionen für TDL basieren auf denen von Giotto. Eine TDL E-Code Instruktion C(args) besteht aus dem Kommando C und seiner Liste aus Integer-Argumenten args. Alle Kommandos werden synchron ausgeführt: call(d) führt den Treiber d aus; release(t) markiert einen Task t als ”Bereit zur Ausführung”; future(a, dt) plant die Ausführung des E-Code Blocks startend bei Adresse a in dt Mikrosekunden; if (g, aelse) fährt mit der nächsten Instruktion fort falls g true (= wahr) zurückliefert, und springt andernfalls zur Adresse aelse; jump(a) springt zum E-Code an der Adresse a; switch(m) führt einen Modus-Wechsel (= mode switch) zum Modus (= Mode) m durch, d. h. die E-Machine fährt mit der ersten Instruktion von m fort; return beendet einen E-Code Block; nop(f) ist eine Pseudo-Instruktion, wobei f als Markierung zur Identifizierung von unterschiedlichen E-Code Bereichen benutzt wird.

[0052] Entsprechend der E-Code Instruktionen übergibt die E-Machine zeitgerecht Tasks an einen Verteiler (= Dispatcher) und führt Treiber aus. Ein Treiber führt Aktivitäten zur Kommunikation durch, wie das Lesen von Sensor-Werten, Bereitstellen von Eingangswerten für Tasks zu deren release-Zeitpunkten, oder das Kopieren von Ausgangswerten zu deren terminate-Zeitpunkten. Zusatzsoftware (= Plug-ins) des TDL-Compilers generiert automatisch Treiber für eine bestimmte Plattform gemäß den Sprachanbindungsregeln. Die Treiber sind Teil des so genannten Zwischen-Codes (Glue-Code), der auch noch Typdeklarationen, etc., und die Kommunikationsschicht beinhaltet. Der Funktionalitätscode, d. h. Task-, Guard-, Initialisierungs-, Sensor-, und Aktor-Funktionen, muss separat bereitgestellt werden.

[0053] Ein Hauptvorteil der LET Abstraktion ist das identische Verhalten von Echtzeit-Anwendungen auf allen Plattformen die genug Leistung bieten. Das macht Simulation von LET-basierten Anwendungen besonders nützlich. Speziell bei verteilten Anwendungen, versagen die meisten Simulationsumgebungen dabei, nicht-funktionale Eigenschaften, wie z. B. das Zeitverhalten, zu beachten.

[0054] Normalerweise können Simulationsergebnisse nur als ungefähre Annäherung des tatsächlichen Verhaltens auf einer echten Hardwareplattform gesehen werden. Bei Betrachtung einer Simulationsumgebung als eine weitere Plattform, garantiert die LET Annahme gleiches Verhalten. Deshalb stimmen das Verhalten bei der Simulation und das Verhalten in der Praxis exakt überein.

[0055] Nachdem Kommunikationslatenzen auch unter der logischen Ausführungszeit (LET) subsumiert werden können, ist die Simulation vollkommen unabhängig von der beabsichtigten Kommunikationstopologie.

[0056] Um für einen Task innerhalb einer Simulationsumgebung LET-Semantik zu erzielen, muss die Simulationsmaschine sicher stellen, dass (1) der Task zu wohldefinierten logischen Zeitpunkten freigegeben und terminiert wird, (2) Eingänge nur genau einmal gelesen werden und zwar wenn der Task freigegeben wird, und (3) Ausgangsports ausschließlich mit einem neuen Wert beschrieben werden, wenn der Task logisch terminiert. **Abb. 5** zeigt ein Datenflussdiagramm, das einer solchen Semantik folgt. Ein Task-Funktionsblock **30**, der die Task-Funktionalität implementiert, besitzt eine Abtastzeit T die übereinstimmt mit der Taskperiode π_{task}. Der Block ist zusätzlich umgeben von einem Zero-Order-Hold Block **31** und einem Diskrete-Delay-Block **32**. Die Abtastzeit des Zero-Order-Hold Blocks ist auf die Taskperiode π_{task} gesetzt, womit sichergestellt wird, dass Eingangswerte während der Taskausführung nicht verändert werden. Die Abtastzeit des Delay-Blocks ist auf die logische Ausführungszeit der Task LET_{task} gesetzt, womit sichergestellt wird, dass die neu berechneten Ausgangswerte erst verfügbar sind, nachdem der Task logisch terminiert ist.

[0057] Dieser Ansatz. ist geradlinig, versagt aber, wenn Moduswechsel-Logik und unterschiedliche Ausführungsraten beachtet werden müssen. Typischerweise haben Kontrollsysteme periodisch ausgeführte Tasks und involvieren Moduswechsel-Logik. Je nach aktuellem Modus (= Mode) eines Moduls, führt die Applikation einzelne Tasks mit unterschiedlichen Zeitbedingungen aus oder ersetzt gar auszuführende Tasks durch andere. Mit Ausnahme von sehr einfachen Anwendungen, ist es schwierig, das Gesamtverhalten des Modells zu verstehen, und seine Ausführung wird zu ineffizient. Des Weiteren ist es ungelöst, wie die exakte TDL-Semantik eines Multi-Mode Multi-Rate Systems im allgemeinen Fall erhalten werden kann.

[0058] Grundsätzlich involviert das Simulieren eines wie in **Abb. 5** dargestellten, als Blockdiagramm beschriebenen Systems das wiederholte (1) Lösen aller Block-Ausgangsgleichungen, (2) Aktualisieren der Zustandsvariablen im System, und (3) das Fortschreiten in der Zeit. Prinzipiell gibt es mehrere Strategien für Punkt (1). Die Blöcke könnten so lange in beliebiger Reihenfolge ausgeführt werden, bis sich ihre Ausgänge nicht mehr verändern, d. h. ein Fixpunkt (Dauerzustand) erreicht ist. Dieser Ansatz ist folglich eher ineffizient.

[0059] Deshalb verfolgen moderne Simulationssysteme wie z. B. die MATLAB/Simulink Werkzeuge eine andere Strategie. Bei der Initialisierung wird eine sortierte Ordnung der Blöcke abgeleitet, wodurch ein Iterieren unnötig ist. Zu jedem Simulationsschritt werden die Blöcke in dieser festen Ordnung ausgeführt, die nicht notwendigerweise mit den Verbindungen der Blöcke übereinstimmt, sondern auch von der Datendurchlass (= Feedthrough)-Charakteristik der einzelnen Blöcke abhängt. Ein Block mit direktem Feedthrough muss seine aktuellen Eingangswerte kennen, bevor er seine Ausgangswerte setzen kann. Bei Modellen mit ausschließlich diskreten Blöcken berechnet die Simulationsmaschine (z. B. Simulink) zu jedem Simulationsschritt die Ausgangswerte des Modells, indem die Ausgabe-Methode (z. B. „mdlOutput" für Simulink) eines jeden Blocks in der während der Initialisierung ermittelten Reihenfolge aufgerufen wird. Danach berechnet die Simulationsmaschine den Zustand des Modells, indem die Aktualisierungsmethode (z. B. „mdlUpdate" für Simulink) von jedem Block wiederum in der vordefinierten Reihenfolge ausgeführt wird.

[0060] Im Normalfall kontrolliert die Simulationsumgebung die Ausführungsreihenfolge der Blöcke. Dies trifft nicht für so genannte „bedingt-ausgeführte Subsysteme" zu, wie z. B. getriggerte Subsysteme. Ein Subsystem ist ein Block, der andere Blöcke enthält, und somit ein Mittel zur hierarchischen Strukturierung von Blöcken darstellt. Ein Beispiel für einen Trigger ist ein Funktionsaufruf, der beispielsweise von einem S-Function Block stammt (siehe unten). Bedingt-ausgeführte Subsysteme sind nicht Teil der sortierten

[0061] Ausführungsreihenfolge, sondern werden im Kontext ihres initiierenden Blocks ausgeführt und „erben" sozusagen ihre Position in der sortierten Blockreihenfolge. Diese Position wird durch oben genannte Feedthrough-Charakteristiken und Datenabhängigkeiten des Blocks selbst und all der von ihm getriggerten Blöcke bestimmt. Ein initiierender Block entscheidet autonom, welche Blöcke er ausführt und in welcher Reihenfolge.

[0062] Für die folgende Diskussion wurde die MATLAB/Simulink Simulationsumgebung als Beispiel ssimulationsumgebung ausgewählt. Trotzdem ist die vorgestellte Erfindung nicht auf MATLAB/Simulink beschränkt, sondern auch anwendbar bei anderen Simulationswerkzeugen, die eine statische Ausführungsordnung der Blöcke verwenden, die auf einer

topologischen Sortierung entsprechend des involvierten Datenflusses basiert.

[0063] Das eingebaute Simulink-Blockset kann mittels so genannter „S-Functions" erweitert werden. S-Functions sind Simulink-Blöcke, die in einer Programmiersprache wie z. B. C implementiert sind. Sie können die Ausführung von Function-Call-Subsystemen triggern, und zwar mittels Simulink Makro "ssCallSystemWithTid" in ihrer "mdlOutputs" Methode. Nachdem das Function-Call-Subsystem ausgeführt wurde, wird die Programmkontrolle wieder der S-Function übergeben, die mit ihrer Ausführung fortfährt.

[0064] Die oben beschriebene E-Machine ist in Simulink mittels S-Function realisiert, die den E-Code Interpreter enthält. Task-, Guard- und Initialisierungsfunktionen sind als Function-Call-Subsysteme realisiert und müssen vom Applikationsentwickler mit standardmäßigen Simulink-Blöcken implementiert werden. Der für die Kommunikationsaktivitäten, wie z. B. Lesen von Sensoren oder Kopieren von Taskausgangswerten auf Aktoren, benötigte Zwischen-Code ist ebenfalls mit Function-Call-Subsystemen realisiert. Sie alle werden zu den passenden Zeitpunkten von der E-Machine aufgerufen (d. h. getriggert).

[0065] Die Ausgangsports solcher Treibersubsysteme sind direkt mit den Eingangsports verbunden. Das entspricht Zuweisungen im imperativen Programmierparadigma, sobald das System getriggert wird. Die TDL-Werkzeugkette generiert die Treibersubsysteme automatisch, sobald die Simulation startet. Nur das Zeitverhalten, d. h. die TDL-Beschreibung, muss spezifiziert und die Funktionalität, z. B. Task-Funktionssubsysteme, modelliert werden.

[0066] Abb. 6 zeigt diesen E-Machine Ansatz exemplarisch für ein vereinfachtes Beispiel. Die Platzierung der einzelnen Blöcke entspricht dem Datenfluss, der von links nach rechts entlang den Pfeilen von einer Quelle (Ausgang eines gegebenen Blocks) zu einer Senke (Eingang eines darauf folgenden Blocks) verläuft. Der Quellwert wird von einem Sensor gelesen, der den Wert einem Guard und einem Task zur Verfügung stellt. Der Aktorblock verwendet den Ausgangsport eines Tasks, um eine Senke zu beschreiben. In Hinblick auf die Ausführungsreihenfolge sollte die Abbildung nicht von links nach rechts gelesen werden. Die E-Machine triggert die einzelnen Blöcke dem E-Code entsprechend, was zu oben genannter Ausführungsreihenfolge der Aktivitäten führt. Dies stellt auch das korrekte LET-Verhalten eines Tasks sicher, indem seine Freigabe- und Terminierungstreiber zu den richtigen Zeitpunkten getriggert werden.

[0067] Sobald die E-Machine die Ausführung eines Guards getriggert hat, liest sie sofort das Ergebnis über einen Eingangsport. Das Ergebnis beeinflusst die weitere Ausführung der E-Code Instruktionen und in weiterer Folge auch welches Subsystem getriggert werden muss. Deshalb hat die E-Machine direktes Feedthrough. Die E-Machine muss immer dann aufgerufen werden, wenn die Simulationszeit gleich der logischen Zeit einer TDL-Aktivität gemäß dem E-Code ist. Um dies sicherzustellen, verwenden wir für die E-Machine eine konstante Abtastrate, die dem GGT (größten gemeinsamen Teiler) aller Aktivitätsperioden entspricht.

[0068] Die S-Function Implementierung der E-Machine für eine, wie in Abb. 6 dargestellte, Simulationsumgebung ergibt ein geradliniges und effizientes Simulationsmodell. Wie auch immer, aufgrund von Problemen mit Datenabhängigkeiten, wie sie in Simulationsumgebungen auftreten können, hat sich seine praktische Anwendbarkeit als eingeschränkt herausgestellt; insbesondere, wenn Regelungen mit einer oder mehreren Rückkoppelungen (= Feedback loops) simuliert werden.

[0069] Die folgenden Anwendungsszenarien können im Normalfall nicht mit dem einfachen oben beschriebenen Ansatz hinsichtlich Abb. 6 behandelt werden: (1) Zyklische Importbeziehungen zwischen LET basierten Komponenten (Kontrollern), (2) Kontrollschleifen mit Regelstrecken ohne Zeitverzögerung, (3) Systeme mit sowohl LET basierten als auch konventionell modellierten Reglern. Diese drei Fälle sind weiter unten detailliert beschrieben. Sie alle haben mit zyklischem Datenfluss zu tun und der Fähigkeit einer Simulationsumgebung eine gültige Ausführungsstrategie für alle einzelnen Blöcke zu finden. Die meisten der für gewöhnlich genutzten Simulationsumgebungen (MATLAB/Simulink, LabView etc.) unterstützen Modelle mit Schleifen ohne Zeitverzögerung (algebraic loops) nicht oder nicht vollständig.

[0070] Zeitverzögerungen (= delays) werden durch explizite Delay-Blöcke oder durch andere Blöcke mit indirektem Feedthrough eingeführt. Indirektes (= indirect oder non-direct) Feedthrough bedeutet, dass der Output y eines Blocks am Zustand x, aber nicht direkt vom Eingang u des Blocks abhängt. Der Delay-Block z. B., der eine Zeitverzögerung Δ bietet, hat folgende Zustandsraumbeschreibung

$$y(t) = x(t),\ x(t + \Delta) = u(t),$$

wobei Δ das Delay ist (z. B. die Abtastrate des Blocks).

[0071] Im Folgenden wird wiederum die Simulationsumgebung MATLAB/Simulink zur illustrativen Erklärung als Beispiel für eine Simulationsumgebung verwendet.

[0072] Importbeziehungen führen zu Simulink-Signalen zwischen Treiberblöcken von unterschiedli-

chen Modulen. In Abb. 2 ist der Termination-Treiber des Task 2 in Modul **7** „Service" mit dem Release-Treiber von Task 1 in Modul **8** „Client" verbunden. Wie bereits oben erwähnt, benutzt Simulink eine konstante Reihenfolge, in der Blöcke ausgeführt werden. Im Falle einer Feedback-Schleife zwischen Modul **8** „Client" und Modul **7** „Service" schließt solch eine vordefinierte Ausführungsreihenfolge einen Ansatz aus, der eine E-Machine pro Modul (vgl. Abb. 6) benutzt, nachdem relativ zu anderen Blöcken alle Treiber eines Moduls auf einmal ausgeführt werden. Nur eine globale E-Machine, die den E-Code aller Module interpretiert, führt zu einem Modell, das der LET Semantik folgt und von einer Simulationsmaschine aufgelöst werden kann.

[0073] Feedbackschleifen sind ein altbekanntes Konzept in der Kontrolltheorie (z. B. PID oder sonstige Controller). Mittels Sensoren überwacht ein Controller einen Ausgang der Regelstrecke (= Plant), d. h. das zu regelnden System, und passt die Aktoren um die spezifizierte Reaktion zu erhalten. Plants mit direktem Feedthrough, d. h. ohne ein Delay einzuführen, müssen als Ganzes ausgeführt werden, nachdem die Aktoren aktualisiert, aber bevor die Sensoren gelesen werden. Mit dem E-Machine Ansatz in Abb. 6 sind sowohl Sensoren als auch Aktoren unter der Kontrolle eines einzelnen S-Function Blocks. Deshalb kann Simulink keine gültige Ausführungsreihenfolge der Blöcke finden.

[0074] Ein anderes Szenario betrifft die Interaktion zwischen TDL-Modulen (Controllern) und Controllern, die nicht LET-basiert sind, z. B. wenn eine Kontrollapplikation schrittweise in Richtung LET migriert wird. Typischerweise werden Controller als atomare (= atomic, non-virtual) Subsysteme in Simulink modelliert, um die gewünschte Softwarepartitionierung in der Simulation und der Programmsynthese widerzuspiegeln. Die durch ein atomares Subsystem definierten Gleichungen werden als Einheit evaluiert. Für gewöhnlich werden Unit-Delay-Blöcke zwischen Controllern und der Regelstrecke, aber auch zwischen mehreren Controllern hinzugefügt, um ‚falsche' algebraische Schleifen (= algebraic loops) zu vermeiden und die Rechenzeit auf einer echten Hardwareplattform annäherungsweise zu imitieren.

[0075] **Abb.** 7a illustriert ein Beispielmodell mit zwei konventionell modellierten Controllern (mit "c" und "c2" bezeichnet) und eine Plant. **Abb.** 7b illustriert das gleiche Modell, welches nun teilweise auf LET basiert. Der Controller c ist durch ein TDL-Modul ersetzt, welches die ehemalige Implementierung als Task ausführt. Das Unit-Delay ist nun implizit durch die LET der Task ersetzt, welche von der E-Machine sichergestellt wird. Wiederum kann die Simulationsumgebung MATLAB/Simulink keine gültige Ausführungsreihenfolge finden.

[0076] In jedem dieser Fälle meldet Simulink eine Datenabhängigkeitsverletzung (= data dependency violation), die ähnlich einem Algebraic-Loop-Fehler (= algebraic loop error) ist. Aus der Sicht eines Regelungstechnikers erscheint dies uneinsichtig, nachdem die LET eines Tasks immer größer als Null ist und somit die benötigte Zeitverzögerung einführen müsste. Allerdings „sieht" die Simulationsumgebung durch die Struktur der E-Machine Implementierung, wie in Abb. 6 dargestellt, die Zeitverzögerung nicht.

[0077] Im Folgenden wird ein neuer Mechanismus zum Simulieren LET-basierter Softwarekomponenten beschrieben. Die Simulation basiert auch auf einer E-Machine (vgl. Abb. 6), leidet aber nicht an oben beschriebenen Datenabhängigkeitsverletzungen. Zuerst werden neue Anforderungen betreffend der E-Code Darstellung und danach die Ausführungsstrategie während der Simulation diskutiert. Der beschriebene Mechanismus wurde in MATLAB/Simulink implementiert. Allerdings ist das zugrunde liegende Konzept auch für andere Simulationsumgebungen anwendbar. Schlussendlich wird eine Erweiterung zum Abdecken ereignisgesteuerter Systeme diskutiert.

[0078] Datenabhängigkeiten zwischen TDL-Modulen setzt eine Partitionierung des E-Codes in zwei Abschnitte voraus. Um allerdings die Plant oder andere Nicht-TDL-Blöcke zwischen dem Aktualisieren von Aktoren dem Lesen von Sensoren auszuführen, muss der E-Code in drei disjunkte Abschnitte geteilt werden. Diese E-Code Abschnitte repräsentieren die folgenden drei Phasen:

tt Task-Terminierungsphase (= task termination phase): Die tt-Phase beinhaltet alle Aktivitäten die ausgeführt werden müssen, um Ausgangsports am Ende der LET einer Taskausführung bereitzustellen. Danach sind aktualisierte Ausgangsports sichtbar für Client-Module und als Eingang für andere Aktivitäten verwendbar.

au Aktor-Aktualisierungsphase (= actuator update phase): Die au-Phase beinhaltet alle Aktivitäten die ausgeführt werden müssen, um Aktoren mit neuen Werten zu aktualisieren und potentielle Guards auszuwerten.

mstr Modewechsel- und Taskfreigabephase (= mode switch and task release phase): Die letzte Phase mstr beinhaltet alle Aktivitäten die ausgeführt werden müssen, um Modewechsel durchzuführen und neue Taskaufrufe freizugeben. Einen Taskaufruf freizugeben bedeutet die Aktualparameter, die von Sensoren oder Taskausgängen stammen, zu den Eingangsports des freigegebenen Tasks zu kopieren. Diese Phase umfasst auch die Ausführung von Task-Funktionen.

[0079] Jede einzelne Phase muss für alle TDL-Module ausgeführt werden. Diese E-Code Dreiteilung ist die Grundvoraussetzung um Modelle mit oben

genannten Datenabhängigkeiten zu simulieren. Von diesen Abhängigkeiten kann man unmittelbar die Ausführungsreihenfolge der drei Phasen und der übrigen Blöcke ableiten: (1) tt, (2) au, (3) Nicht-TDL-Blöcke, (4) mstr. Um die einzelnen Phasen zur Laufzeit zu unterscheiden, werden Markierungen (= Tags) im E-Code eingeführt, welche die zugehörigen Abschnitte von einander trennen. Markierungen werden als NOP (= no operation) Instruktionen mit einer entsprechenden Kennzeichnung (= Flag) dargestellt: „eot" markiert das Ende des tt Abschnitts (Phase), „eoa" markiert das Ende des au Abschnitts (Phase).

[0080] Man sollte beachten, dass eine triviale Lösung der Datenabhängigkeitsverletzung darin besteht, auch alle Nicht-TDL-Blöcke unter die Kontrolle der E-Machine zu stellen. Dies würde in einer vierten E-Machine Phase resultieren, welche zwischen der au- und der mstr-Phase mittels E-Machine Trigger ausgeführt werden würde. Dies widerspricht allerdings dem Verständnis einer unabhängigen Plant und bringt einige Nachteile mit sich; allen voran die Einschränkung, keine kontinuierlichen Simulink-Blöcke zu unterstützen.

[0081] Ein Ziel ist es, das Triggern von Akoren und Sensoren zu entkoppeln, um Nicht-TDL-Blöcke dazwischen auszuführen. **Abb. 8** illustriert die Ausführung der einzelnen Trigger entlang einer Zeitachse für ein einfaches Beispiel, wo ein Task von einem Sensor liest und einen Aktor aktualisiert: Zum Zeitpunkt t_0-LET wird ein Sensorausgangswert gelesen (sr: „sensor read"), der Task freigegeben (tr: „task release") und ausgeführt (tx: „task execute"); zum Zeitpunkt t_0 wird der Task terminiert (tt: „task terminate") und der Aktor aktualisiert (au: „actuator update). Hierauf, aber zum gleichen logischen Zeitpunkt, beginnt der nächste Aufruf mit der Ausführung von sr, tr und tx. Die Idee dieses Ansatzes ist es, die Ausführung aller Trigger für einen bestimmten Zeitpunkt in zwei getrennte Schritte (= Steps) aufzuteilen. Step 1 führt die sensorunabhängigen Aktivitäten wie die Terminierung von Tasks oder das Aktualisieren von Aktoren aus, während hingegen Step 2 Aktivitäten ausführt, welche potentiell von Sensoren abhängen, wie z. B. das Freigeben von Tasks. Jeder Step wird von einer anderen S-Function ausgeführt, so dass die Plant oder andere Nicht-TDL-Blöcke dazwischen ausgeführt werden können.

[0082] E-Machine 1 führt Step 1 aus, welcher die folgenden Aktivitäten beinhaltet: (1) Task-Terminierungstreiber, (2) Aktortreiber, und (3) deren Guards falls beide nicht von einem Sensorwert abhängen; zusätzlich führt E-Machine 1 (4) Initialisierungsfunktionen von Ports aus, sobald die Simulation startet.

[0083] E-Machine 2 führt Step 2 aus, welcher die folgenden Aktivitäten beinhaltet: (1) Sensortreiber, (2) Modeswitch-Treiber, (3) Modeswitch-Guards, (4) Task-Freigabetreiber, (5) Task-Ausführungstreiber, und (6) Task-Guards; zusätzlich (7) Aktortreiber und (8) deren Guards, wenn der Aktor selbst oder der Guard von einem Sensorwert abhängt.

[0084] Diesem Schema entsprechend, verteilt der Glue-Code-Generator die Liste der Treibera (Function-Call-Subsysteme) zwischen den beiden E-Machinen auf. Beide E-Machines operieren auf demselben E-Code und beide basieren auf der selben Implementierung. Sie unterscheiden sich lediglich in ihren mdlOutputs-Funktionen.

[0085] Die Aufteilung der E-Code Interpretation eines Moduls in separate E-Machines stellt zusätzliche Synchronisationsanforderungen. Von E-Machine 2 durchgeführte Modeswitches müssen an E-Machine 1 signalisiert werden. Präziser gesagt, liest E-Machine 1 das a und dt Argument der letzten Future-Instruktion von E-Machine 2 mittels eines Simulink Signals um die zeitgerechte Ausführung der richtigen E-Code Instruktion wieder aufzunehmen.

[0086] Nach der Ausführung von Step 2 vergeht Zeit, da die LET eines Tasks immer größer als Null ist. Allerdings erkennt Simulink dies nicht, nachdem es aus dem Datenfluss nicht hervorgeht. Simulink könnte demnach ohne weitere Maßnahmen keine gültige Ausführungsreihenfolge finden und würde weiterhin eine Datenabhängigkeitsverletzung melden. Immer wenn Datenfluss implizit durch die E-Machine (durch die Future-Instruktionen) verzögert wird, muss sich die Zeitverzögerung im Simulationsmodell widerspiegeln. Es vergeht Zeit sowohl zwischen der Ausführung und der Terminierung eines Tasks, als auch zwischen den E-Machine Aufrufen von Step 2 und Step 1. Das Platzieren eines Unit-Delay-Blocks zwischen jedem Paar von tx und tt Treibern und zwischen den beiden E-Machine Blöcken erlaubt Simulink eine gültige Ausführungsreihenfolge zu finden, die exakt der LET-Semantik folgt. Die Abtastrate der Delay-Blöcke wird auf die Periode der E-Machine gesetzt. Effektiv haben die Delay-Blöcke keinen Einfluss auf das beobachtbare Zeitverhalten.

[0087] **Abb. 9** zeigt die 2-Steg E-Machine Architektur und die Ausführungsreihenfolge des gesamten Simulationssystems anhand eines Beispiels dieser Erfindung. Die eingeführten Delay-Blöcke zwischen den Ausführungs- und Terminierungstreibern der Task und zwischen E-Machine 2 und E-Machine 1 werden als erstes ausgeführt. Danach führt die Simulationsumgebung E-Machine 1 aus. Die Plant oder irgendein anderer Nicht-TDL-Block wird als dritter ausgeführt. Danach führt E-Machine 2 den Step 2 aus.

[0088] Um durch zyklische Importverhältnisse ausgelöste unzulässige Datenabhängigkeiten zu vermeiden, müssen alle TDL-Module in einem Simulations-

modell von einem einzigen E-Machine Paar kontrolliert werden.

[0089] Abb. 10 illustriert das System aus Abb. 9 in einem abstrakteren Kontext. Es wird ein System zum Simulieren eines Echtzeitsystems gezeigt, wobei eine blockorientierte Simulation benutzt wird, die eine statische Ausführungsreihenfoge benutzt (d. h. topologisch sortierte Blöcke werden für die Simulation verwendet). Um einen bestimmten Task eines Modes eines Moduls eines Echtzeitsystems zu simulieren, besteht das Beispielsystem aus Abb. 10 aus einem ersten Task-Funktionsblock (task function) **9**. Der Task-Funktionsblock **9** beinhaltet, wie oben mit Referenz auf Abb. 8 und Abb. 9 erklärt, mindestens (1) einen Task-Freigabeblock (task release) **92**, (2) einen Task-Ausführungsblock (task execution, tx) **93**, (3) einen Task-Terminierungsblock (task termination, tt) **94**, und (4) einen Delay-Block **96**, welcher z. B. ein Unit Delay ist, angedeutet durch die Transferfunktion z^{-1} in der z-Domäne (= z-domain). Das System in Abb. 10 enthält weiters einen ersten Trigger-Block **10a** (E-Machine 2) und einen zweiten Trigger-Block **10b** (E-Machine 1) zum triggern der oben erwähnten Blöcke **92**, **93**, **94** in Übereinstimmung mit einem Trigger-Schema gegeben durch den Zeitverhalten beschreibenden E-Code.

[0090] Demnach ist der erste Trigger-Block **10a** so konfiguriert, dass er zumindest den Task-Freigabeblock (task release, tr) **92** zu einem ersten Zeitpunkt t_0 triggert. Der zweite Trigger-Block **10b** ist so konfiguriert, dass er zumindest den Task-Terminierungsblock (task termination, tt) **94** zu einem dritten Zeitpunkt $t_1 = t_0 + LET$ triggert, wobei LET die logische Ausführungszeit des Task-Funktionsblocks **9** ist.

[0091] Der Task-Ausführungsblock (task execution block, tx) **93**, der die gewünschte Funktionalität des Task-Funktionsblocks bereitstellt, wird – abhängig von der Position des Delay-Blocks **96** – entweder vom ersten Trigger-Block **10a** oder vom zweiten Trigger-Block **10b** zu einem zweiten Zeitpunkt (t_{0X}) getriggert. Im ersten Fall ist der Delay-Block **96** dem Task-Ausführungsblock (task execution block, tx) **93** nachgeschalten, in letzterem Fall ist der Delay-Block **96** dem Task-Ausführungsblock **93** vorgeschalten. Aufgrund des Delay-Blocks **96** werden algebraische Schleifen verhindert und es kann immer eine statische Ausführungsreihenfolge gefunden werden bevor die Simulation tatsächlich startet. Für die Simulation von Echtzeitsystemen erlaubt es die Verwendung des Systems in Abb. 10 einen Task zu simulieren, der den LET Anforderungen genügt, ohne auf das Problem zu stoßen, eine statische Ausführungsreihenfolge zu finden.

[0092] Die Uhr **8** bietet die Zeitbasis für die Simulation. Nachdem die Simulation gestartet wurde, „sieht" (zu regelmäßigen Zeitintervallen gegeben von der Zeitbasis, wobei sowohl fixe als auch variable Abtastraten unterstützt werden können) die E-Machine (Trigger-Blöcke **10a** und **10b**) wie viel Zeit seit dem Begin der Simulation vergangen ist und triggert in der Folge die unterschiedlichen Blöcke des Task-Funktionsblocks (der dem LET Konzept folgt) zu den korrekten, vom E-Code definierten Zeitpunkten.

[0093] Der Task-Ausführungsblock **93** ist konfiguriert, basierend auf einen Eingangswert in Übereinstimmung mit der gewünschten Transferfunktion einen Ausgangswert zu berechnen. Das heißt, der Task-Ausführungsblock **93** sorgt für die gewünschte Funktionalität der Task (welche vom Task-Funktionsblock **9** implementiert ist), während der Task-Freigabeblock **92** und der Task-Terminierungsblock **94** in logisch Null-Zeit (LZT, vgl. Abb. 3) ausführt. Der Task-Freigabeblock **92** ist konfiguriert, Eingangsdaten von einer Datenquelle (z. B. von einem Sensor **91** im Beispiel von Abb. 9 oder von einem vorangehenden Task-Funktionsblock **12**) zu empfangen und den Eingangswert vom Task-Ausführungsblock **93** abhängig vom empfangenen Wert zu setzten (z. B auf den empfangenen Sensorausgangswert). Der Task-Terminierungsblock **94** ist konfiguriert, den Ausangswert des Task-Ausführungsblocks **93** als Ausgangswert zu einer Datensenke (z. B. zu einen Aktor **95** im Beispiel von Abb. 9 oder zu einem anderen nachfolgenden Task-Funktionsblock **11**) bereitzustellen.

[0094] Aufgrund des Takts des durch den Trigger-Block (E-Machine) gewährleisteten Triggerings von den Blöcken **92**, **93**, **94**, ist die Zeitdifferenz zwischen dem Triggern des Task-Freigabeblocks **92** und dem Triggern des Task-Terminierungsblocks **94** immer gleich der LET des zugrunde liegenden Tasks. Im dem Fall, wo der Delay-Block **96** dem Task-Ausführungsblock **93** nachgeschalt et ist, können der Task-Freigabeblock **92** und der Task-Ausführungsblock **93** zum gleichen Zeitpunkt getriggert werden (d. h. der Task-Freigabeblock **92** und der Task-Ausführungsblock **93** können zusammengelegt werden).

[0095] Abhängig von der tatsächlichen E-Machine Implementierung und von den Anforderungen der tatsächlich verwendeten Simulationsumgebung, kann Datenfluss zwischen dem ersten und dem zweiten Trigger-Block **10a** und **10b** notwendig sein, und somit die Verzögerung des Datenflusses zwischen dem ersten Trigger-Block **10a** und dem zweiten Trigger-Block **10b** (z. B. mittels Unit-Delay z^{-1}).

[0096] Je nach Anwendung können der Sensorblock **91** und der Aktorblock **95** im Beispiel von Abb. 9 mit dem Task-Freigabeblock **92** bzw. dem Task-Terminationsblock **94** zusammengelegt werden. Dieser Fall ist im Beispiel von Abb. 10 dargestellt. Allerdings kann der Sensorblock **91** und/oder der Aktorblock benötigt werden, um Eingangs/Ausgangs-Aktivitäten

von Steuerungsaktivitäten der Task während der Programmentwicklung zu trennen.

[0097] Generell empfängt der Task-Freigabeblock **92** Eingangsdaten von einer Datenquelle (z. B. der Sensorblock **91** oder irgendein vorangehender Taksfunktionsblock **12**) und setzt den Eingangswert eines Task-Ausführungsblocks **93** entsprechend des empfangenen Werts, z. B. von einem Sensorausgangswert. Der Task-Terminierungsblock **94** stellt den Ausgangswert des Task-Ausführungsblocks **93** als Ausgangswert einer Datensenke (z. B. der Aktorblock **95** oder irgendein darauf folgender Task-Funktionsblock **11**) bereit.

Patentansprüche

1. Ein System zum Simulieren eines Echtzeitsystems mittels blockorientierter Simulation mit statischer Ausführungsreihenfolge; das System umfasst Folgendes:
ein Taktgenerator (**8**) der eine Zeitbasis für die Simulation bereitstellt;
einem ersten Task-Funktionsblock (**9**) der einen Task-Freigabeblock (**92**), einen Task-Ausführungsblock (**93**), einen Task-Terminierungsblock (**94**) und einen Verzögerungs-Block (**96**) beinhaltet;
einen ersten Trigger-Block (**10a**), der konfiguriert ist, den Task-Freigabeblock (**92**) zu einem ersten von dem Taktgenerator (**8**) bereitgestellten Zeitpunkt (t_0) zu triggern;
einen zweiten Trigger-Block (**10b**) der konfiguriert ist, den Task-Terminierungsblock (**94**) zu einem dritten (t_1, t_0 + LET) von dem Taktgenerator (**8**) bereitgestellten Zeitpunkt zu triggern;
wobei der Task-Ausführungsblock (**93**) konfiguriert ist, abhängig von einem Eingangswert in Übereinstimmung mit der gewünschten Transferfunktion einen Ausgangswert zu berechnen;
wobei der Task-Freigabeblock (**92**) konfiguriert ist, Daten von einer Datenquelle (**91**) zu empfangen und den Eingangswert des Task-Ausführungsblock (**93**) gemäß den empfangenen Daten zu setzen;
wobei der Task-Terminierungsblock (**94**) konfiguriert ist, den Ausgangswert des Task-Ausführungsblocks (**93**) als Daten für eine Datensenke (**95**) bereitzustellen;
wobei der erste Trigger-Block (**10a**) oder der zweite Trigger-Block (**10b**) konfiguriert ist, den Task-Ausführungsblock (**93**) zu einem zweiten von dem Taktgenerator (**8**) bereitgestellten Zeitpunkt (t_{0x}) zu triggern; und
wobei der Verzögerungs-Block (**96**) zwischen Task-Freigabeblock (**92**) und Task-Terminierungsblock (**94**) angeordnet ist.

2. Das System gemäß Anspruch 1,
wobei der erste Zeitpunkt (t_0) und der dritte Zeitpunkt (t_1) aufeinander folgende Zeitpunkte sind,
wobei die Zeit, die zwischen erstem Zeitpunkt (t_0) und drittem Zeitpunkt (t_1) vergeht, gleich der logischen Ausführungszeit (LET) des Task-Funktionsblocks (**9**) ist, und
wobei der zweite Zeitpunkt (t_{0X}) im Intervall zwischen erstem Zeitpunkt (t_0) und drittem Zeitpunkt (t_1) liegt, wobei das Intervall den ersten und den dritten Zeitpunkt (t_0, t_1) beinhaltet.

3. Das System gemäß Anspruch 1 oder 2,
wobei die Datenquelle (**91**) konfiguriert ist, Daten von einem, dem ersten Task-Funktionsblock (**9**) vorgeschaltenen, zweiten Task-Funktionsblock (**12**) als Eingangsdaten bereitzustellen, und
wobei die Datensenke (**95**) konfiguriert ist, die Ausgangsdaten einem dritten Task-Funktionsblock (**11**) bereitzustellen.

4. Das System gemäß einem der Ansprüche 1 bis 3, wobei ein weiterer Verzögerungs-Block (**96**) dem ersten Trigger-Block (**10a**) nachgeschaltet und dem zweiten Trigger-Block (**10b**) vorgeschalten ist, sodass alle Signale, die vom ersten Trigger-Block (**10a**) zum zweiten Trigger-Block (**10b**) fließen, verzögert werden.

5. Das System gemäß einem der Ansprüche 1 bis 4, das zusätzlich umfast:
einen weiteren Task-Funktionsblocks, der einen weiteren Task-Freigabeblock, einen weiteren Task-Ausführungsblock, einen weiteren Task-Terminierungsblock und einen weiteren Verzögerungs-Block enthält,
wobei der weitere Task-Freigabeblock, der weitere Task-Ausführungsblock und der weitere Task-Terminierungsblock auch vom ersten und/oder zweiten Trigger-Block (**10a**, **10b**) getriggert werden.

6. Ein Verfahren zum Simulieren eines Echtzeitsystems mittels eines blockorientierten Simulationssystems mit statischer Ausführungsreihenfolge der Blöcke; das Verfahren umfasst:
Bereitstellen eines ersten Task-Funktionsblocks (**9**), der einen Task-Freigabeblock (**92**), einen Task-Ausführungsblock (**93**), einen Task-Terminierungsblock (**94**) und einen Verzögerungs-Block (**96**) umfassst;
Triggern des Task-Freigabeblocks (**92**) zu einem ersten Zeitpunkt (t_0);
Triggern des Task-Ausführungsblocks (**93**) zu einem zweiten Zeitpunkt (t_{0X}) gleich oder nach dem ersten Zeitpunkt;
Triggern des Task-Terminierungsblocks (**94**) zu einem dritten Zeitpunkt (t_1, t_0 + LET) gleich oder nach dem zweiten Zeitpunkt (t_{0X});
Verzögern des Datenflusses zwischen dem Task-Freigabeblock (**92**) und dem Task-Ausführungsblock (**93**) und/oder Verzögern des Datenflusses zwischen dem Task-Ausführungsblock (**93**) und dem Task-Terminierungsblock (**94**);

wobei der Task-Ausführungsblock (**93**), sobald er getriggert wurde, abhängig von einem Eingangswert in Übereinstimmung mit der gewünschten Transferfunktion einen Ausgangswert berechnet;
wobei der Task-Freigabeblock (**92**), sobald er getriggert wurde, Daten von einer Datenquelle (**91**) empfängt und den Eingangswert des Task-Ausführungsblocks (**93**) gemäß empfangener Daten setzt;
wobei der Task-Terminierungsblock (**94**), sobald er getriggert wurde, den Ausgangswert eines Task-Ausführungsblocks (**93**) als Daten für eine Datensenke (**95**) bereitstellt.

7. Das Verfahren gemäß Anspruch 6, das zusätzlich umfasst:
Bereitstellung eines Taktgenerators (**8**), der den ersten Zeitpunkt (t_0), den zweiten Zeitpunkt (t_{0X}) und den dritten Zeitpunkt (t_1, t_0 + LET) bereitstellt.

8. Das Verfahren gemäß Anspruch 6 oder 7, wobei die Zeit, die zwischen dem ersten Zeitpunkt (t_0) und dem dritten Zeitpunkt (t_1, t_0 + LET) vergeht, gleich der logischen Ausführungszeit (LET) des Task-Funktionsblocks ist.

9. Das Verfahren gemäß einem der Ansprüche 6 bis 8,
wobei die Datenquelle (**91**) Daten von einem, dem ersten Funktionsblock (**9**) vorgeschaltenen, zweiten Task-Funktionsblock (**12**) als Eingangsdaten bereitstellt, und
wobei die Datensenke (**95**) die Ausgangsdaten einem dritten Task-Funktionsblock (**11**) bereitstellt.

Es folgen 4 Blatt Zeichnungen

Anhängende Zeichnungen

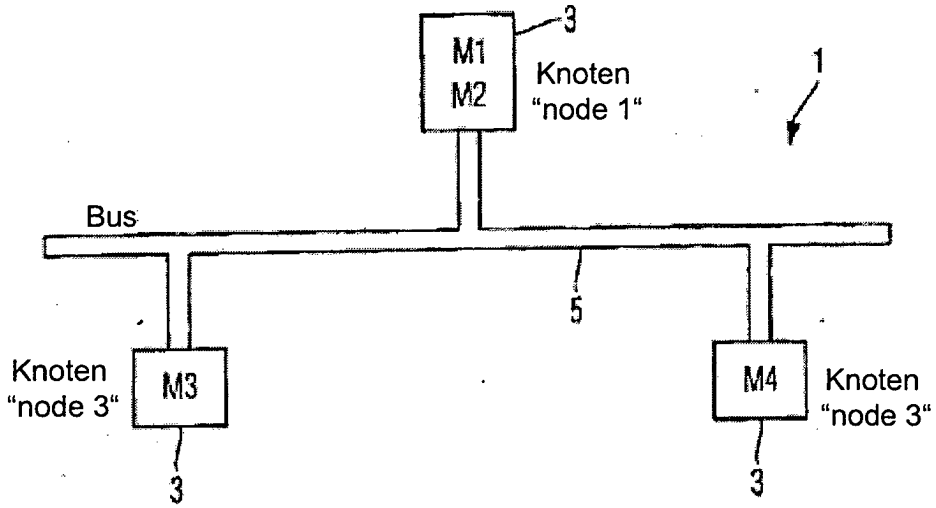


FIG. 1 (Stand der Technik)

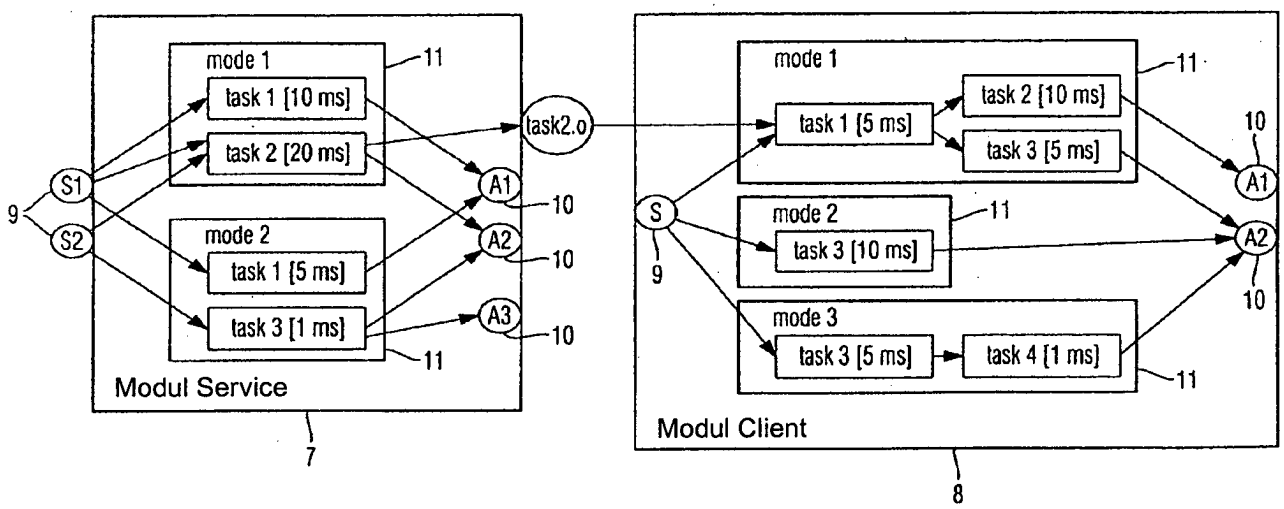


FIG. 2 (Stand der Technik)

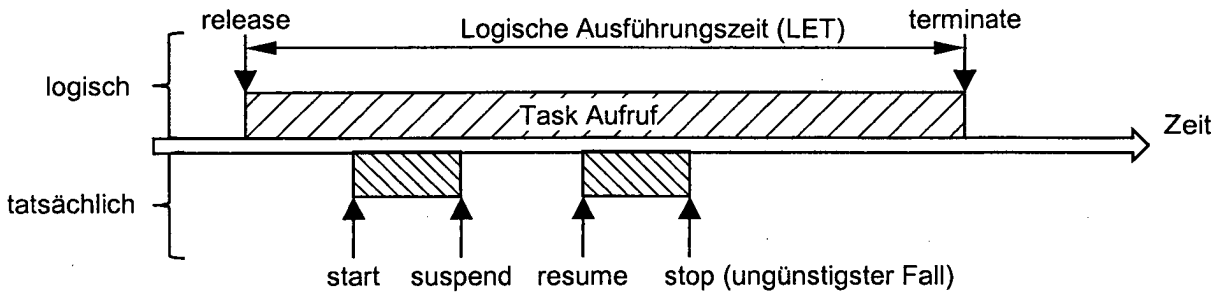


FIG. 3 (Stand der Technik)

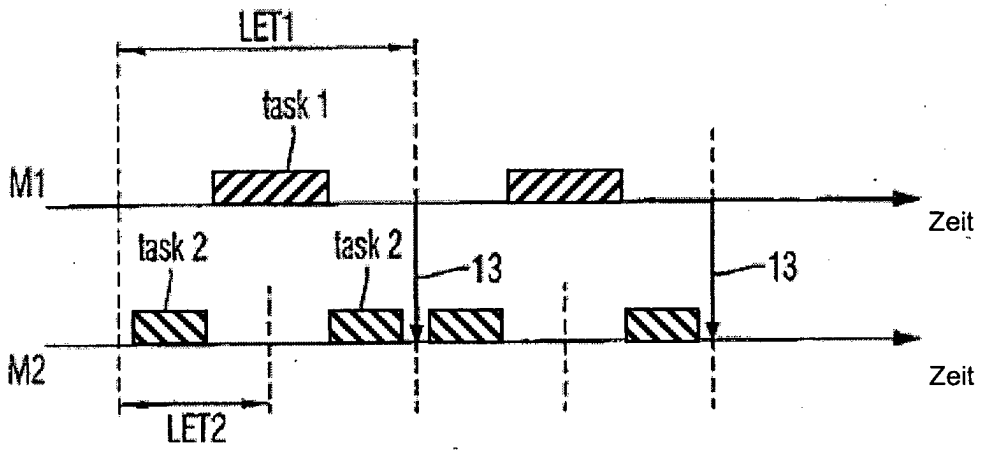


FIG. 4 (Stand der Technik)

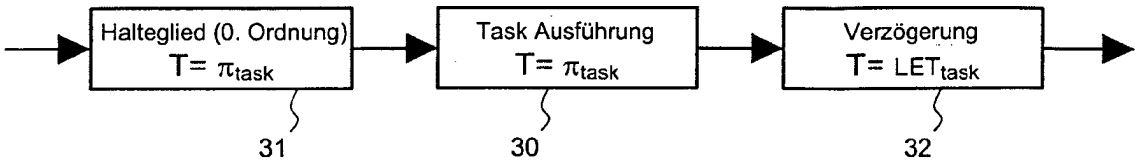


FIG. 5 (Stand der Technik)

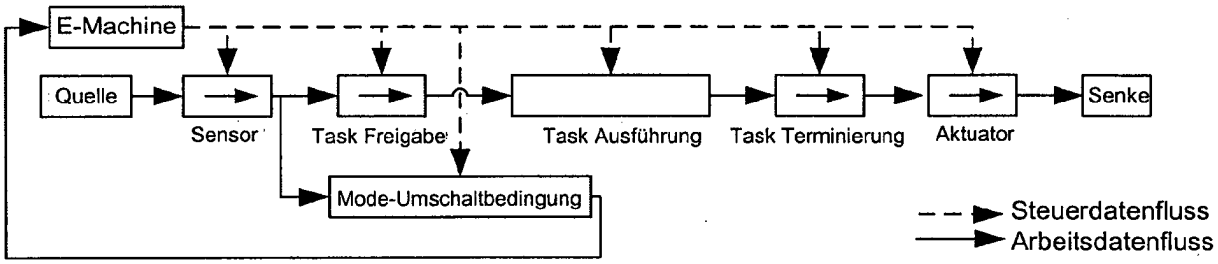


FIG. 6 (Stand der Technik)

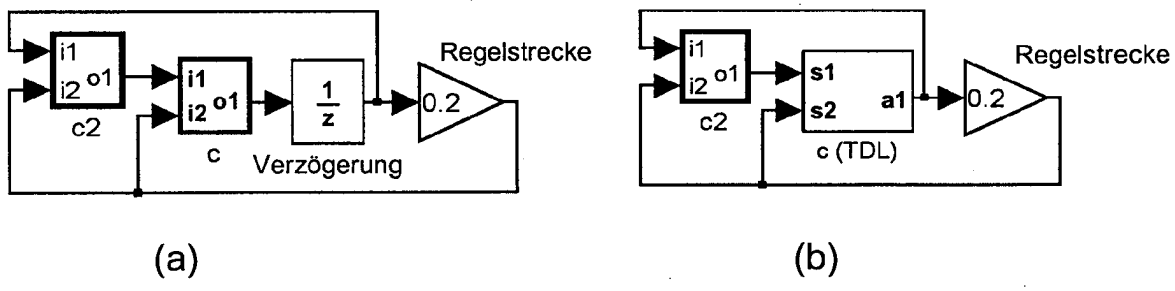


FIG. 7 (Stand der Technik)

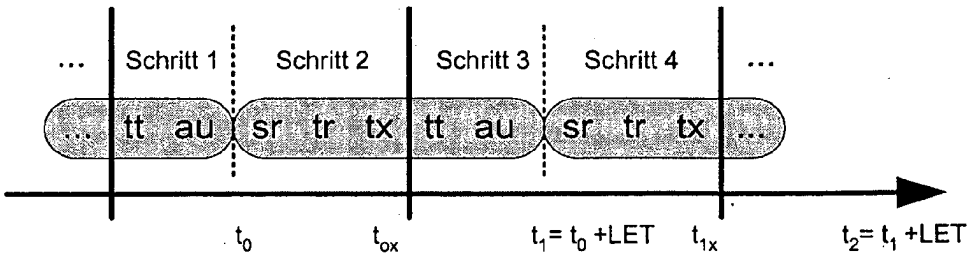


FIG. 8

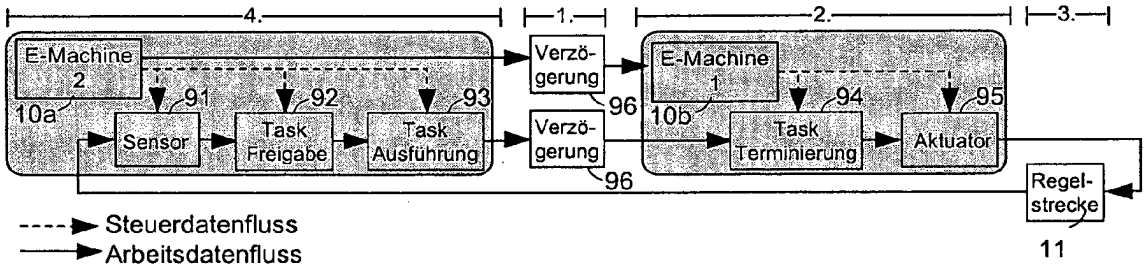


FIG. 9

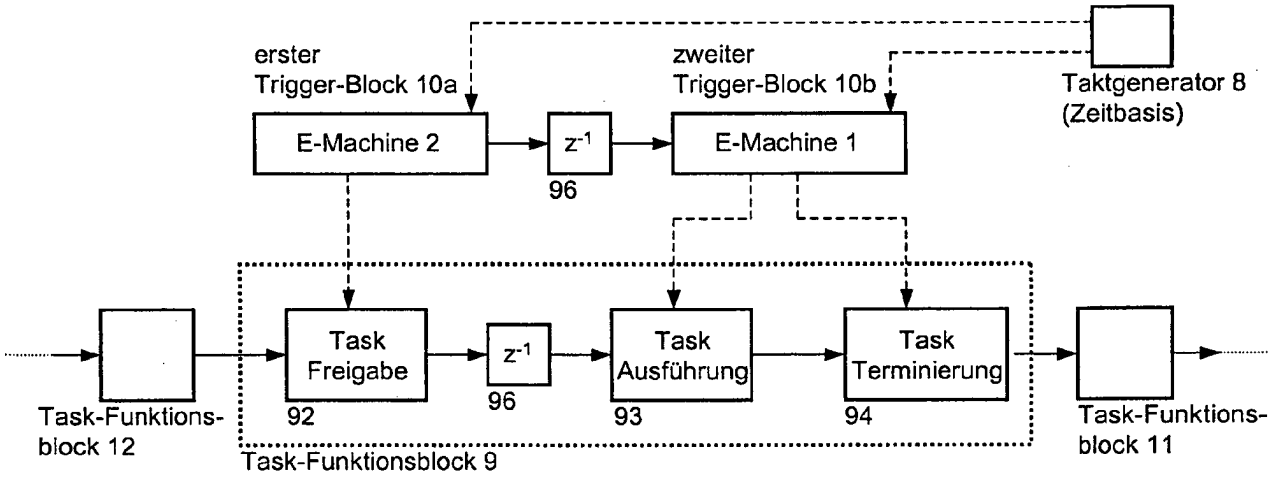


FIG. 10