Simulation mit abstrakten RTOS-Modellen in SystemC

Henning Zabel
Wolfgang Müller

C-LAB Report

Vol. 6 (2007) No. 4

Cooperative Computing & Communication Laboratory

ISSN 1619-7879

C-LAB ist eine Kooperation der
Universität Paderborn und der Siemens AG
http://www.c-lab.de
info@c-lab.de

C-LAB Report

Herausgegeben von
Published by

Dr. Wolfgang Kern, Siemens AG
Prof. Dr. Franz-Josef Rammig, Universität Paderborn

Das C-LAB - Cooperative Computing & Communication Laboratory - leistet Forschungs- und Entwicklungsarbeiten und gewährleistet deren Transfer an den Markt. Es wurde 1985 von den Partnern Nixdorf Computer AG (nun Siemens AG) und der Universität Paderborn im Einvernehmen mit dem Land NordrheinWestfalen gegründet.
Die Vision, die dem C-LAB zugrunde liegt, geht davon aus, dass die gewaltigen Herausforderungen beim Übergang in die kommende Informationsgesellschaft nur durch globale Kooperation und in tiefer Verzahnung von Theorie und Praxis gelöst werden können. Im C-LAB arbeiten deshalb Mitarbeiter von Hochschule und Industrie unter einem Dach in einer gemeinsamen Organisation an gemeinsamen Projekten mit internationalen Partnern eng zusammen.

C-LAB - the Cooperative Computing & Cooperation Laboratory - works in the area of research and de-velopment and safeguards its transfer into the market. It was founded in 1985 by Nixdorf Computer AG (now Siemens AG) and the University of Paderborn under the auspices of the State of North-Rhine Westphalia.
C-LAB's vision is based on the fundamental premise that the gargantuan challenges thrown up by the transition to a future information society can only be met through global cooperation and deep interworking of theory and practice. This is why, under one roof, staff from the university and from industry cooper-ate closely on joint projects within a common research and development organization together with international partners. In doing so, C-LAB concentrates on those innovative subject areas in which cooperation is expected to bear particular fruit for the partners and their general well-being.

ISSN 1619-7879

C-LAB
Fürstenallee 11
33102 Paderborn
fon: +49 5251 60 60 60
fax: +49 5251 60 60 66
email: info@c-lab.de
Internet: www.c-lab.de

©Siemens AG und Universität Paderborn 2007
Alle Rechte sind vorbehalten. Insbesondere ist die Übernahme in maschinenlesbare Form sowie das Speichern in Informationssystemen, auch auszugsweise nur mit schriftlicher Genehmigung der Siemens AG und der Universität Paderborn gestattet.

All rights reserved. In particular transfer of data into machine readable form as well as storage into information systems, (even extracts) is only permitted prior to written consent by Siemens AG and Universität Paderborn.

Zusammenfassung

An komplexe Steuersoftware werden harte Anforderungen bezüglich der Einhaltung von Zeitschranken gestellt. Die richtige Wahl eines Echtzeit-Schedulingverfahrens ist hier entscheidend. Die Verwendung von Interrupts in einem Echtzeitsystem erlaubt die zeitnahe Reaktion auf externe Ereignisse. Sie erschwert allerdings die Vorhersagbarkeit von Zeitschranken.

In diesem Bericht wird eine Methode zur abstrakten Simulation eines Echtzeitbetriebssystems (Real-Time Operating System/RTOS) in der Systembeschreibungssprache SystemC vorgestellt. Die Verwendung eines RTOS-Modelles in SystemC erlaubt die Evaluierung verschiedener Schedulingverfahren zu einem frühen Zeitpunkt im Entwurf des Systems. Interrupts unterliegen allerdings nicht einem RTOS-Schedulingverfahren und müssen daher gesondert behandelt werden.

Dieser Bericht beschreibt die Implementierung eines kanonischen RTOS-Modells zur frühen Analyse von Schedulingverfahren, das eine getrennte Modellierung von Task- und Interrupt-Scheduling erlaubt. Dieser Ansatz erlaubt die Trennung des hardwareabhängigen Interrupt-Scheduling in ein separates Modell und eine Evaluierung des Schedules, das sich rein auf Tasks bezieht.

1 Einleitung

Komplexe Steuersoftware, an die harte Anforderungen bezüglich der Einhaltung von Zeitschranken gestellt werden, müssen durch ein Echtzeitbetriebsystem (Real-Time Operating System/RTOS) mit einem vorhersehbaren (predictable) Zeitverhalten unterstützt werden [BB97]. Echtzeitbetriebssysteme besitzen daher im Gegensatz zu anderen Betriebssystemen die Möglichkeit zur Angabe von Ausführungszeiten und Zeitschranken (Deadlines) zu denen die Ausführung zwingend beendet sein muss. Diese Angaben fließen in dem RTOS mit in die Schedulingverfahren ein.

Ein sehr wichtiger Aspekt bei Echtzeitsystemen ist die Behandlung von Interrupts. Wie der Name sagt, unterbrechen sie die Ausführung von anderen Echtzeit-Prozessen und erschweren damit die Vorhersagbarkeit der Zeitschranken. Sie werden durch externe Signale (der Hardware) ausgelöst und ermöglichen die zeitnahe Reaktion auf Ereignisse von Timern und E/A-Bausteinen, wie z.B. ,,Puffer leer" (Ausgabe) oder ,,Puffer voll" (Eingabe). Es gibt verschiedene Ansätze zur Behandlung dieser Ereignisse [BB97, S.14ff]: Zunächst kann man Interrupts deaktivieren und über wiederholtes Lesen (Polling) die entsprechenden E/A-Register abfragen. Dieser Ansatz erhöht zwar die Vorhersagbarkeit, aber ständiges (aktives) Anfragen der E/A z.B. beim Warten auf eine Beendigung einer Übertragung kann die Prozessorlast erhöhen. Insbesondere bei Low-Power-Anwendungen ist dies aber unerwünscht. Die Behandlung der Ereignisse mit Interrupt-Service-Routinen (ISR) ermöglicht eine zeitnahe Reaktion auf einen Interrupt-Request (IRQ) mit kurzen Verzögerungen. Kurze Antwortzeiten sind neben der Vorhersagbarkeit ein wichtiges Kriterium für Echtzeitsysteme. Um den Einfluss auf die Echtzeit-Prozesse zu minimieren, sollten ISRs nur die nötigsten Anweisungen zur Behandlung des IRQ beinhalten. Komplexere Aufgaben, wie z.B. die Berechnung von Prüfsummen bei empfangenen Paketen, werden daher meistens als eigenständige Task implementiert und durch die ISR aktiviert. Da durch die Behandlung von ISRs das aktive Warten entfällt, kann man dann durch Einsatz eines RTOS während dieser Wartezeit andere Tasks ausführen oder den Prozessor in einen energiesparenden Wartezustand setzen.

Mit Hilfe von geeigneten Schedulingverfahren muss das Einhalten der Zeitschranken einzelner Tasks eines RTOS sichergestellt werden. Bei vielen Echtzeit-Schedulingverfahren gibt es so genannte Schedulability-Tests mit denen bereits vor der Implementierung geprüft werden kann, ob sich eine Menge von Tasks mit gewissen Zeitschranken überhaupt schedulen lässt. Voraussetzung ist aber, dass man die Laufzeiten der einzelnen Tasks auf dem Zielsystem kennt. Solche Laufzeiten lassen sich mit Hilfe von Instruction-Set-

Simulationen für die entsprechende Zielplattform oder mit einer *Worst-Case Execution Time Analysis* (WCET-Analyse) [PB00] ermitteln. Bei WCET-Analysen wird eine sichere obere Schranke für die Laufzeit einer Tasks oder Teilen einer Funktion auf einem gegeben Zielprozessor ermittelt. Die Laufzeit dient zur Planung der für den Task notwendigen Rechenzeit. Eine Schwierigkeit bei der Analyse ist die zuverlässige Vorhersage, wie häufig Schleifen im Programm durchlaufen werden. In vielen Arbeiten werden daher obere Laufzeitschranken für die Anzahl der Durchläufe im Quelltext annotiert. In dem vorliegenden Bericht werden die Tasks zur Analyse während der Simulation ausgeführt, womit die Notwendigkeit der Vorhersage entfällt.

2 Simulation eines RTOS

Eine naheliegende Möglichkeit zur Analyse und Validierung von eingebetteter Software inklusive dem verwendeten RTOS ist die Simulation mit Instruction-Set-Simulatoren oder Debuggern für einen entsprechenden Zielprozessor. Solche Werkzeuge sind meist in die Entwicklungsumgebungen für die Prozessoren eingebaut und werden von dem Hersteller oder Drittanbietern zur Verfügung gestellt. Mit diesen Werkzeugen lassen sich Veränderungen von Register- und Variableninhalten instruktionsgenau simulieren. Das erlaubt sowohl eine Überprüfung der funktionalen Eigenschaften, als auch Laufzeitmessungen von einzelnen Funktionen. Für dieses Vorgehen ist es allerdings erforderlich, dass man sich frühzeitig für ein konkretes RTOS entscheidet. In frühen Entwurfsphasen führt diese Festlegungen zu Einschränkungen bei der späteren Entwicklung. Beispielsweise kann sich im weiteren Verlauf herausstellen, dass die Kommunikationskonzepte oder Schedulingverfahren des ausgewählten RTOS für die Anwendung unzureichend sind. Eine reine funktionale Simulation würde zwar diese Problematik umgehen, allerdings haben Schedulingentscheidungen und Laufzeiten in der Implementierung einen großen Einfluß auf die Echtzeitfähigkeit: Ein funktional korrekt implementierter Regler ist nutzlos, wenn er nicht innerhalb seiner Abtastrate berechnet werden kann.
Um im frühen Entwurfsstadium neben der funktionalen Simulation Schedulinguntersuchungen durchzuführen, ohne sich jedoch auf ein konkretes RTOS festzulegen, gibt es die Möglichkeit der Analyse mit abstrakten RTOS-Modellen. Ein abstraktes RTOS-Modell ist eine Reduktion der RTOS-Funktionalität auf die Grundbestandteile Taskwechsel und Scheduling. Diese lassen sich mit einer kleinen Programmierschnittstelle (Application Programming Interface/API) mit einer Handvoll Funktionen darstellen. Da man davon ausgeht, dass sich alle gängigen RTOSs auf diese Grundfunktionen reduzieren

lassen und umgekehrt diese Grundfunktionen auf jedem RTOS implementierbar sind, bezeichnet man ein solches Modell auch als kanonisch. Um die Simulationen effizienter, d.h. schneller, durchzuführen, werden anstatt von Instruktion-Set-Simulationen Simulationen mit Hilfe von Systembeschreibungssprachen (System Level Design Language/SLDL) wie SystemC oder SpecC durchgeführt [GZD+00, Sys08]. Zur Simulation werden einzelne Programme in eine Menge von atomaren Blöcken unterteilt und direkt auf dem Simulationsrechner ausgeführt.
Es gibt bereits verschiedene Implementierungen solcher Simulationen in den genannten SLDLs. Schwerpunkt der Simulation in dem vorliegendem Bericht ist die getrennte Modellierung des Schedules von Tasks und Interrupt-Service-Routinen innerhalb der Simulation. In vielen Arbeiten werden Tasks und ISR gleichwertig behandelt (s.h. Abschnitt 3). Interrupts werden aber nicht aufgrund von Schedulingentscheidung ausgelöst, sondern aufgrund von Ereignissen in der Hardware. Sie müssen deshalb zumindest als hochpriorisierte Tasks simuliert werden. Allerdings basieren nicht alle Schedulingverfahren auf Prioritäten (z.B. Round Robin oder statisches Scheduling). Die Simulation von Interrupt-Service-Routinen als Tasks würde hier zu Fehlern in der Ausführungsreihenfolge und damit zu einer funktional falschen Simulation führen. Desweiteren müssen die Prioritäten für jedes Simulationsmodell und Scheduler jedes mal neu vergeben werden, was den Modellierungsaufwand unnötig erhöht. Die Trennung der Interrupt-Schedulings hingegen, unterbindet diese Probleme und ermöglich darüber hinaus die Implementierung des Schedulings als getrenntes Hardwaremodell.

2.1 Segmentierte Simulation

Die Simulation von Software in einem Instruction-Set-Simulator (ISS) und die abstrakte Simulation mit Hilfe eines RTOS-Modells in SystemC (oder SpecC) unterscheiden sich grundlegend: Instruction-Set-Simulatoren beinhalten ein komplettes Prozessormodell und simulieren die Ausführung von Assembler-Anweisungen auf den Registern der CPU. Die Simulation der Funktion und die Ermittlung der dafür notwendigen Taktzyklen, d.h. der Ausführungszeit, erfolgt in einem Schritt (vergleiche Abbildung 1). Die Simulationen sind taktgenau und Unterbrechungen von der Hardware werden exakt nachgebildet. Ebenso werden Laufzeiten von Pipelines und Speicherzugriffen berücksichtigt.
Bei der Simulation von Software-Tasks in einer SLDL werden die einzelnen Tasks zunächst in atomare Blöcke unterteilt und jeder Block mit einer Ausführungszeit annotiert. Manche Arbeiten fügen zu diesen Blöcken zusätz-

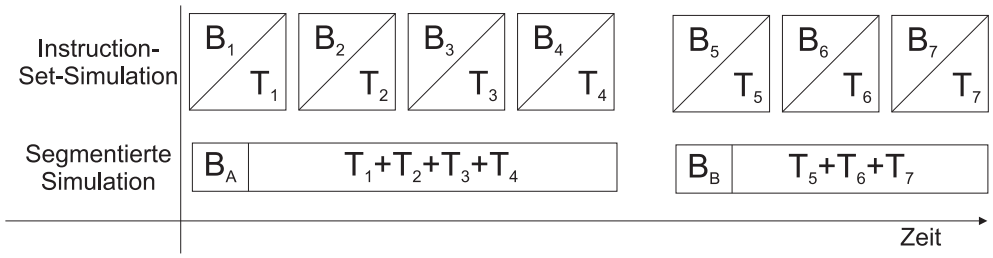

Abbildung 1: Kombination von Verhalten (B für "Behaviour") und Zeit (T für "Time") bei Instruction-Set-Simulationen und bei der segmentierten Simulation von Blöcken.

lich Annotationen mit dem Energieverbrauch hinzu. Zur Simulation wird der Quelltext direkt für den Simulationsrechner übersetzt und ausgeführt. Die Unterteilung der Software in Blöcke erschwert jedoch die Simulation von Unterbrechungen. Da die Blöcke atomar sind, können Unterbrechungen des Verhaltens nur nach einem Block erfolgen. Es ist daher notwendig Blöcke beim Zugriff auf globale Variablen und Systemaufrufen zu unterteilen, um eine funktional korrekte Simulation zu gewährleisten.
Die Simulation der Zeit, d.h. das Warten nach Ausführen des zugehörigen Verhaltens, kann allerdings mit gleicher Granularität wie bei einer ISS simuliert werden. Die Implementierung erlaubt die Unterbrechung der Wartesimulation und Wiederaufnahme zu einem späteren Zeitpunkt, wodurch Laufzeiten durch Interrupts oder höher priorisierte Tasks mit simuliert werden können.

2.2 Kanonisches Modell zur Synthese

Beim Systementwurf mit einer SLDL liegt die Verhaltensbeschreibungen des Systems als Menge von Tasks vor, die über Kanäle miteinander kommunizieren. Im Rahmen eines Hardware/Software-Codesign wird entschieden, welche Prozesse als Hardwarebestandteile und welche als Software realisiert werden. Die Hardwarekomponenten werden bis zur Register-Transfer-Ebene detailiert und anschließend durch eine Hardwaresynthese implementiert. Die Softwareanteile werden verschiedenen Prozessoren zugewiesen und die Kommunikation zwischen Tasks muss über die gegebene E/A der Prozessoren realisiert werden.

Typischerweise erfolgt die konkrete Implementierung von Software-Tasks manuell mit Hilfe von Prototyping-Boards oder den oben genannten Instruction-Set-Simulatoren. Yu stellt in seiner Dissertation mit dem Titel ,,Software

Synthesis for System-on-Chip" eine schrittweise Vorgehensweise zur Abbildung der reinen Verhaltensbeschreibung eines Modells bis hin zur finalen Implementierung in C vor [Yu05]. Die Arbeit wurde mit der SLDL SpecC realisiert [GZD+00]. Die Grundidee hinter seiner Arbeit ist die Verwendung eines abstrakten RTOS-Modells, das grundlegende Operationen für Prozesswechsel und Semaphoren beinhaltet. Die zu einem Prozessor zugeordneten Tasks müssen zur Ausführung sequentialisiert werden. Dazu werden Tasks an definierten Stellen um diese Grundoperation erweitert, so dass darauf basierend ein Schedulingverfahren für die Tasks entwickelt und evaluiert werden kann. Dieses Vorgehen ist ebenfalls in [GYG03] beschrieben. Das abstrakte RTOS-Modell existiert sowohl als SpecC-Modell zur Simulation, als auch als konkrete Realisierung auf einem realen RTOS. Das entwickelte Schedulingverfahren entspricht in der realen Implementierung dem Scheduler. Die Kommunikation zwischen den Tasks wird ebenfalls über RTOS-Aufrufe mittels Events abgebildet. Kommunikation zwischen Prozessoren (PE für ,,processing element") wird über Busse realisiert, die über Bustreiber von einem PE angesprochen werden. Die Ankunft neuer Daten wird über Signale (Interrupts) gemeldet. Die Behandlung der Signale werden durch Interrupt-Service-Routinen übernommen, die ebenfalls als Task in die Menge der zu sequentialisierenden Tasks aufgenommen werden.
Das Einfügen der Taskwechseloperationen zusammen mit der Schedulingstrategie und das Ersetzen der abstrakten Kanalkommunikation durch Bustreiber und ISR resultiert in einem SpecC-Modell, das stark an die tatsächliche Programmierung eines Microcontrollers oder Prozessors angelehnt ist. Mit Hilfe der Implementierung des abstrakten RTOS-Modells auf Basis einer API[1] eines realen RTOS, lässt sich in weiteren Schritten das Modell in eine reale Implementierung für einen Zielprozessor überführen.

2.3 Simulation mit RTOS Modellen

Im Gegensatz zu dem obigen Ansatz, eine auf Basisfunktionen reduzierte Darstellung eines RTOS zur Software-Synthese zu verwenden, implementieren andere Arbeiten den POSIX-Standard [PAS+06, PAV+05] oder μ - ITRON-OS-Standard [AKS+04, HSTI05] in SystemC. Die Implementierung dieser Standards ermöglicht die Zeitanalyse von Software (oder Softwarekomponenten), die die API dieser Standards verwenden oder verwenden sollen. Im Idealfall ist keine Modifikation der Software notwendig. Diese Modelle kann man zwar nicht als kanonisch bezeichnen, sie abstrahieren allerdings dennoch von einer konkreten Hardware und einem RTOS. Da sich viele Implementie-

[1] Aplication Programming Interface

rungen an einen dieser Standards halten, schränkt man sich lediglich auf den Standard, nicht aber auf ein spezielles RTOS ein. Ein Wechsel der letztendlichen Zielarchitektur ist somit auch noch zu einem späteren Zeitpunkt möglich.

3 Existierende Arbeiten

In [AKS+04, HSTI05] wird ein Modell zur Co-Simulation von Hardware und Software in SystemC beschrieben. Der Zustand eines Prozesses wird als zyklisches, eins-sicheres Petrinetz mit atomaren Transitionen dargestellt. Die Zustandsübergänge werden durch Events des RTOS-Kerns ausgelöst. Transitionen sind mit Zeiten und Energiewerten annotiert, die entsprechend während einer Transition konsumiert werden. Der Kern ist basierend auf dem μ-ITRON-OS-Standard realisiert und die Hardware wird mit einem ,,Bus-Functional-Model (BFM)" beschrieben. Zur Durchführung von E/A-Operationen rufen die Prozesse Funktionen des BFM auf, welches die Aufrufe in zyklengenaue Busaufrufe umsetzt. Als Scheduler wird ein Round-Robin-Verfahren benutzt.

Bei der Simulation eines RTOS in SystemC ist eine Synchronisation zwischen den Tasks und dem RTOS-Modell notwendig. Die dafür notwendigen Kontextwechsel wirken sich jedoch negativ auf die Simulationsgeschwindigkeit aus. In [RBMD03] wird ein Scheduling-Algorithmus für zeitbehaftete funktionale Simulation von nebenläufiger Software in SystemC vorgeschlagen. Dieser reduziert die Häufigkeit der Kontextwechsel. Die genaue Funktionsweise der Synchronisation bleibt allerdings unklar, genauso wie die Auswirkung auf die Genauigkeit der Simulation bezüglich der Zeit.

Auf die genaue Modellierung der internen Zeiten für einen Taskwechsel, d.h. dem Laden und Speichern des Kontextes und Treffen der Scheduling-Entscheidung, wird in [MPC04] eingegangen. Zum Vergleich wurde der Scheduler selbst als eigenständiger SystemC-Thread und über Funktionsaufrufe durch die Tasks modelliert. Der zweite Ansatz reduzierte die Anzahl notwendiger Kontextwechsel innerhalb der SystemC Simulation.

In [HRR+06] wird ein Verfeinerung von TLM-Modellen in SystemC vorgestellt. Die Verhaltensbeschreibungen für die Softwareanteile des Systems werden hierzu gruppiert. Jeder Gruppe wird ein RTOS-Modell zugeordnet. Damit wird die Simulation einer Multiprozessorumgebung auf abstraktem Niveau erlaubt. Ebenso wie in der Arbeit von Yu, definiert das RTOS-Modell eine API mit grundlegenden Befehlen zur Synchronisation von Tasks. Wait-Anweisung zur Modellierung der Zeit werden hierbei durch entsprechende

Aufrufe im RTOS-Modell ersetzt, um den Zustand der Task zu bestimmen. Ebenso wird bei der Kommunikation zwischen Tasks bei blockierenden Schreib- und Lese-Anweisungen verfahren.

[HKH04] stellt ebenfalls eine RTOS-Erweiterung für SystemC vor. Durch Ableiten einer Basisklasse können eigene Scheduler entwickelt werden. Über Callback-Funktionen wird hierzu der aktuelle Zustand der Tasks an den Scheduler weitergeleitet. Eine explizite Behandlung von Interrupts gibt es nicht. In der Dissertation von Klaus [Kla05] wird dieser Ansatz zur Entwurfsraumexploration für eingebette Systeme verwendet. In [HK07] wird der Ansatz am Beispiel einer mobilen Robotersteuerung evaluiert. Mit Hilfe der Gumbel-Verteilung zur Modellierung der stochastischen Verteilung von Ausführungszeiten werden verschiedene Schedulingstrategien untersucht.

In [KBR06] wird eine automatisierte Methode beschrieben, um eine Verfeinerung von SystemC-Modellen vorzunehmen. Die Aufbereitung der SystemC-Modelle erfolgt mit einem SystemC-Parser, der die Modelle in eine XML-Repräsentation überführt. Je nach Bedarf kann die Verfeinerung der Kommunikation durch Modelle auf PV-T-Ebene (Programmers View with Timing) oder auch durch CA-Modelle (Cycle Accurate) erfolgen. Zur Abbildung der SystemC-Prozesse auf ein RTOS werden dann die SystemC-Konstrukte durch Aufrufe der API eines RTOS ersetzt. Dies kann sowohl die API eines realen RTOS sein, als auch die eines Simulationsmodells.

In [PAS+06, PAV+05] wird ein POSIX kompatibles RTOS-Modell für SystemC vorgestellt. Zur Simulation werden die Tasks in Segmente unterteilt. Die Ausführungszeit jedes Segmentes wird an einen Time-Manager weitergeleitet. Bei Unterbrechungen in Form von Interrupts wird unterschieden, ob der Zeitpunkt des Auftretens vorhersagbar ist oder nicht. Vorhersagbare Unterbrechungen sind zum Beispiel Timer oder Timeouts. Diese werden direkt vom Time-Manager berücksichtigt. Ein Segment wird zwar immer in einem Stück simuliert, die angegebene Simulationszeit wird aber durch die Unterbrechung aufgeteilt und um die Simulationszeit der Unterbrechung ergänzt. Auf diese Weise wird sichergestellt, dass Zeitpunkte für Kommunikation korrekt simuliert werden. Um die funktionale Simulation der Zugriffsreihenfolge auf globale Variablen und Kommunikation zu verbessern, werden Systemaufrufe und Zugriffe auf globale Variablen in einem eigenen Segmenten bzw. am Start oder Ende eines Segmentes realisiert. Der Simulationsfehler gegenüber einer ISS wird mit maximal 8% angegeben. Aufbauend auf diesen Arbeiten wird in [QPV+06] ein TLM-Modell zur Anbindung von Hardware Komponenten an ein RTOS-Modell vorgestellt. Die Hardwaremodelle können über ein TLM-Businterface Interrupts an das RTOS-Modell senden. Ein spezieller SystemC-Thread überwacht diese Anforderung und instantiiert entsprechend

einen POSIX-Thread zur Simulation der zugehörigen ISR.

Die vorgestellten Arbeiten zeigen, dass die abstrakte Simulation mit einem RTOS-Modell sehr genaue Zeitanalysen in frühen Entwurfsphasen erlaubt. Die Simulationen lassen sich je nach Abstraktion der Kommunikation um Grössenordnungen bis zum 800-fachen schneller Ausführen, als komplexe Instruction-Set-Simulationen. Das ermöglicht neben eingehenderen Analysen verschiedener Scheduling-Strategien auch die Simulation komplexer Systeme mit vielen Prozessoren. Alle Arbeiten haben gemeinsam, dass sie die Software zur Simulation zunächst in Segmente unterteilen und diesen eine Ausführungszeit zuweisen, so wie es bereits in 2.1 beschrieben wurde. Eine weitere Gemeinsamkeit ist, dass alle Arbeiten zur Wahl des nächsten zu simulierenden Segmentes bekannte Scheduling-Strategien aus konkreten RTOS-Implementierungen verwenden. Allerdings schränken sich einige Arbeiten auf bestimmte Verfahren ein. In der Auswahl der Segmente müssen auch Segmente aus ISRs berücksichtigt werden, die allerdings in der Realität nicht dem Software-Scheduling unterliegen. Diese Tatsache wird in den meisten Arbeiten nicht oder nur unzureichend berücksichtigt. In diesem Zusammenhang ist besonders die Simulation der annotierten Wartezeit von besonderer Bedeutung: Die Simulation des Wartens muß zur zeitgenauen Simulation von ISRs analog zur Ausführung von Instruktionen auf einem Prozessor unterbrechbar sein. Dies erfordert eine genaue Synchronisation der Simulation der Segmente und der annotierten Wartenzeiten mit der Ausführung des Schedulers. Die genaue Beschreibung dieser Synchronisation fällt in den vielen Arbeiten teils zu kurz oder schwer verständlich aus.

Diese Tatsachen motivierten uns zu der Implementierung einer eigenen RTOS-Bibliothek in SystemC, die die obigen Punkte erfüllt. Das Modell wurde zur korrekten Simulation von ISR um einen weiteren Scheduler für ISR erweitert, der unabhängig von dem Software-Scheduler definiert werden kann. Erste kleinere Beispiele verdeutlichen die Verwendung der vorgestellten API und die Annotation von Ausführungszeiten.

4 RTOS-Implementierung in SystemC

Im Folgenden stellen wir unsere SystemC-Implementierung zur abstrakten Simulation eines RTOS in SystemC vor. Basis bildet ein *sc_rtos_context* mit Funktionen zur Modellierung von Taskwechseloperationen. SystemC-Threads, die eine Task oder eine Interrupt-Service-Routine modellieren, werden einem Kontext zugeordnet. Durch Annotieren der SystemC-Threads mit Funktionen der Kontext-API werden Aktionen wie das Warten auf einen Interrupt-Request oder Warten auf Signale anderer SystemC-Threads auf einen RTOS

Funktionaufruf abgebildet. Mit Hilfe dieser Aufrufe ist es dem Kontext möglich immer nur einer Task die Aktivität zu geben, d.h. die Ausführung der angemeldeten Tasks und Interrupt-Service-Routinen zu sequentialisieren. Das ist eine zwingende Voraussetzung zur Simulation des Verhaltens verschiedener Routinen auf einem Prozessor.
Die Wahl der nächsten zu simulierenden Task wird, wie auch bei einer konkreten RTOS-Implementierung, durch einen Scheduler ermittelt. Die API erlaubt die Implementierung verschiedener Scheduling-Strategien mittels eines Interfaces. Die Aufrufe und die Priorisierung von ISRs werden durch die Hardware eines Prozessors festgelegt. Unser Modell besitzt daher zur korrekten Simulation von ISRs einen weiteren Scheduler für ISRs. In anderen Arbeiten wird nicht zwischen dem Schedulen von Tasks und Interrupt-Service-Routinen unterschieden. Letztere werden allerdings auf einer realen CPU, nicht von dem Betriebssystem geschedult, sondern von der Logik der CPU selbst, also der Hardware. Ferner besitzen in der Realität ISRs eine höhere Priorität als Tasks. Die Trennung in zwei Scheduler erlaubt damit die Auslagerung des Interrupt-Scheduling in ein separates CPU Modell. Beide Scheduler können dann unabhängig voneinander betrachtet und evaluiert werden.
Wird eine Interrupt-Service-Routine in der Simulation lauffähig, ist es für eine funktional korrekte Simulation zwingend erforderlich, dass diese zum nächst möglichen Zeitpunkt die aktuelle Task unterbricht und ausgeführt wird. Bei der Simulation mit SystemC sind diese Unterbrechnung nur innerhalb der SystemC-*wait*-Anweisungen möglich. Unsere RTOS-Modellierung stellt die zeitgenaue Simulation dieser Bedingung sicher.
Die Trennung des Scheduling von Interrupt-Service-Routinen und Tasks ermöglicht in der Summe die hardwareunabhängige Evaluierung von Schedulingverfahren und liefert gleichzeitig eine funktional korrekte Simulation.
Im Folgenden stellen wir die Implementierung unserer SystemC RTOS-Bibliothek vor. In 4.1 werden zunächst einige Begriffe definiert, die das Verständnis der folgenden Beschreibung erleichtern sollen. In Abschnitt 4.2 wird eine Übersicht über die Klassen der RTOS-Bibliothek gegeben. Die Synchronisation der Tasks und ISRs untereinander und mit dem Kontext wird in Abschnitt 4.3 und 4.5 erläutert. Eine detaillierte Beschreibung der kanonischen API folgt in Abschnitt 4.6.

4.1 Der Task-Begriff

SystemC
Die SystemC-Bibliothek implementiert einen event-basierten Simulationskern zur Co-Simulation von elektronischer Hardware und Software in C++

auf verschiedenen Abstraktionsebenen. Das Verhalten des Systems wird in Prozessen implementiert, die über Kanäle miteinander kommunizieren. Die Prozesse synchronisieren sich über Events. Events können entweder direkt oder durch Änderungen in den Kanälen erzeugt werden. Prozesse reagieren sensitiv auf Events, d.h. sie werden beim Auftreten eines Events als lauffähig markiert. Das Warten findet innerhalb einer SystemC-*wait*-Anweisung statt. Die Simulation erfolgt in sogenannten Δ-Zyklen. In jedem Zyklus werden alle lauffähigen Prozesse solange ausgeführt, bis die Simulationszeit inkrementiert werden kann. Dabei können neue Events für den aktuellen oder für zukünftige Zeitpunkte erzeugt werden. Solange Events für aktuellen Zeitpunkt existieren werden weitere Δ-Zyklen ausgeführt. Anschließend wird die Zeit auf den nächstfolgenden Zeitpunkt, für den ein Event erzeugt wurde, inkrementiert. Die pseudo-parallele Ausführung von SystemC-Prozessen ist in dem SystemC-Kern mittels kooperativen Multi-Tasking implementiert. Der Kontext-Wechsel von einem zum anderen Prozess findet nur innerhalb der *wait*-Anweisungen statt, d.h. wenn sicher ist, dass ein Prozess zur Zeit keine weiteren Aufgaben ausführen kann.

Betriebssystem und Tasks
Ein Betriebssystem ermöglicht die Ausführung mehrerer paralleler Aufgaben, die im folgenden als Tasks bezeichnet werden. Prozessoren können die Ausführung von Tasks bei Auftreten eines IRQs unterbrechen. Als Reaktion auf einen IRQ wird die Bearbeitung einer ISR gestartet. Der Wechsel von der Ausführung einer Task zu einer anderen wird hier ebenfalls als Kontextwechsel bezeichnet.
Ein Prozessor mit nur einer Ausführungseinheit kann immer nur eine Task oder ISR gleichzeitig ausführen. Zur Nachbildung von Paralellität werden Tasks daher mit Hilfe von einem Betriebssystem stückweise hintereinander, also sequentiell, ausgeführt.

Simulation von Tasks
In der Simulation bilden wir jeden Task und jede ISR auf einen SystemC-Prozess ab. Die konkrete Implementierung erfolgt hier als *SC_THREAD*. Die Ausführung der SystemC-Prozesse ist allerdings, wie oben beschrieben, quasiparallel. Diese Paralellität bezieht sich auf das Fortschreiten der Zeit innerhalb der SystemC Simulation, nicht auf die Ausführung der Simulation selbst. Die Aufgabe eines RTOS-Modells in SystemC ist es, die Ausführung der einzelnen SystemC-Threads, die eine Task oder ISR modellerien, über geeignete Synchronisation zu sequentialisieren. Unsere Implementierung verwendet hierzu SystemC-Events. Wie oben beschrieben werden Tasks und ISRs zur Simulation segmentiert. Die Simulation der Wartezeit eines Segmentes er-

folgt innerhalb von SystemC durch Warten auf Timer-Events. Das Warten impliziert Kontextwechsel der SystemC-Prozesse. Dadurch lassen sich Kontextwechel eines RTOS zwischen Tasks auf den Kontextwechsel der SystemC-Prozesse abbilden. Für Simulation von ISRs gilt das gleiche, obwohl es sich beim Aufruf einer ISR nicht um einen Kontextwechsel des RTOS handelt.
Die Segmente der Tasks und ISRs werden durch diese Abbildung atomar, da Task-Wechsel und IRQs nur am Ende der Verhaltenssimulation stattfinden können. Bei geigneter Wahl der Segmente ist dies aber keine Einschränkung für eine funktional korrekte Simulation.

4.2 RTOS-Klassen

In Abbildung 2 ist eine Übersicht der Klassen zu finden, die zur Simulation des RTOS benötigt werden. Angelehnt an die Kontext-Klasse von SystemC selbst, gibt es die Klasse *sc_rtos_context*. Alle Tasks, die sequentialisiert ablaufen sollen, werden an dieser Klasse angemeldet und in einer Liste aus *sc_rtos_context_tcb* Objekten gespeichert. Der Klasse sind weiterhin die beiden Scheduler zugeordnet. Innerhalb der Klasse wird ein spezielles Event *event_schedule* verwendet, um das Scheduling anzustoßen.
RTOS-Tasks und ISR werden jeweils als SystemC-Thread innerhalb der Klasse *sc_rtos_module* implementiert. Der Klasse wird beim Instanziieren der zugehörige Kontext übergeben und damit dem Kontext eindeutig zugeordnet. Die Klasse *sc_rtos_module* bietet dieselbe API von Prozessfunktionen wie die Kontextklasse an, leitet die Aufrufe allerdings an den zugeordneten Kontext weiter. Die Besonderheit der Klasse liegt in den überschriebenen *wait*-Anweisungen. Die Funktionen wurden jeweils um die Aufrufe *task_deactivate* und *task_activate* erweitert, um die Synchronisation der Tasks und ISR mit dem Kontext ohne Änderungen an eventuell vorhanden Verhaltensbeschreibungen in SystemC vorzunehmen. Zur Modellierung der CPU Zeit existiert die Memberfunktion *CONSUME_CPU_TIME*.
Die Deklaration von Modulklassen für den RTOS-Kontext erfolgt mit den Macros *SC_RTOS_MODULE* und *SC_RTOS_CTOR* analog zu den üblichen SystemC-Makros. Dies sei durch das folgende Beispiel erläutert, in dem eine Funktion *run* als *SC_THREAD* deklariert wird, innerhalb der dann eine Software-Task implementiert werden kann:

```
SC_RTOS_MODULE(Task)
{
public:
    SC_RTOS_CTOR(Task)
    {
        SC_THREAD(run);
    }
```

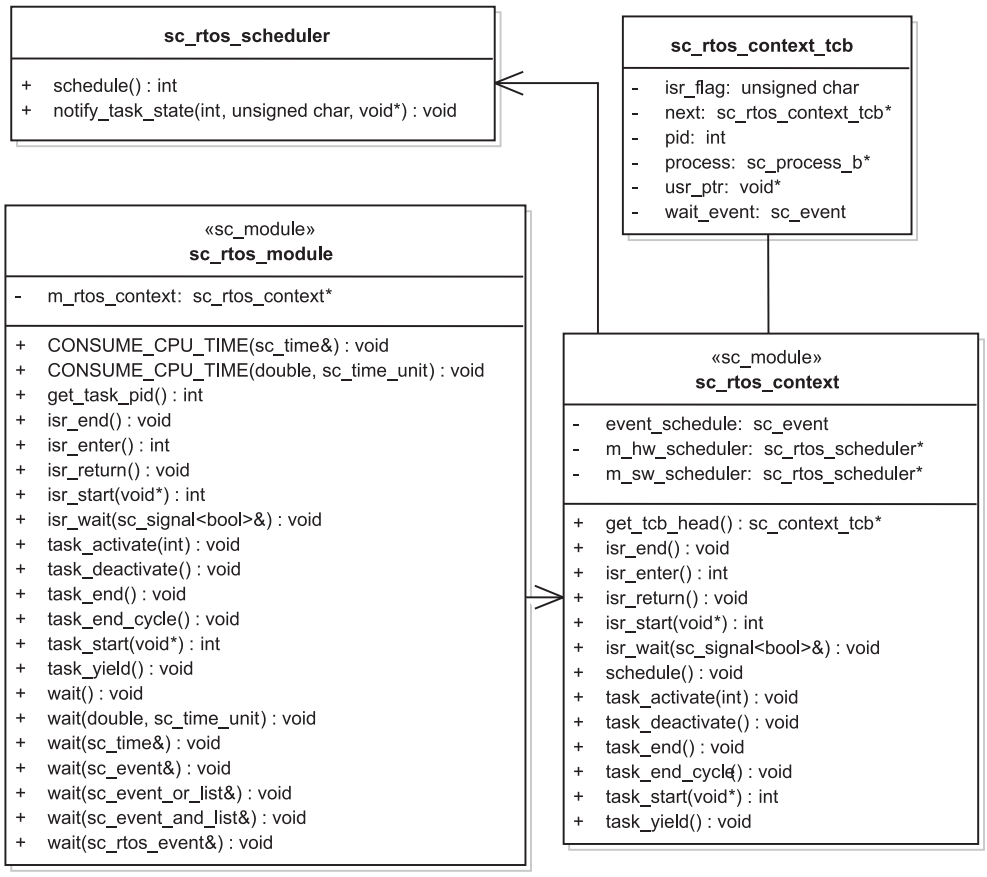

Abbildung 2: Klassendiagramm der RTOS SystemC Erweiterung

```
    // my methods ..
};
```

Die Implementierung dieser Task kann dann wie folgt aussehen:

```
void run ()
{
    task_start (<ptr>);
    while (true)  // endless loop
    {
        // do work
        // ...
        CONSUME_CPU_TIME (1,SC_MS);

        // wait
        // ...
        wait (<event>);
```

```
        ....
    }
    task_end ();
}
```

Einzelne Scheduler werden von der Klasse *sc_rtos_scheduler* abgeleitet. Letztere definiert eine Callback-Funktion, um Zustandsänderungen der Tasks an den entsprechenden Scheduler weiterzuleiten. Das Erstellen und Beenden von Tasks wird expliziet beschrieben und während der Simulation an die Scheduler gemeldet, so dass innerhalb der Scheduler-Klasse eine eigene Prozessverwaltung aufgebaut werden kann. Dem ISR-Scheduler werden dann nur die SystemC-Threads übermittelt, die eine ISR implementieren, d.h. bei denen das *isr_flag* im Task-Kontroll-Block (TCB für Task Control Block) gesetzt ist. Der Software-Scheduler bekommt Nachricht von allen am Kontext angemeldeten SystemC-Threads.

Die Klasse *sc_rtos_context_tcb* speichert einen abstrakten TCB, der folgende Informationen zu jeder Task enthält:

- **pid**: Interner Bezeichner (Process ID) der Task. Er wird fortlaufend von dem entsprechenden Kontext vergeben.
- **isr_flag**: Das Flag gibt an, ob es sich bei der Task um eine Interrupt-Service-Routine handelt oder nicht. Alle Interrupt-Service-Routinen werden immer bevorzugt behandelt. Erst wenn es keine lauffähige ISR gibt, werden die Tasks gescheduled.
- **process**: Die Variable enthält einen Zeiger auf das Task-Objekt des zugehörigen *SC_THREAD* und dient als eindeutige Identifizierung der Thread-Instanzen.
- **state**: Die Variable enthält den aktuellen Zustand der Task. Die Zustände werden über die API der Kontextklasse gesetzt und entsprechen möglichen Taskzuständen, wie sie auch bei anderen Betriebssystemen Verwendung finden. Abbildung 3 zeigt den Zyklus der Zustände.
- **user_ptr**: Die Variable enthält einen Zeiger auf beliebige Daten, die zu einer Task oder ISR gespeichert werden sollen. Dahinter verbirgt sich die Idee, dass der Anwender innerhalb der Implementierung der Scheduler jeweils eigene Tasklisten anlegt, auf Basis derer eine Scheduling-Entscheidung getroffen wird. Dazu müssen z.B. Daten wie Prioritäten, Zeitschranken gespeichert werden. Diese Daten können mit Hilfe dieser Variable dem TCB zugeordnet werden.

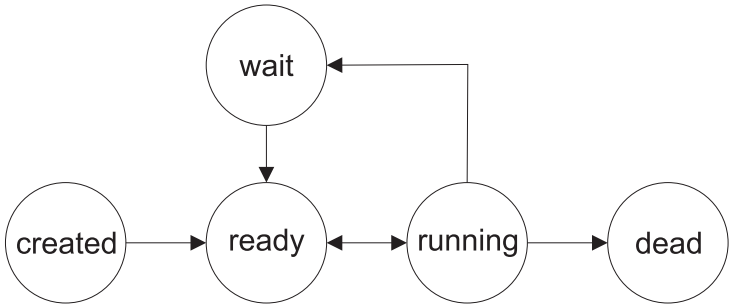

Abbildung 3: Zustandsübergänge innerhalb der TCB

- **wait_event**: Mit Hilfe dieses Events werden die Prozesse synchronisiert (s. 4.3).

4.3 Realisierung der RTOS Synchronisation

Die Aufgabe der vorgestellten Klassen liegt in einer Sequentialisierung der Ausführung von einer Menge von Tasks und Interrupt-Service-Routinen. Diese Sequentialisierung wird dadurch erreicht, dass alle Threads nach dem Warten auf ein SystemC-Event zunächst den Kontext über den Erhalt des Events informieren. Dann werden sie innerhalb des Kontextes blockiert, indem sie auf das Event *wait_event* in ihrem TCB warten müssen. Innerhalb des Kontext wird genau einer Task auf Basis des Rückgabewertes der Scheduler ein *notify* für dieses Event gesendet. Somit ist sichergestellt, dass immer nur eine Task Operationen ausführt.
Wie oben erwähnt, sind die *wait*-Funktionen in der Klasse *sc_rtos_module* überschrieben. Eine konkrete Implementierung für ein SystemC-Event sieht wie folgt aus:

```
void sc_rtos_module::wait (sc_event &e)
{
    int pid = m_rtos_context->task_wait ();
    sc_module::wait (e);
    m_rtos_context->task_activate (pid);
}
```

Die Funktion teilt dem Kontext zunächst mit, dass der Prozess in den Zustand *WAIT* versetzt werden soll. Innerhalb der Funktion *task_wait* wird für den nächsten Δ-Zyklus ein Rescheduling getriggert. Die Funktion kehrt zurück und der eigentliche Taskwechsel findet innerhalb der ursprünglichen *wait*-Anweisung durch den SystemC-Scheduler statt. Nach Erhalt des Events wird der Kontext durch Aufruf von *task_activate* über den Empfang des Signales informiert, d.h. darüber, dass der Prozess weiterarbeiten kann. In

dieser Funktion findet das erwähnte Warten auf *wait_event* statt, d.h. die Funktion kehrt erst nach Aktivierung durch das Scheduling zurück.
Zur Simulation von CPU-Zeit wird die folgende Methode verwendet:

```
void sc_rtos_context::CONSUME_CPU_TIME (sc_time &d)
{
    sc_event e_wakeup;
    sc_time  wait_time (d);
    sc_time  wait_start;

    // time to wait ?
    while (wait_time > SC_ZERO_TIME)
    {
        // notice current time
        // and start timer
        wait_start = sc_time_stamp ();
        e_wakeup.notify (wait_time);

        // wait on timer or reschedule
        wait (e_wakeup | m_event_activity_change);
        e_wakeup.cancel ();

        // evaluate real waiting time
        sc_time now = sc_time_stamp ();
        wait_time -= now - wait_start;

        // release processor
        task_yield ();
    }
}
```

Diese Funktion simuliert das Warten, indem das Event *e_wakeup* für den aktuellen Zeitpunkt plus der Wartezeit gesendet und anschließend darauf gewartet wird. Zusätzlich wird auf das externe Event *m_event_activity_change* gewartet, das eine Veränderung am Schedule (d.h. einen folgenden Reschedule) signalisiert. Dieses Event wird immer einen Simulationszyklus vor dem Ausführen des Schedulers gesendet. Das Warten wird bei Erhalt des Events unterbrochen und die bisherige Wartezeitspanne von der Wartezeit subtrahiert. Anschliessend wird durch Aufruf von *task_yield* die Aktivität an den RTOS-Kontext zurückgegeben. Bei Reaktivierung wird die Restzeit entsprechend noch durch weitere Iterationen simulierten. Dieser Mechanismus implementiert somit ein unterbrechbares Warten, was analog zu der Unterbrechung der Ausführung von Instruktionen anzusehen ist. Der Zeitpunkt für Simulationen von Interrupts kann damit sehr genau nachgebildet werden. Ein weiterer Vorteil ist, dass die Genauigkeit der Zeitpunkte nicht von der Größe der Segmente des Quelltextes abhängt.

4.4 Modellierung von RTOS-Events

Das Überladen der SystemC-*wait*-Anweisungen ist eine Möglichkeit zur Synchronisation mit dem RTOS-Modell bzw. dem Kontext. Der große Vorteil liegt darin, dass der Kommunikationsmechanismus über SystemC-Events ohne Änderungen übernommen werden kann. Das ist insbesondere hilfreich, wenn bestehende SystemC-Implementierungen mit Hilfe des RTOS-Modells sequentialisiert werden sollen.
Events lassen sich allerdings auch mit Hilfe der Funktionen *task_deactivate* und *task_activate* realisieren. Dies ist in der Klasse *sc_rtos_event* implementiert. Sie beinhaltet eine Liste mit Tasks, die auf das Event warten. Durch Aufruf der Member-Methode *wait* wird die aufrufende Task in der Liste eingetragen und mittels *task_deactive* unterbrochen. Innerhalb von *task_deactive* wartet sie auf das Event *wait_event* ihres TCB. Der Aufruf von *notify* in der Klasse *sc_rtos_event* setzt alle wartenden Tasks wieder in den Zustand *READY*.
Die Implementierung ist vergleichbar mit Events in konkreten RTOS-Implementierungen.

4.5 Scheduling von Tasks und ISR

In der Kontextklasse *sc_rtos_context* wird die SystemC-Methode *schedule* implementiert, die sensitiv auf das Event *event_schedule* reagiert. Wann immer eine Task eine Zustandsänderung an den Kontext meldet, wird dieses Event für den folgenden Simulationsschritt ausgelöst. Gleichzeitig wird für den aktuellen Schritt das Event *m_event_activity_change* gesendet, um sicherzustellen, dass ein aktivierter Task seine Aktivität zunächst an den Kontext zurückgibt. Die Funktion *schedule* sucht zunächst nach lauffähigen ISR, die sich im Zustand *READY* befinden. Findet sich mindestens eine solche ISR, so wird der ISR-Scheduler aufgerufen und die ausgewählte ISR aktiviert. Danach beendet sich die Funktion *schedule*.
Wenn es keine lauffähige ISR gibt oder der ISR-Scheduler keine ID für eine ISR zurückgegeben hat, wird der Task-Scheduler aufgerufen. Task- und ISR-Scheduler werden jeweils von der Klasse *sc_rtos_scheduler* abgeleitet.
Immer wenn eine Task oder ISR den Zustand ändert, werden die Scheduler darüber durch Aufruf von *notify_task_state* informiert. Die Scheduler müssen daraufhin ihren internen Zustand, d.h. wenn vorhanden ihre lokale Prozesstabelle, anpassen. Die jeweils für Task- und ISR-Scheduler überschriebenen Funktionen *schedule* der Klasse *sc_rtos_scheduler* berechnen das eigentliche Scheduling und liefern eine ID der nächsten lauffähigen Task bzw. ISR zurück. Die Scheduler haben hier nur über ihre Rückgabewerte die Möglichkeit auf

das Scheduling Einfluss zu nehmen.
Das Sperren von Interrupts, NMI (non maskable interupts) und Nested-Interrupts muss durch den ISR-Scheduler realisiert werden. Z.B. kann das Sperren von Interrupts dadurch realisiert werden, dass die Funktion *scheduler* keine ID liefert, bzw. wenn bereits eine ISR läuft, dieselbe ID zurückliefert, selbst wenn höher priorisierte Interrupts vorliegen.

4.6 Funktionen des Kontextes

Im folgenden Abschnitt wird die API des Kontextes im Detail vorgestellt. Mit ihr lassen sich die beschriebenen Prozesswechsel und Zustandsübergänge in einem SystemC-Thread annotieren, so dass der Thread entweder als Task oder als Interrupt-Service-Routine mit in die Simulation eines RTOS-Kontextes aufgenommen werden kann.

4.6.1 Erzeugen und Beenden von Tasks

Tasks und ISRs werden nicht direkt über die Kontext-API erstellt. Die API dient vielmehr dazu einen bereits bestehenden SystemC-Thread als Task oder Interrupt-Service-Routine am Kontext anzumelden und ihn damit in das sequentialisierte Scheduling aufzunehmen. Hierzu stehen folgende Funktionen zur Verfügung:

- **task_start:** Diese Funktion markiert das Starten einer Task. Innerhalb des Kontextes wird ein neuer TCB erstellt. Der Zustand der Task wird auf *CREATED* gesetzt und an den Scheduler gemeldet. Direkt darauf wird der Zustand auf *READY* gesetzt und ebenfalls gemeldet. Die Funktion blockiert an dieser Stelle durch Warten auf das *wait_event* des zugehörigen TCB. Ab diesem Zeitpunkt unterliegt der SystemC-Thread dem Task-Scheduler des Kontextes.

  Der Funktion wird ein Zeiger (*user_ptr*) auf Schedulingdaten übergeben. Diese Daten können Schedulinginformationen für die Task enthalten und werden an den Scheduler weitergereicht.

- **task_end:** Diese Funktion markiert das Ende einer Task. Entsprechend wird der TCB aus dem Kontext gelöscht und die Task wird nicht mehr im Kontext bearbeitet.

- **isr_start:** Diese Funktion markiert den Start einer Interrupt-Service-Routine. Innerhalb des Kontextes wird ein neuer TCB erstellt. Der Zustand der Task wird auf *CREATED* gesetzt und an den ISR-Scheduler gemeldet. Im Gegensatz zu Tasks wechselt der Zustand allerdings nicht

auf *READY* und die Funktion blockiert nicht den SystemC-Thread. ISR blockieren in der Simulation nur beim Warten auf den Interrupt oder innerhalb der Simulation der Ausführungszeit während der eigentlichen Interruptbehandlung.

Der Funktion wird ein Zeiger (*user_ptr*) auf Schedulingdaten übergeben. Diese Daten können Schedulinginformationen für die ISR enthalten und werden an den ISR-Scheduler weitergereicht. Scheduling-Information bei einer ISR kann z.B. eine eindeutige ID der ISR sein.

- **isr_end:** Diese Funktion markiert das Ende einer Interrupt-Service-Routine, d.h. der TCB des SystemC-Threads wird aus dem Kontext gelöscht.

4.6.2 Kontextwechsel

Die wichtigsten Funktionen zur sequentiellen Simulation von Tasks sind die zur Modellierung der Kontextwechsel. Mit Kontextwechsel ist hier die Übergabe der Aktivität von einer RTOS-Task oder einer Interrupt-Service-Routine an den Scheduler und zurück gemeint. Da der eigentliche Kontextwechsel im SystemC-Kernel nur in den *wait*-Anweisungen erfolgen kann, muss das RTOS-Modell (d.h. der Scheduler) über den Aufruf und das Verlassen einer *wait*-Anweisung in einer Task informiert werden.
Der Zustand der Tasks bzw. ISRs wird in den zugehörigen TCBs im Kontext gespeichert. Mögliche Zustände sind *CREATED*, *READY*, *RUNNING*, *WAIT* und *DEAD*. Die folgenden Funktionen implementieren die Zustandsübergänge entsprechend der Darstellung aus Abbildung 3.

- **task_wait:** Diese Funktion wird vor einer (SystemC) *wait*-Anweisung aufgerufen. Innerhalb der Funktion wird der Zustand der Task auf *WAIT* gesetzt und ein Aufruf des Schedulers wird für den nächsten Δ-Zyklus getriggert. Die Funktion blockiert nicht, da der eigentliche Kontextwechsel (im SystemC-Kernel) erst im nachfolgenden Aufruf von *wait* stattfindet (siehe auch 4.3). Die Funktion ist wie folgt realisiert:

```
int task_wait ()
{
    int task = get_tcb ();

    // set task to sleep
    set_task_state (task, WAIT);

    // enforce rescheduling
    event_schedule.notify (SC_ZERO_TIME);
```

```
    return task->pid;
}
```

- **task_deactivate:** Diese Funktion dient zum deaktivieren einer Task. Wie auch bei *task_wait* wird der Zustand auf *WAIT* gesetzt und der Scheduler für den nächsten Δ-Zyklus getriggert. Die Funktion wartet anschließend auf das *wait_event* ihres TCBs. Nach Aufwecken durch den Scheduler wird der Zustand der Tasks auf *RUNNING* gesetzt. Die Funktion ist wie folgt implementiert:

```
int task_deactivate ()
{
    task = get_tcb ();

    // set task to sleep
    set_task_state (task, WAIT);

    // suspend
    event_schedule.notify (SC_ZERO_TIME);
    wait (task->wait_event);

    // Mark Task as running (again)
    set_task_state (task, RUNNING);
}
```

  Die Funktion findet ihre Anwendung in der Implementierung von Events und Semaphoren, die direkt auf der API des RTOS-Modells realisiert sind (siehe 4.4).

- **task_activate:** Diese Funktion setzt den Zustand der übergebenen Task auf *READY*. Sie besitzt zwei unterschiedliche Ausprägungen. Bei der ersten reaktiviert die aufrufende Task die über den Parameter angegebene Task. Hier genügt das Wechseln des Zustandes. Die zweite Variante dient zum Warten auf die Reaktivierung durch den Scheduler. Diese Variante findet direkt nach einer SystemC-*wait*-Anweisung ihre Verwendung. Der SystemC-Thread ist beim Verlassen von *wait* zwar lauffähig, doch die Zuteilung des Prozessors zur Task wird durch das RTOS-Modell bestimmt. Aus diesem Grund muss auf die Benachrichtigung des Scheduler gewartet werden. Implementiert ist dies (wie auch oben) durch Anstoßen des Schedulers für den nächsten Δ-Zyklus und Warten auf *wait_event*. Anschließend wird der Zustand auf *RUNNING* gesetzt.

```
void task_activate (int pid)
{
```

```
        task = get_tcb ();

        // "wake up myself" ?
        if ( task->pid == pid )
        {
            // set state back to ready
            set_task_state (task, READY);

            // suspend and wait on reactivation
            event_schedule.notify (SC_ZERO_TIME);
            wait (task->wait_event);

            // set state to ready
            set_task_state (task, RUNNING);

        } else {

            // Wake up another thread
            task = get_tcb (pid);

            // Set task state to ready
            set_task_state (task, READY);

            // implicit rescheduling,
            // if calling tasks gives away it's activity
        }
    }
```

4.7 Periodische Tasks

Das Modell erlaubt ebenfalls die Darstellung periodischer Task, die in vielen Implementierungen von eingebetteten Systemen zu finden sind (z.B. Regler). Die allgemeine Implementierung von Tasks in unserer RTOS-Bibliothek wurde bereits am Anfang des Kapitels vorgestellt. Periodische Tasks stellen allerdings eine Besonderheit dar. Sie haben bei der Ausführung eher den Charakter einer Funktion, die regelmäßig aufgerufen und komplett bearbeitet wird. Um sie denoch als eigenständigen Task darstellen zu können, werden sie innerhalb einer Schleife wie folgt implementiert:

```
void run ()
{
    task_start (<ptr>);
    while (true) // endless loop
    {
        // wait on trigger
        wait ( <trigger> );

        // here follows periodic work
```

```
        // consume time
        CONSUME_CPU_TIME (<X>,SC_MS);

        // notify about end of cycle
        task_end_cycle ();
    }
    task_end ();
}
```

Innerhalb der Schleife wird auf eine Aktivierung durch einen Timer oder ein Event gewartet und anschließend die eigentliche periodische Aufgabe ausgeführt. Um das Ende des Zyklus zu bestimmen (wichtig für den Scheduler) wird dieses durch den Aufruf der Funktion *task_end_cycle* explizit markiert. Der Aufruf dient zur Erzeugung einer Benachrichtigung an den Scheduler. Der Scheduler muss dann allerdings entsprechend für Schedules von periodisichen Tasks ausgelegt sein.

4.8 Interrupt-Behandlung

Neben den Funktionen zur Markierung von Start und Ende zur Modellierung einer Interrupt-Service-Routine als SystemC-Thread, gibt es drei weitere Funktionen:

- **isr_wait:** IRQs werden als *signal<bool>* modelliert. Wenn das Signal *true* ist, liegt ein Interrupt-Request vor, und entsprechend bei *false* nicht. Auch die Interrupt-Service-Routinen müssen in der Simulation auf die Zuteilung des Prozessors durch das RTOS-Modell warten. Die Zeit zwischen Auftreten des Interrupt-Request und dem Start der Ausführung der Interrupt-Service-Routine wird im Allgemeinen als Interrupt-Latenz bezeichnet.

  Die Simulation sieht vor, dass eine höher priorisierte ISR die Ausführung einer nieder priorisierten ISR verzögern kann. Weiterhin kann in dieser Zeit ein IRQ zurückgesetzt werden. (z.B. durch Setzen von E/A-Registern). In diesem Fall wird die zugehörige ISR nicht ausgeführt. In der Simulation bzw. der Implementierung findet sich dieses Verhalten darin wieder, dass neben dem Warten auf *wait_event*, d.h. der Aktivierung durch den Scheduler, ebenfalls auf die fallende Flanke des Interrupt-Request gewartet wird. Sollte der Request vor der Aktivierung erlöschen, so wird die ISR wieder in den Zustand *WAIT* gesetzt und erneut auf eine steigende Flanke des Interrupt Signals gewartet.

  ```
  void sc_rtos_context::isr_wait (sc_signal<bool> &irq)
  {
  ```

```
    task = get_tcb ();

    for (;;)
    {
        // set task to sleep
        set_task_state (task, WAIT);

        // reschedule
        event_schedule.notify (SC_ZERO_TIME);

        // wait on high signal
        if (irq.read () == false)
        {
            wait (irq.posedge_event ());
        }

        // set task to aktive state
        set_task_state (task, READY);

        // fire scheduler
        event_schedule.notify (SC_ZERO_TIME);

        // wait on scheduler or irq going low
        wait ( (task->wait_event)
                    | irq.negedge_event ());

        // irq remains high ?
        if (signal.read () == true)
            break;

        // irq has been canceled

    } // for

    // Task is running
    set_task_state (task, RUNNING);
}
```

Die Zustandsübergänge in der obigen Funktion entsprechen, mit Ausnahme der Schleife, denen bei einem SystemC-*wait* geschachtelt mit *task_wait* und *task_activate*. Die ISR wird zunächst in den Zustand *WAIT* gesetzt und es wird auf den Interrupt-Request gewartet. Bei Empfang des Request wird der Zustand der ISR auf *READY* gesetzt und auf die Aktivierung durch den Scheduler gewartet. Beim Verlassen der Funktion *isr_wait* wechselt der Zustand der ISR auf *RUNNING*. Ausnahme bildet hier die oben beschriebene Sonderbehandlung, wenn der Interrupt-Request während des Wartens auf die Aktivierung verschwindet.

- **isr_enter:** Diese Funktion markiert den Anfang der eigentlichen Inter-

rupt-Service-Routine und dient zur Benachrichtigung der Scheduler. Innerhalb des ISR-Scheduler kann dann z.B. das Schedulen anderer Interrupts unterbunden werden.

- **isr_leave:** Die Funktion markiert das Verlassen einer Interrupt-Service-Routine. Da auch Interrupt-Service-Routinen Aufrufe von *CONSUME_CPU_TIME* zur Modellierung der Zeit enthalten können bzw. sollten, geben sie an diesen Stellen in der Simulation auch ihre Aktivität an den Scheduler ab und können von höher priorisierten Interrupt-Service-Routinen unterbrochen werden. Das Ende der Interrupt-Behandlung kann daher nicht aus dem Zustand *WAIT* oder *RUNNING* gefolgert werden und muss durch den Aufruf von *isr_leave* explizit modelliert werden.

Die Modellierung von ISR ähnelt denen der von periodischen Tasks. Sie erfolgt durch die Verwendung der *isr_...*-Methoden. Zusätzliche Callbacks für das Eintreten und Verlassen der eigentlichen Interrupt-Behandlung erleichtern die Implementierung von komplexen ISR-Schedulern. Eine typische ISR-Deklaration ist im Folgenden zu sehen:

```
void run ()
{
    isr_start (<ptr>);
    while (true)
    {
        // wait on irq request
        isr_wait (<event>);

        // notify entering ISR
        isr_enter ();

        // do work and consume time
        CONSUME_CPU_TIME (..);

        // notify return from interrupt
        isr_return ();
    }
    isr_end ();
}
```

5 Anwendungen

Die Simulationen mit abstrakten RTOS-Modellen in SystemC oder SpecC haben sich in den Evaluierungen der unter Kapitel 3 vorgestellten Arbeiten

als sehr effizient in Bezug auf die Simulationsgeschwindigkeit und als geeignete Methode zur frühen Evaluierung von Schedulingverfahren erwiesen. Der Mehraufwand bei Verwendung eines RTOS-Modelles gegenüber einem unschedulten Modell wird bei Gerstlauer et al. [GYG03] und Posadas et al. [PAV+05] mit weniger als 6% angegeben. Da die Verwendung eines RTOS-Modelles die Genauigkeit der Simulation verbessert, ist dieser Mehraufwand gerechtfertigt. Für effektive Beschleunigung der Simulation gegenüber einer ISS wird von Gerstlauer et al. ein Faktor frößer als 800 angegeben.
In mehreren Anwendungen, die im Folgenden beschrieben werden, konnten wir diese Zahlen bestätigen und teilweise übertreffen.

In [ZM08] haben wir die Evaluierung unserem RTOS-Modell am Beispiel der Software zur Motoransteuerung eines mobilen Roboters mit Differentialantrieb beschrieben. Die Steuerung wird durch einen Atmel Mikrocontroller AT90CAN128 mit 16Mhz ausgeführt. Sie hat die folgende Aufgaben: (1) Auswertung der Radencoder zur Messung der Umdrehungsgeschwindigkeit und Drehrichtung, (2) Motorregelung für die Antriebsmotoren, (3) Messung von Motorströmen und Versorgungsspannung und (4) Kommunikation über UART zum Datenaustausch. Die Software verfügt über 5 ISR und 11 Tasks. Die Tasks werden jeweils über Ereignisse aus den ISR oder anderen Tasks selbst aktiviert. Wir konnten durch unsere Simulation validieren, dass (1) die Regelung innerhalb des Intervalls von 250Hz berechnet wird und (2) die ISR für die Radencoder ausreichend schnell bearbeitet werden, um bei einer Radumdrehung von bis zu 300U/min den Wert für die Umdrehung und die Drehrichtung zuverlässig zu ermitteln.

In weiteren Untersuchungen zur Simulationsgeschwindigkeit von zeitannotierten Simulationsmodellen, haben wir verschiedene Funktionen auf dem AT90CAN128 implementiert und die Laufzeit der Simulation mit dem AVR-Studio von ATMEL gegen das zeitannotierte Modell verglichen. Die Messungen zeigten eine Steigerung der Simulationgeschwindigkeit um einen Faktor bis zu 4000. Die Abweichung gegenüber einer ISS lag bei wenigen Instruktionen. Der AT90CAN128 verfügt über eine 2-stufige Pipeline und hat keinen Cache. Bei größeren Prozessoren können die Abweichung bedingt durch Cache oder längere Pipelines größer ausfallen.

In einer ersten Implementierung unseres RTOS-Modells wurde das unterbrechbare Warten nicht implementiert. Die Implementierung der Funktion *CONSUME_CPU_TIME* war somit ein direkter Aufrufe der SystemC-*wait*-Funktion. Die Genauigkeit bei der Simulation von IRQs und Taskwechsel ist damit allerdings auf die Zeitspanne der größten annotierten Ausführungszeit beschränkt. Ein kleines Beispiel mit 3 Tasks und einer ISR macht dies deut-

lich: Eine Task aktiviert alle 20ms die zwei andere Tasks, die jeweils eine Aufgabe mit annotierter Dauer von 60ms bzw. 100ms ausführen. Alle 20ms wird ein IRQ ausgelöst, auf den die ISR reagiert. Die Auführung der ISR ist mit 4ms annotiert. Wir haben die Simulation mit verschiedenen Genauigkeiten durchgeführt, indem die maximale annotierte Länge eines Intervalls auf maximalen Wert Δt_{max} begrenzt wurde. Längere Intervalle wurden entsprechend in einer Schleife aufgeteilt. Die folgende Tabelle zeigt die Simulationsdauer, minimale, maximale und durchschnittliche IRQ-Latenz und die Anzahl simulierte IRQ für verschiedene Werte von Δt_{max}. Wir nehmen an, dass Kontextwechsel selbst keine Zeit benötigen. Bei einer Simulationzeit von 5h werden 900.000 IRQs ausgelöst. In der Simulation müssen daher 899.999 Ausführungen der ISR gemessen werden. Eine Ausführung fehlt, da der IRQ genau am Ende des Simulationsintervalles liegt.

		IRQ Latenz			
Δt_{max}	$t_{simulation}$	Min	Max	Durchschnitt	#ISR
100ms	2.1s	0.0ms	99ms	44.0ms	307893
50ms	2.9s	0.0ms	44ms	21.3ms	510000
25ms	3.7s	0.0ms	22ms	10.3ms	707142
18ms	4.5s	0.0ms	17ms	7.2ms	899999
10ms	5.1s	0.0ms	10ms	4.9ms	899999
5ms	6.3s	0.0ms	5ms	2.7ms	899999
1ms	17.1s	0.0ms	1ms	0.9ms	899999
0.5ms	30.1s	0.0ms	0.5ms	0.5ms	899999
$1/\infty$	6.1s	0ms	0ms	0ms	899999

Erwartungsgemäß zeigt sich, dass erst ab einer maximalen Intervallgröße von $\Delta t_{max} < 20ms$ alle ISR-Ausführungen simuliert werden. Bei größeren Werten treten zwei oder mehrere IRQs in einem Intervall auf, wobei allerdings nur der letzte IRQ einen Aufruf der ISR bewirkt. Würde z.B. dieser IRQ einen leeren Ausgangspuffer für eine UART-Schnittstelle signalisieren, führt die mangelnde Genauigkeit zu der Simulation der niedrigeren Übertragungsrate. Für $\Delta t_{max} < 20ms$ werden zwar alle ISR simuliert, allerdings mit einer von Null verschiedenen Latenz. Da wir für die Zeit zum Aufruf einer ISR 0 angenommen haben, stellen diese Werte ebenfalls einen Simulationsfehler da. Die maximale Latenz ist durch Δt_{max} nach oben begrenzt. Für kleinere Werte von Δt_{max} steigt allerdings die Simulationszeit stetig an. In der letzten Zeile ist die Messung mit Unterbrechbaren Warten dargestellt. Sowohl Latenz als auch Anzahl der ISR ist korrekt. Die Simulationszeit ist gegenüber dem Wert für $\Delta t_{max} = 18ms$ um 35% größer. Zur Simulation mehrerer ISR muß der Wert Δt_{max} entsprechend weiter reduziert werden. Neben der exakten Simulation, relativiert sich der Mehraufwand von 35% dadurch. Das

Beispielmodell beinhaltet darüberhinaus keine funktionale Simulation, d.h. die gemessen Simulationzeiten beinhalten im Wesentlichen die Laufzeiten zur Ausführung des RTOS-Modells. Wenn innerhalb der Tasks komplexe Algorithmen (z.B. Audio Encoder/Decoder) simuliert werden, relativieren sich die gemessenen Zeiten nocheinmal.

Das RTOS-Modell wird z. Zt. ferner im Rahmen einer Diplomarbeit in Kooperation mit einem Industriepartner am Beispiel der Steuerung einer Einspritzanlage untersucht. Ziel der Arbeit ist, die bisherigen Simulationen der Steuersoftware um die Möglichkeit zur Schedulinganalyse zu erweitern. Bei den bisherigen Simulationen wird davon ausgegangen, dass die ausgeführten Funktionen keine Rechenzeit benötigen (Null-Zeit Annahme). Aufrufe von Reglern zu Zeitpunkten entsprechend ihrer Abtastrate ermöglichen mit dieser Annahme funktionale Simulationen, allerdings lassen sich keine Einflüsse der Regler untereinander bedingt durch das zu Grunde liegende Scheduling des RTOS untersuchen. Ein Ergebnis der Arbeit ist die Integration unserer RTOS-Analyse in die Simulation, ohne dabei Änderungen (bis auf Zeitannotation) an der zu untersuchenden Software vornehmen zu müssen.

6 Danksagung

Die hier beschreibenen Arbeiten wurden im Rahmen des ITEA2-Projektes TIMMO durchgeführt. Sie wurden teilweise durch das Bundesministerium für Bildung und Forschung (BMBF) unter der Nummer 01IS07002 gefördert.

Vielen Dank an Andreas Gerstlauer von der Universität Irwine/California für viele interessante Anregungen und Korrekturen, die in diesen Bericht eingeflossen sind. Ebenso vielen Dank an Kay Klobedanz und Nitza Pampoukidou für die Korrektur des Berichtes.

Literatur

[AKS+04] H. M. AbdElSalam, S. Kobayashi, K. Sakanushi, Y. Takeuchi, and M. Imai. Towards a Higher level of Abstraction in Hardware/Software Co-Simulation. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 824–830, Washington, DC, USA, 2004. IEEE Computer Society.

[BB97] Giorgio C. Buttazzo and Giorgio Buttanzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[GYG03] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *Proceedings of Design, Automation and Test in Europe, March 2003.*, 2003.

[GZD+00] Daniel Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methology*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[HK07] Sorin A. Huss and Stephan Klaus. Assessment of Real-Time Operating Systems Characteristics in Embedded Systems Design by SystemC Models of RTOS Services. In *DVCon 07: Design and Verification Conference and Exhibitation*, San Jose, CA, 2007.

[HKH04] Prih Hastano, Stephan Klaus, and Sorin A. Huss. An Integrated SystemC Framework for Real-Time Scheduling Assessments on System Level. In *Proceedings of IEEE Int. Real-Time Systems Symposium*, 2004.

[HRR+06] Fabiano Hessel, Vitor M. Da Rosa, Carlos Eduardo Reif, Cäsar Marcon, and Tatiana Gadelha Serra Dos Santos. Scheduling refinement in abstract RTOS models. *Trans. on Embedded Computing Sys.*, 5(2):342–354, 2006.

[HSTI05] M. AbdElSalam Hassan, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 554–559, Washington, DC, USA, 2005. IEEE Computer Society.

[KBR06] M. Krause, O. Brinkmann, and W. Rosenstiel. A SystemC-based Software and Communication Refinement Framework for Distributed Embedded Systems, 2006.

[Kla05] S. Klaus. *System-Level-Entwurfsmethodik eingebetteter Systeme*. PhD thesis, Technische Universität Darmstadt, 2005.

[MPC04] R. Le Moigne, O. Pasquier, and J-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *DATE '04: Proceedings of the conference on Design, automation and*

test in Europe, page 30082, Washington, DC, USA, 2004. IEEE Computer Society.

[PAS+06] Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, and Francisco Blasco. POSIX Modeling in SystemC. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 485–490, New York, NY, USA, 2006. ACM Press.

[PAV+05] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*, 10(4):209–227, December 2005.

[PB00] Peter Puschner and Alan Burns. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[QPV+06] D. Quijano, H. Posadas, E. Villar, F. Escuder, and M. Martinez. TLM interrupt modeling for HW/SW co-simulation in systemc. *XXI Conference on Design of Circuits and Integrated Systems, DCIS'06*, 2006.

[RBMD03] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation, 2003.

[Sys08] 2008. Homepage of SystemC: http://www.systemc.org/.

[Yu05] Haobo Yu. *Software Synthesis for System-on-Chip*. PhD thesis, University of California, Irvine, 2005.

[ZM08] Henning Zabel and Wolfgang Müller. Präzises Interrupt Scheduling in abstrakten RTOS Modellen in SystemC. In *MBMV 08: Proceedings of the 11. Workshop ,,Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, March 2008.