Atomic Basic Blocks
Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur

Fabian Scheler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Lehrstuhl für Informatik 4 Verteilte Systeme und Betriebssysteme
fs@cs.fau.de

Abstract: Je nach Anwendungsfall ist das ereignis- oder das zeitgesteuerte Paradigma für die Realisierung eines Echtzeitsystems zu bevorzugen. Diese Paradigmen nehmen aber auch entscheidenden Einfluss auf die interne Struktur eines Echtzeitsystems, etwa die Koordinierung gleichzeitiger Ereignisbehandlungen bei Erzeuger-Verbraucher-Beziehungen oder kritischen Abschnitten, was eine spätere Anpassung des gewählten Paradigmas schwierig gestaltet. Um trotz dieser strukturellen Unterschiede eine werkzeuggestützte Anpassung des Echtzeitparadigmas zu ermöglichen, wurde im Rahmen dieser Dissertation die Abstraktion *Atomic Basic Block* und darauf basierend der *Real-Time Systems Compiler* entwickelt, ein Transformationswerkzeug, das einen automatisierten Übergang von ereignis- zu zeitgesteuerten Systemen ermöglicht.

1 Einleitung

Echtzeitrechensysteme sind oft in eine physikalische Umwelt eingebettet, die ihr Verhalten entscheidend prägt. Aus der Umwelt hervorgehende Stimuli initiieren Berechnungen im Echtzeitrechensystem, die sowohl funktional korrekt ablaufen müssen, als auch zeitlichen Beschränkungen unterworfen sind. Allein korrekte Ergebnisse zu liefern reicht also nicht aus, Berechnungsergebnisse müssen auch bis zu einem bestimmten Termin vorliegen. Wird ein Termin verpasst und das Berechnungsergebnis zu spät geliefert, kann dies ebenso ernsthafte Konsequenzen haben wie fehlerhaft berechnete Werte.

Die Abläufe innerhalb eines Echtzeitrechensystems, vom Eintreten des Stimulus bis hin zur Bereitstellung des Ergebnisses, unterscheiden sich je nach der verwendeten *Echtzeitsystemarchitektur*. Sie entscheidet, wie Ereignisbehandlungen aktiviert und wie Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen implementiert werden. Die bekanntesten Ausprägungen solcher Echtzeitsystemarchitekturen sind das *ereignisgesteuerte* und das *zeitgesteuerte Paradigma*. Ersteres koppelt Ereignisbehandlungen durch Unterbrechungen direkt an externe Ereignisse und implementiert Abhängigkeiten zwischen Ereignisbehandlungen explizit mithilfe von Synchronisationsmechanismen wie Semaphoren oder Schlossvariablen. Zeitgesteuerte Systeme hingegen ordnen Ereignisbehandlungen in einer

vorab berechneten Ablauftabelle an, die zyklisch abgearbeitet wird und Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen bereits berücksichtigt.

Je nach Anwendungsszenario bieten beide Paradigmen unterschiedliche Vorteile. So punkten ereignisgesteuerte Systeme durch Flexibilität, wenn sich die Ereigniszeitpunkte in der physikalischen Umwelt nicht exakt vorhersagen lassen, während zeitgesteuerte Systeme bei strikt periodischen Anwendungen ihre Stärken ausspielen können. Bei der Entwicklung des Space Shuttle entschied man sich beispielsweise für einen ereignisgesteuerten Ansatz, um flexibel auf sich ändernde Anforderungen in dem lang andauernden Entwicklungsvorhaben eingehen zu können [Car84]. Die Vorhersagbarkeit zeitgesteuerter Systeme hingegen kommt vor allem bei der Entwicklung und Verifikation fehlertoleranter Systeme zum Tragen, weshalb die Flugsteuerung des F-18-Kampfflugzeugs aufwändig von einer ereignis- auf eine zeitgesteuerte Architektur übertragen wurde [SG90].

Kann man die Wahl zwischen dem ereignis- und dem zeitgesteuerten Paradigma zu Beginn eines Entwicklungsvorhaben noch relativ frei treffen, ist eine spätere Migration mit hohem Aufwand verbunden. Die verwendete Echtzeitsystemarchitektur beeinflusst nämlich in großem Maße die Implementierung eines Echtzeitsystems. Ereignisgesteuerte Implementierungen sind häufig mit Aufrufen von Synchronisationsmechanismen durchsetzt, um Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen explizit herzustellen, während diese Abhängigkeiten in zeitgesteuerten Systemen kaum noch sichtbar sind. Eine Migration zwischen diesen beiden Welten erfordert also zunächst eine mühsame Aufarbeitung dieser Abhängigkeiten, um sie korrekt auf die Zielarchitektur zu übertragen.

Im Rahmen dieser Dissertation wurde nun eine Möglichkeit entwickelt, diese Migration mithilfe eines Transformationswerkzeugs zu automatisieren. Die Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen werden hierfür durch globale Abhängigkeitsgraphen unabhängig von der Echtzeitsystemarchitektur beschrieben. Diese Abhängigkeitsgraphen bestehen aus *Atomic Basic Blocks* (ABBs) und basieren auf interprozeduralen Kontrollflussgraphen, wie sie im Übersetzerbau verwendet werden. Im Gegensatz zu Kontrollflussgraphen können aber auch Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen ausgedrückt werden. Die zeitlichen Eigenschaften der physikalischen Umwelt, etwa die Periodizität der einzelnen Stimuli oder der Termin, bis zu dem eine Ereignisbehandlung abgeschlossen sein muss, werden in einem *Systemmodell* gekapselt und mit den durch ABBs beschriebenen Ereignisbehandlungen verknüpft. ABBs und das Systemmodell sind die Grundlage des im Rahmen dieser Dissertation entwickelten *Real-Time Systems Compilers* (RTSC), eines Quelltexttransformationswerkzeugs, um die Echtzeitsystemarchitektur eines Echtzeitsystems gezielt zu manipulieren.

Im folgenden Abschnitt 2 werden kurz die charakteristischen Merkmale einer Echtzeitsystemarchitektur dargelegt. Anschließend beschreibt Abschnitt 3 die Abstraktion *Atomic Basic Block*. Zusammen mit dem in Abschnitt 4 vorgestellten *Systemmodell* bildet sie die grundlegenden Abstraktion für den *Real-Time Systems Compiler*, der in Abschnitt 5 präsentiert wird. Abschnitt 6 fasst die wesentlichen Beiträge dieser Dissertation noch einmal zusammen und gibt Ausblick auf zukünftige Arbeiten.

```
Message *serialMsg;

ISR(SerialByte) {
  unsigned char rcv = getByte();
  msg_addTo(serialMsg,rcv);

  if(msg_complete(serialMsg)) {
    buffer_insert(buf,serialMsg);
    SetEvent(MsgHandler,MsgRcv);
  }
}
```

```
TASK(MsgHandler) {
  Message *currentMsg = 0;
  Initialisation();

  WaitEvent(MsgRcv);
  ClearEvent(MsgRcv);
  currentMsg = buffer_get(buf);
  handler(currentMsg);

  TerminateTask();
}
```

Abbildung 1: Byte-weiser Empfang einer Nachricht über die serielle Schnittstelle, implementiert durch die Unterbrechungsbehandlung **ISR**(SerialByte) und den Faden **TASK**(MsgHandler)

.

2 Echtzeitsystemarchitekturen

Eine Echtzeitsystemarchitektur stellt Mechanismen zur Verfügung, um Ereignisbehandlungen als Reaktion auf regelmäßig und unregelmäßig auftretende Stimuli (*periodische* und *nicht-periodische Ereignisse*) zu aktivieren und um gerichtete Abhängigkeiten (Sequentialisierung von Vorgänger und Nachfolger) und ungerichtete Abhängigkeiten (Koordinierung kritischer Abschnitte) zu implementieren. Außerdem bietet sie eine deterministische Ablaufplanung, um mehrere Ereignisbehandlungen abwechselnd auf demselben Prozessor auszuführen.

Anhand des Beispiels in Abbildung 1 soll ihr Einfluss auf die Struktur einer Echtzeitanwendung erläutert werden. Das Beispiel verwendet das Betriebssystem AUTOSAR OS [AUT09], das eine ereignisgesteuerte Echtzeitsystemarchitektur implementiert, und stellt den Empfang einer Nachricht über die serielle Schnittstelle Byte für Byte durch die Unterbrechungsbehandlung **ISR**(SerialByte) und den Faden **TASK**(MsgHandler) dar.

Die Unterbrechungsbehandlung **ISR**(SerialByte) wird durch das Eintreffen eines Bytes an der seriellen Schnittstelle aktiviert, wodurch sie auch direkt an diesen Stimulus gebunden wird. Die Unterbrechungsbehandlung holt das empfangene Byte von der seriellen Schnittstelle ab und fügt es zur aktuellen Nachricht hinzu. Vollständige Nachrichten werden anschließend in einem Puffer zwischengespeichert. Die gerichtete Abhängigkeit zwischen dem Erzeuger **ISR**(SerialByte) der Nachricht und deren Verbraucher **TASK**(MsgHandler) wird im vorliegenden Beispiel explizit durch die Systemaufrufe SetEvent und WaitEvent hergestellt. Sobald die explizite Wartebedingung dieser gerichteten Abhängigkeit erfüllt ist, kann **TASK**(MsgHandler) die Nachricht aus dem Puffer entnehmen und weiter verarbeiten. Zusätzlich existiert hier noch eine ungerichtete Abhängigkeit, die jedoch nicht abgebildet ist. Das Einfügen der neuen Nachricht und ihre Entnahme aus dem Puffer stellen kritische Abschnitte dar, die geeignet abgesichert werden müssen. Im Falle von AUTOSAR OS könnte dies beispielsweise durch Schlossvariablen geschehen, die eine überlappende Ausführung dieser kritischen Abschnitte verhindern.

Ereignisbehandlung	Startzeitpunkt
ISR(SerialByte)	50 μs
...	...
ISR(SerialByte)	100 μs
...	...
ISR(SerialByte)	450 μs
...	...
TASK(MsgHandler)	500 μs

Tabelle 1: Ausschnitt aus einer möglichen statischen Ablauftabelle für das Beispiel aus Abbildung 1.

Solche Abhängigkeiten werden in zeitgesteuerten Echtzeitsystemarchitekturen, wie sie etwa OSEKtime [OSE01] anbietet, implizit ohne Zuhilfenahme von Systemaufrufen wie `SetEvent` und `WaitEvent` umgesetzt. Hierfür werden die Ereignisbehandlungen in einer Ablauftabelle zeitlich so angeordnet, dass diese Abhängigkeiten gewahrt bleiben. Tabelle 1 skizziert auszugsweise eine solche Ablauftabelle. Sie platziert **ISR**(SerialByte) vor **TASK**(MsgHandler), um so implizit die gerichtete Abhängigkeit sicherzustellen und eine überlappungsfreie Ausführung der kritischen Abschnitte zu erreichen. Die konkreten Startzeitpunkte der einzelnen Ereignisbehandlungen in der Ablauftabelle ergeben sich dabei aus einer Analyse der physikalischen Umgebung. So können die Startzeitpunkte von **ISR**(SerialByte) beispielsweise aus der Datenrate der seriellen Schnittstelle abgeleitet werden. **TASK**(MsgHandler) wird im Vergleich zu **ISR**(SerialByte) wesentlich seltener aktiviert, weil eine komplette Nachricht meist aus mehreren Bytes besteht, die erst von der Unterbrechungsbehandlung geliefert werden müssen.

Dieses Beispiel demonstriert bereits anschaulich, wie unterschiedlich verschiedene Echtzeitsystemarchitekturen mit gerichteten und ungerichteten Abhängigkeiten umgehen, und welchen Einfluss dies auf die Implementierung einer Anwendung hat. Die Abhängigkeiten zwischen **ISR**(SerialByte) und **TASK**(MsgHandler) existieren auch in der zeitgesteuerten Umsetzung, nur sind sie nicht mehr explizit sichtbar, weil sie sich einzig in der Reihenfolge der Ablauftabelle niederschlagen, wohingegen sie in der ereignisgesteuerten Implementierung direkt an den entsprechenden Systemaufrufen abgelesen werden können. Eine generische und architekturneutrale Beschreibung dieser Abhängigkeitsbeziehungen ist also die Voraussetzung, um die Abbildung eines Echtzeitsystems auf verschiedene Echtzeitsystemarchitekturen zu ermöglichen. Dies leisten ABBs, die im folgenden Abschnitt vorgestellt werden.

3 Atomic Basic Blocks

ABBs beschreiben gerichtete und ungerichtete Abhängigkeiten zwischen verschiedenen Ereignisbehandlungen unabhängig von der verwendeten Echtzeitsystemarchitektur. Hierfür erweitern ABBs die aus dem Übersetzerbau bekannten *Grundblöcke*. Grundblöcke und auch interprozedurale Kontrollflussgraphen alleine reichen nämlich nicht aus, um auch kontrollflussübergreifende Abhängigkeiten zwischen verschiedenen Aktivitätsträgern dar-

zustellen. ABBs fassen daher mehrere aneinander hängende Grundblöcke zusammen und erzeugen so eine Vergröberung des lokalen Kontrollflussgraphen einer Funktion. Auf Funktionsebene zeichnen sie den Kontrollflussgraphen der Funktion nach und bilden so *lokale ABB-Graphen*. ABBs stellen aber auch Ansatzpunkte für kontrollflussübergreifende gerichtete und ungerichtete Abhängigkeiten dar. ABBs enden daher immer an *ABB-Endpunkten*, die genau diese Ansatzpunkte im Kontrollflussgraphen markieren. ABBs lassen sich mithilfe folgender Regeln aus einem Kontrollflussgraphen extrahieren:

1. ABBs umfassen einen oder mehrere Grundblöcke einer Funktion, die einen zusammenhängenden Teilgraphen des Kontrollflussgraphen dieser Funktion bilden.
2. Jeder ABB besitzt einen eindeutigen Grundblock $Entry(ABB)$, über den dieser ABB betreten wird, seinen *Eingang*. Ebenso existiert für jeden ABB höchstens ein Grundblock $Exit(ABB)$, über den der ABB verlassen wird, der *Ausgang*. Sie sind die einzigen Grundblöcke eines ABB, die Vorgänger bzw. Nachfolger im Kontrollflussgraphen der Funktion besitzen können, die nicht Bestandteil des ABB sind.
3. ABBs reichen vom Ende des vorhergehenden ABB bis zu einem ABB-Endpunkt.
4. Liegt ein ABB-Endpunkt innerhalb eines Grundblocks, wird dieser Grundblock entsprechend in zwei Grundblöcke geteilt.

ABB-Endpunkte sind *Quellen* oder *Ziele* kontrollflussübergreifender Abhängigkeiten oder *künstliche ABB-Endpunkte*. Letztere dienen lediglich dazu, die oben genannte Regel 2 zu erfüllen und eine eindeutige Zerlegung eines Kontrollflussgraphen in ABBs zu gewährleisten. Quellen und Ziele werden durch Systemaufrufe markiert, die Abhängigkeiten zwischen gleichzeitigen Kontrollflüssen herstellen, beispielsweise die Systemaufrufe `SetEvent` und `WaitEvent` des in Abschnitt 2 betrachteten Beispiels.

Gerichtete kontrollflussübergreifende Abhängigkeiten zwischen *vereinbaren ABB-Endpunkten* verknüpfen schließlich lokale ABB-Graphen einzelner Funktionen zu einem Wald *globaler ABB-Graphen*, die sämtliche Abhängigkeitsbeziehungen in einem Echtzeitsystem beschreiben. ABB-Endpunkte heißen vereinbar, wenn es sich bei ihnen um kompatible Quellen und Ziele handelt (z. B. die bereits genannten Systemaufrufe `SetEvent` und `WaitEvent`) und sie sich auf entsprechend zusammenhängende Systemobjekte beziehen (z. B. dasselbe Ereignis). Kritische Abschnitte werden ihrerseits durch eine Menge von zusammenhängenden ABBs eines lokalen ABB-Graphen beschrieben. Zwischen zwei kritischen Abschnitten besteht eine ungerichtete Abhängigkeit, wenn ihnen dasselbe Betriebsmittel zugeordnet wurde. Sie bilden also einen *ungerichteten globalen ABB-Graphen*.

Abbildung 2 zeigt den zu Abbildung 1 gehörenden globalen ABB-Graphen. Grundblöcke sind dabei als eckige Kästen und ABBs als Kästen mit abgerundeten Ecken dargestellt. Die gerichteten Kanten innerhalb der Ereignisbehandlungen **`ISR`**`(SerialByte)` und **`TASK`**`(MsgHandler)` spiegeln den ursprünglichen Kontrollflussgraphen wider. Wegen der zu den Systemaufrufen `SetEvent` und `WaitEvent` gehörenden ABB-Endpunkten wurden die Grundblöcke $BB2$ und $BB4$ jeweils in die Grundblöcke $BB2a$ und $BB2b$ beziehungsweise $BB4a$ und $BB4b$ aufgeteilt. Nachdem diese ABB-Endpunkte auch vereinbar sind, werden die beiden lokalen ABB-Graphen an dieser Stelle durch eine gerichtete

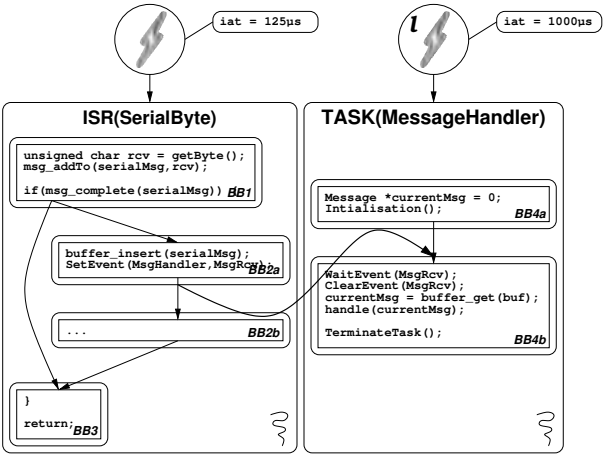

Abbildung 2: Globaler ABB-Graph des Beispiels aus Abbildung 1

Abhängigkeit zu einem globalen ABB-Graphen verknüpft. `SetEvent`, die Quelle der Abhängigkeit, wurde dem ersten Fragment $BB2a$ des ehemaligen Grundblocks $BB2$ zugeordnet, während das Ziel `WaitEvent` nun Bestandteil des zweiten Fragments $BB4b$ des früheren Grundblocks $BB4$ ist. Die übrigen ABBs in diesem Beispiel werden durch künstliche ABB-Endpunkte abgeschlossen. Auf die Darstellung der ungerichteten, in den Operationen `buffer_insert` und `buffer_get` enthaltenen Abhängigkeit wurde zugunsten der besseren Übersicht verzichtet.

4 Systemmodell

Neben der internen, durch ABB-Graphen beschrieben Struktur einer Echtzeitanwendung, müssen in Echtzeitsystemen zeitliche Rahmenbedingungen befolgt werden, die das kontrollierte physikalische Objekt bestimmt. Diese Informationen werden für die Manipulation der Echtzeitsystemarchitektur benötigt. Die hier angestrebte Migration ereignisgesteuerter Systeme auf zeitgesteuerte Echtzeitsystemarchitekturen verwendet dieses Wissen beispielsweise für die Berechnung von Ablauftabellen. Demzufolge werden diese Eigenschaften in einem *Systemmodell* notiert und dem im nachfolgenden Abschnitt beschriebenen Transformationswerkzeug zur Verfügung gestellt.

Diese Informationen umfassen *Ereignisse*, die angeben, wie häufig bestimmte *Aufgaben* und die damit verbundenen Ereignisbehandlungen im Echtzeitsystem ausgelöst werden, und *Termine*, die den spätesten möglichen Fertigstellungszeitpunkt einer Ereignisbehandlung markieren. Weiterhin unterscheidet man *periodische Ereignisse*, von denen *Periode*, *Phase* und *Jitter* bekannt sind, und *nicht-periodische Ereignisse*, von denen man nur die *minimale Zwischenankunftszeit* kennt, sowie *physikalische* und *logische Ereignisse*. Physikalische Ereignisse werden durch das kontrollierte Objekt erzeugt, während logische Ereignisse

Zustandsübergänge innerhalb der Anwendung kenntlich machen, die gesonderte Behandlung erfordern. Die Unterscheidung physikalischer und logischer Ereignisse erlaubt eine wesentlich präzisere Angabe der zeitlichen Eigenschaften des Echtzeitsystems. Während sich die zeitlichen Eigenschaften physikalischer Ereignisse alleine durch die Analyse des zu kontrollierenden Objekts erschließen, werden die eines logischen Ereignisses nämlich auch durch die Anwendung selbst bestimmt. Zusätzliches Anwendungswissen kann so in die zeitliche Beschreibung des Systems eingebracht werden. Die Verknüpfung zwischen globalen ABB-Graphen und dem Systemmodell erfolgt, indem Ereignisse und Termine immer einer entsprechenden Ereignisbehandlung zugeordnet werden, die wiederum durch einen ABB-Graphen dargestellt wird.

In Abbildung 2 sind zwei Ereignisse als in Kreise gefasste Blitze abgebildet. Ein physikalisches Ereignis aktiviert dabei die Aufgabe **`ISR`**`(SerialByte)` mit einer minimalen Zwischenankunftszeit von 125 μs, die durch die Übertragungsrate der seriellen Schnittstelle bestimmt wird. **`TASK`**`(MsgHandler)` hingegen wird von einem logischen Ereignis ausgelöst, dessen größere minimale Zwischenankunftszeit von 1000 μs auch von der Größe einer Nachricht abhängt. Könnte man dieses Anwendungswissen nicht durch ein logisches Ereignis nutzbar machen, müsste man für beide Aufgaben die kürzere Zwischenankunftszeit von 125 μs annehmen, was zu einer deutlich pessimistischeren Betrachtung führen würde.

5 Der Real-Time Systems Compiler

Der *Real-Time System Compiler* (RTSC) ist ein Quelltexttransformationswerkzeug, um die Echtzeitsystemarchitektur eines Echtzeitsystems gezielt zu beeinflussen. Der RTSC verarbeitet hierfür ein als Quelltext vorliegendes *Quellsystem* und das zugehörige Systemmodell, das die behandelten Ereignisse und Termine mit den Ereignisbehandlungen im Quellsystem verknüpft und in Form einer *Aufgabendatenbank* bereitgestellt wird. Eine weitere Eingabe des RTSC ist die Aufgabendatenbank des Zielsystems. Sie beschreibt, durch welche im Quellsystem implementierten Ereignisbehandlungen die Ereignisse des Zielsystems behandelt werden sollen und welchen zeitlichen Beschränkungen sie unterliegen. Hier erfordert beispielsweise der Übergang von nicht-periodischen auf entsprechende abfragende Ereignisse eine Anpassung ihrer zeitlichen Eigenschaften. Auf eine algorithmische Übertragung der Aufgabendatenbanken vom Quell- ins Zielsystem wurde aber bewusst verzichtet, um gezielt auf abweichende zeitliche Eigenschaften des Zielsystems eingehen zu können. Ausgabe des RTSC ist die Implementierung des Zielsystems, das die transformierten Ereignisbehandlungen des Quellsystems enthält.

Abbildung 3 stellt die an einen Übersetzer angelehnte Struktur des RTSC schematisch dar. Das Front-End und das Back-End hängen jeweils von der Echtzeitsystemarchitektur des Quell- beziehungsweise des Zielsystems ab. Die Implementierung des Middle-Ends hingegen ist architekturunabhängig, es wird aber durch das Zielsystem beeinflusst, was in der Abbildung durch entsprechende Schattierungen verdeutlicht wird. Die gerichteten, durchgezogenen Pfeile stehen für Ein- und Ausgabebeziehungen zwischen dem RTSC und dem Quell- beziehungsweise Zielsystem, die gepunkteten Pfeile zwischen den Implementierungen des Quell- und des Zielsystems und den Aufgabendatenbanken deuten die Verknüpfung der Ereignisse in der Aufgabendatenbank und den Ereignisbehandlungen

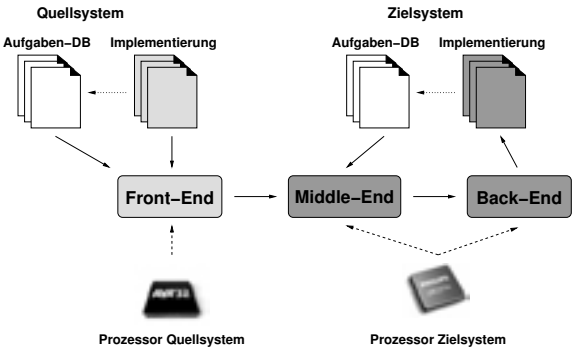

Abbildung 3: Grobgliederung des RTSC in Front-End, Middle-End und Back-End

in der Implementierung an. Die gestrichelten, von den Prozessoren des Quell- und des Zielsystems ausgehenden Pfeile kennzeichnen prozessorabhängige Schritte im RTSC. Die Prozessorabhängigkeit schlägt sich vor allem in der Analyse der maximalen Ausführungszeit nieder, die in verschiedene Analysen und Transformationsschritte eingebracht wird.

Grundsätzlich ist der RTSC als generisches Transformationswerkzeug zwischen verschiedenen Echtzeitsystemarchitekturen konzipiert, diese Dissertation konzentriert sich aber exemplarisch auf die Migration ereignisgesteuerter auf AUTOSAR OS basierender Echtzeitsysteme auf die zeitgesteuerte Architektur von OSEKtime. Ferner unterstützt der RTSC derzeit noch keine verschiedenen Prozessortypen im Quell- und Zielsystem, wie es in Abbildung 3 angedeutet ist, auch wenn dies prinzipiell möglich wäre.

5.1 Front-End

Das Front-End extrahiert einen globalen ABB-Graphen aus dem Quellsystem, der gerichtete und ungerichtete Abhängigkeiten innerhalb des Quellsystems unabhängig von dessen Echtzeitsystemarchitektur darstellt. Hierzu werden zunächst die Elemente des Quellsystems identifiziert, die Ereignisbehandlungen implementieren. Anschließend wird in der kompletten Implementierung nach ABB-Endpunkten gesucht, und es werden lokale ABB-Graphen erzeugt, die an den ABB-Endpunkten zu einem globalen ABB-Graphen verknüpft werden.

5.2 Middle-End

Im Middle-End wird der globale ABB-Graph mithilfe des statischen Ablaufplanungsalgorithmus von Adelzaher und Shin [AS99] in einer statischen Ablauftabelle zeitlich geordnet. Dazu werden zunächst Abhängigkeitsmuster aus dem ABB-Graphen entfernt, die der gewählte Algorithmus nicht direkt verarbeiten kann. Zu diesen Mustern zählen beispielsweise auch Funktionsaufrufe. Mehrfach aufgerufene Funktionen führen zu ABBs, die von mehreren Vorgängern aktiviert werden. Der Algorithmus geht aber davon aus, dass immer alle Vorgänger eingeplant werden müssen, bevor ein gemeinsamer Nachfolger ausführungsbereit wird. Bei mehrfachen Funktionsaufrufen wird der lokale ABB-Graph

der aufgerufenen Funktion beispielsweise entsprechend oft vervielfältigt, bis die Einstiegs-ABBs dieser Funktionen nur noch einen einzigen Vorgänger besitzen. Durch eine statische Analyse wird außerdem die maximale Ausführungszeit der einzelnen ABB bestimmt, ohne die eine statische Ablaufplanung nicht möglich wäre.[1]

Die hier angewandten Transformationen selbst sind unabhängig von der jeweiligen Echtzeitsystemarchitektur, es hängt aber von ihr ab, ob sie benötigt werden. In diesem Sinne wird das Middle-End zwar von der Echtzeitsystemarchitektur des Zielsystems beeinflusst, auf die Implementierung der verwendeten Transformationen wirkt sie sich aber nicht aus.

5.3 Back-End

Im letzten Schritt wird das aufbereitete Quellsystem auf die Elemente der Echtzeitsystemarchitektur des Zielsystems abgebildet. Dabei wird ein Anwendungsrumpf erstellt, der die transformierten Ereignisbehandlungen aktiviert, und die im Middle-End berechnete statische Ablauftabelle wird als Konfiguration für das OSEKtime-Betriebssystem ausgegeben. Weiterhin findet im Back-End auch die Übersetzung der transformierten Echtzeitanwendung in ein für den Prozessor des Zielsystems geeignetes Maschinenprogramm statt.

5.4 Implementierung

Als Grundlage für den RTSC wurde das LLVM-Projekt [LA04] ausgewählt, das ein Baukastensystem für die Entwicklung eigener Übersetzer anbietet und sämtliche Anforderungen des RTSC erfüllt. Die LLVM stellt eine Fülle von Analysen und Transformationen aus dem Bereich des Übersetzerbaus zur Verfügung und gestattet auf einfache Weise den Zugriff und die Manipulation der Kontrollflussgraphen der vorliegenden Echtzeitanwendung, was für die Erzeugung und die Transformation des ABB-Graphen notwendig ist. Die Abstraktion ABB und die zugehörigen ABB-Graphen konnten mithilfe der LLVM direkt als Aggregation einzelner, von der LLVM angebotener Grundblöcke implementiert und als zusätzliche Abstraktionsebene über den Kontrollflussgraphen gelegt werden.

6 Zusammenfassung und Ausblick

Die hier zusammengefasste Dissertation ebnet den Weg für eine gezielte Manipulation der Echtzeitsystemarchitektur eines Echtzeitsystems, um es so werkzeuggestützt auf seinen Einsatzzweck zuzuschneiden. Ermöglicht wird dies durch die Abstraktion *Atomic Basic Block*, die mit den Mitteln der Echtzeitsystemarchitektur hergestellte gerichtete und ungerichtete Abhängigkeiten architekturunabhängig darstellt. Ein *Systemmodell* kapselt die zeitlichen Eigenschaften der physikalischen Umwelt und verknüpft sie mit der durch ABB-Graphen beschriebenen Echtzeitanwendung. Darauf aufbauend wurde mit dem *Real-Time Systems Compiler* erstmalig ein Werkzeug geschaffen, um die Manipulation der Echtzeitsystemarchitektur automatisiert und ohne manuelle Eingriffe vorzunehmen.

[1] Die Analyse erfolgt derzeit mit Hilfe des Werkzeugs Absint aiT (`http://www.absint.de/ait`).

Im Rahmen des durch die DFG geförderten Projekts „Aspektorientierte Echtzeitsystemarchitekturen“ (AORTA) werden die Arbeiten an den Atomic Basic Blocks und dem RTSC fortgeführt. Dabei soll zum einen die Manipulation der Echtzeitsystemarchitektur mit der Umkehrung der hier entwickelten Migration, nämlich dem Übergang von zeit- auf ereignisgesteuerte Systeme vervollständigt werden. Zum anderen soll der Begriff „Echtzeitsystemarchitektur“ um verteilte Systeme sowie Mehrkernsysteme erweitert werden. Schließlich beeinflussen auch die Interaktionsmöglichkeiten über gemeinsamen Speicher oder ein dediziertes Kommunikationssystem die interne Struktur einer Echtzeitanwendung und stellen somit eine Facette der Echtzeitsystemarchitektur dar.

Literatur

[AS99] Tarek F. Abdelzaher und Kang G. Shin. Combined Task and Message Scheduling in Distributed Real-Time Systems. *IEEE TPDS*, 10(11):1179–1191, 1999.

[AUT09] AUTOSAR. Specification of Operating System (Version 4.0.0). Bericht, Automotive Open System Architecture GbR, Dezember 2009.

[Car84] Gene D. Carlow. Architecture of the space shuttle primary avionics software system. *CACM*, 27(9):926–936, 1984.

[LA04] Chris Lattner und Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[OSE01] OSEK/VDX Group. Time Triggered Operating System Specification 1.0. Bericht, OSEK/VDX Group, Juli 2001. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.

[SG90] T. Shepard und M. Gagne. A model of the F18 mission computer software for pre-run-time scheduling. In *10th Int. Conf. on Dist. Comp. Sys. (ICDCS '90)*, Seiten 62–69, Washington, DC, USA, Mai 1990. IEEE.

Fabian Scheler geboren 1981, begann im Jahr 2000 ein Informatik-Studium in Erlangen, welches er 2005 mit Auszeichnung abschloss. Im Anschluss daran trat er dort eine Stelle als wissenschaftlicher Mitarbeiter bei Prof. Dr.-Ing. Wolfgang Schröder-Preikschat am Lehrstuhl für Verteilte Systeme und Betriebssysteme an. Seine Tätigkeit verband er mit einer Lehrtätigkeit und intensiven Forschungsarbeiten auf den Gebieten Echtzeitsysteme, eingebettete Systeme und Betriebssysteme, die schließlich in seiner Dissertation mündeten. Im April 2011 promovierte Fabian Scheler schließlich mit Auszeichnung an der Technischen Fakultät der Friedrich-Alexander-Universität. Nach einem kurzen Aufenthalt bei der AREVA NP GmbH kehrte er im August 2011 an den Lehrstuhl für Verteilte Systeme und Betriebssysteme zurück, um die während seiner Doktorarbeit begonnene Forschung an *Atomic Basic Blocks* und dem *Real-Time Systems Compiler* im Rahmen des von der DFG geförderten AORTA-Projekts fortzuführen.