BAKKALAUREATSARBEIT

Modellierung und Scheduling von flexiblen, zeitgesteuerten Systemen

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich
Institut für Technische Informatik 182

durchgeführt von

Christian Paukovits
Matr.-Nr. 0127145
Linzer Straße 429/5/5204, A-1140 Wien

Wien, im September 2004 .....

Kurzfassung

Diese Arbeit präsentiert ein Konzept zur *generischen Modellierung* von TTP/A-Applikationen. Es wird ein Software-Prototyp vorgestellt, der die Spezifikation aus einer XML-Applikation bezieht und eine Round Defintion List (RODL) generiert. Zu diesem Zweck implementiert der Prototyp namens *TTP/A Scheduler* einen experimentellen Scheduling Algorithmus, den *TTP/A Scheduling Algorithmus*, der im Zuge dieser Arbeit entwickelt und hierin dokumentiert ist. Ebenso wird auf die Spezifikation einer TTP/A-Applikation mittels einer auf XML basierenden Spezifikationssprache, die im Rahmen dieser Arbeit erstmals erprobt worden ist, eingegangen.

Inhaltsverzeichnis

1 Einführung

1.1 Aufgabenstellung und Motivation

Ein *Embedded System* ist ein für einen bestimmten Aufgabenbereich konzipiertes Computersystem, das mit einem Minimum an Benutzerinteraktion betrieben werden kann (vgl. [Hea03], S. 2). Per Definition können Embedded Systems *monolitisch* oder *verteilt* (distributed) strukturiert werden[1].

Distributed Embedded Systems können durch mehrere Aspekte charakterisiert werden ([Kop97], S. 12 ff.), beispielsweise:

physikalische Anordnung: Einzelne Elemente (z.B. Sensoren und Aktuatoren) können räumlich voneinander getrennt sein, wie das in Fabriks- und Automationsanlagen der Fall ist.

Parallelität: bspw. "grid computing"

Fehlertoleranz: durch Einführung redundanter Rechnerknoten und Kommunikationskanäle

modularer Designansatz: bspw. zwecks Skalierbarkeit und Erweiterbarkeit

Ein Distributed Embedded System kann als Zusammensetzung beliebig vieler Subsysteme oder Komponenten betrachtet werden. Die Abbildung der funktionalen und zeitlichen Erfordernisse (im Folgenden als *"Modellierung"* bezeichnet) eines solchen Systems wird als Herausforderung erachtet, weil sich gegenseitig ausschließende Anforderungen an die Modellierung gestellt werden.

- Zum Einen muss das Modell strenge und eindeutige Ausdrucksweisen ermöglichen, da nur so kritische Parameter wie Timing und Verfügbarkeit mit ausreichender Präzision abgebildet werden können.
- Zum Anderen soll das Modell für unterschiedlichste Applikationen eingesetzt werden können, was ein gewisses Maß an Flexibilität und Erweiterbarkeit voraussetzt.

[1] Diese Arbeit beschäftigt sich ausschließlich mit Distributed Embedded Systems.

Diese Arbeit präsentiert ein Konzept zur *generischen Modellierung* von Distributed Embedded Systems, die das Time-Triggered Protocol for SAE Class A *(TTP/A)* [Kop02] zwecks Kommunikation einsetzen. Sie basiert auf der Publikation von Elmenreich, Pitzek und Schlager *Modeling Distributed Embedded Applications on an Interface File System* [EPS04].
Es wird eine XML-basierte Spezifikationssprache für die Modellierung und ein statischer, deterministischer Scheduling-Algorithmus vorgestellt, welcher aus der Spezifikation eine *Round Description List (RODL)* (siehe [EHK+02], S. 4 ff.) generiert.

Dieses Konzept wurde in einem Software-Prototypen, dem *TTP/A Scheduler*, realisiert. Ein derartiges Programm soll dem Entwickler von Distributed Embedded Systems ein leistungsstarkes Tool bieten, das die oben genannten strengen und eindeutigen Ausdrucksweisen (realisiert in einer XML-Appliaktion) anbietet, ohne auf Flexibilität und vor allem Komfort in der Modellierung verzichten zu müssen.

1.2 Struktur dieser Arbeit

Das Kapitel 2 beschäftigt sich mit den theoretischen Grundlagen der Modellierung von verteilten Echtzeitsystemen (*Distributed Real-Time Systems*) unter Einsatz eines Interface File Systems.
Kapitel 3 skizziert Aufbau und Funktionsweise des hier verwendeten Echtzeit-Kommunikationssystems *TTP/A*.
Das folgende Kapitel 4 befasst sich mit der Repräsentation einer TTP/A-Applikation für ein Distributed Embedded System in Form von XML.
Kapitel 5 beinhaltet das Kernstück dieser Arbeit, den *TTP/A Scheduling Algorithmus*. Es erklärt die Notwendigkeit dieses neu entwickelten Algorithmus, dessen Terminologie und stellt seine Funktionsweise vor.
Das letzte Kapitel 6 präsentiert den Software-Prototypen namens *TTP/A Scheduler*, der die Implementierung des im vorhergehenden Kapitel vorgestellten Algorithmus beinhaltet.

In den Anhängen sind konkrete Ergebnisse dieser Arbeit für das Fallbeispiel der *Smart-Fusion Applikation* zu finden. Dabei handelt es sich um

- die Document Type Definition (DTD) der XML-Repräsentation einer TTP/A-Applikation
- das Fallbeispiel der Smart-Fusion Applikation in Form einer solchen XML-Darstellung
- ein Log-File des TTP/A Schedulers, das die Arbeitsschritte des Programms illustriert

2 Conceptual Model

Die Applikation auf einem Distributed Embedded System wird zunächst *funktional* beschrieben, beispielsweise als Menge von verknüpften Echtzeit-*Services*. Ein Service ist gekennzeichnet durch seine

- Schnittstelle (Interface)
- Funktionalität
- sonstigen Eigenschaften, beispielsweise zeitliches Verhalten und Anforderungen an die Verfügbarkeit

2.1 Schnittstellen

Das Interface eines Services – illustriert in Abbildung 2.1 – kann weiter gegliedert werden:

Service Providing Linking Interface (SPLIF): Die Ergebnisse des Services werden über diese Schnittstelle zur Verfügung gestellt.

Service Requesting Linking Interface (SRLIF): Über diese Schnittstelle bezieht das Service seine benötigten Eingangsdaten.

Diagnostic and Management (DM): Parameter und Zustands- bzw. Debugging-Informationen des Knotens können über diese Schnittstelle abgerufen oder manipuliert werden.

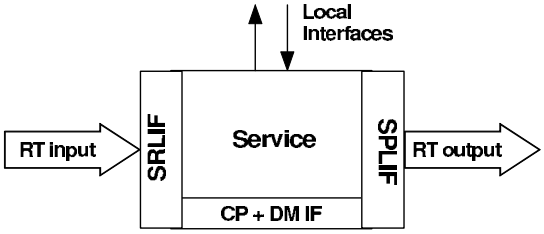

Abbildung 2.1: Schnittstellen eines Services

Configuration and Planning (CP): Diese Schnittstelle ist nicht zeitkritisch. Sie kann für die Integration neuer Services im Cluster und diverse administrative Dienstleistung herangezogen werden.

Local Interfaces: Diese Schnittstelle bezeichnet alle Verbindungen zu vom Service verwalteten Gerätschaften wie Sensoren oder Aktuatoren.

Typischerweise implementiert ein *Smart Sensor* das SPLIF, CP, DM und ein lokales Interface zu seinem physikalischen Sensor. Ein *Aktuator* besitzt eine Ausführung des SRLIF, CP, DM und ein lokales Interface zu seinem physikalischen Aktuator.

Der Datenfluss (*Data Flow*) zwischen SPLIF und SRLIF der einzelnen Knoten funktioniert durch *Ports*. Ein Port spezifiziert einen Namen, eine Beschreibung und die Struktur (im Wesentlichen den Datentpyen) der über ihn ausgetauschten Informationen.

2.2 Tasks

Das funktionale Verhalten oder die Funktionalität eines Services wird in sogenannten *Tasks* gekapselt. Ein Task konsumiert die Daten seines SRLIF und produziert seine Ergebnisse, die er im SPLIF nach Beendigung ablegt. Zum Zeitpunkt der Ausführung müssen die Daten in den beiden Schnittstellen konsistent gehalten werden, d.h. sie dürfen nicht manipuliert werden.

Eine Applikation besteht aus mindestens einem Service, das jeweils von einem Task realisiert wird. Diese Services bzw. Tasks kommunizieren über ihre SPLIFs und SRLIFs, die ein Echtzeit-Kommunikationssystem gebrauchen. Auf jedem Knoten im Cluster können ein oder mehrere Services exekutiert werden.

2.3 Abhängigkeiten

Üblicherweise behandelt die Modellierung einer Applikation die Funktionalität von Services und deren Datenfluss untereinander. Allerdings benötigt man für die vollständige Beschreibung weitere Informationen – sogenannte nichtfunktionale Eigenschaften (*non-functional properties*).

Davon sind zwei Klassen bekannt:

service-specific properties: Diese ergeben sich aus den Erfordernissen der Appliaktion (z.B. Aktualisierungsrate eines Sensors)

end-to-end requirements: Eigenschaften, die aus dem verteilten Design eines Distributed Systems abgeleitet werden.

Erstere werden in der Fein-Spezifikation der Services modelliert.
Die "end-to-end requirements" werden in der Modellierung durch sogenannte Abhängigkeiten (*Dependencies*) ausgedrückt. Beim derzeitigen Stand der Entwicklung konnten folgende Arten von Abhängigkeiten erfasst werden:

Connection: Diese Art von Abhängigkeit beschreibt den Datenfluss zwischen den Ports einzelner Services. Erwartungsgemäß verfügt sie über nähere Angaben über Quelle (Source) und Ziel (Target) der zu übertragenden Informationen. Ein Eingangsport kann nur eine einzelne eingehende Connection aufweisen, während ein Ausgangsport an mehrere gegenüberliegende Eingangsport übermitteln kann.

Causal (Kausalitäten): Diese Abhängigkeit drückt kausale Abhängigkeiten bzw. Ursächlichkeiten zwischen Services aus. Unter Berücksichtigung der Kausalitäten werden die Services entsprechend ihrer logischen Abfolge geordnet. Darüber hinaus beinhaltet eine Kausalität zeitliche Anforderungen (timing requirements) in Form eines Zeitpunktes (*instant*). Wie bei Connections beinhaltet die Abhängigkeit Informationen über Vorgänger (*before instant*) und Nachfolger (*after instant*).

Phase (Phasenbeziehungen): Diese Abhängigkeit kann dazu eingesetzt werden, um nicht-kausale, zeitliche Beziehungen zwischen Services auszudrücken. Beispielsweise kann ein Zeitversatz für die Ausführung von gleichrangigen Services modelliert werden.

3 Das TTP/A-Protokoll

In den vergangenen zwei Jahrzehnten wurde am Institut für Technische Informatik (Technische Universtität Wien) die *Time-Triggered Architecture* [Kop98] entwickelt. Sie bildet die theoretische Grundlage für die Interaktion zwischen den einzelnen Sensoren und Aktuatoren ("Transducer") eines Distributed Embedded Systems. Das *TTP/A-Protokoll* ist ein zeitgesteuertes (*time-triggered*) Feldbus-Protokoll und Abkömmling dieser Architektur, das für die Kommunikation zwischen Low-Cost Smart Transducer Nodes [EP03] prädestiniert ist.

3.1 Aufbau und Synchronisation

Das TTP/A-Protokoll implementiert eine Master/Slave-Architektur mit mindestens einem Master und mehreren Slaves, die eigenständige Knoten ("Nodes") im Netzwerk sind und einen *Cluster* formen. Ein aktiver Master kontrolliert jeweils einen *TTP/A Cluster*, darüberhinaus können sogenannte "Shadow Master" ins Netzwerk integriert werden, die beim Ausfall des primären Masters dessen Funktion übernehmen. Um sowohl systeminterne Zustände als auch Steuerinformation von übergeordneten Systemkomponenten mitzuteilen, kann ein Knoten ("Node") im Netzwerk als *Gateway* eingesetzt werden.

Der Zugriff auf das gemeinsame Kommunikationsmedium ("Bus") erfolgt mittels *Time Division Multiple Access (TDMA)*. Dieses Verfahren garantiert eine periodische, synchrone und vor allem kollisionsfreie Übertragung durch eine *a priori* Festlegung der Busbelegung durch die jeweiligen Knoten.
Jede Komponente des Clusters wird über eine eindeutige, vordefinierte Namensbezeichnung referenziert. Die dazu korrespondierende logische Namensidentifikation im System erfolgt durch den *Baptizing Algorithmus* [EHPS02], der die eigentliche Synchronisation der Knoten implementiert.

3.2 Struktur der Kommunikation

Abbildung 3.1 illustriert den Ablauf der Kommunikation mittels TTP/A.

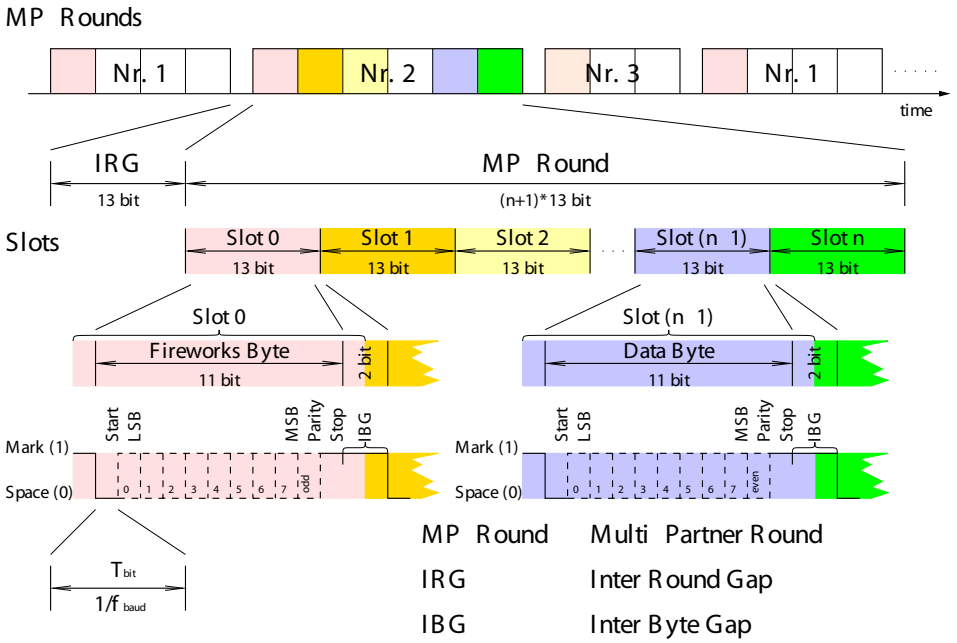

Abbildung 3.1: Abfolge von TTP/A Rounds und Slots innerhalb der Rounds

Das des TDMA-Prinzip teilt die Zeit in sogenannte *Slots*. Pro Slot darf nur ein einziger Knoten den Bus für sich beanspruchen, allerdings darf jeder Knoten den Zustand des Busses zu jedem Zeitpunkt mitlesen. Innerhalb dieses Intervalls wird vom jeweiligen Sender ein Byte an Daten zuzüglich Steuerinformationen (d.s. Startbit, Stopbit, Paritybit) übertragen.

Eine bestimmte Abfolge von Slots bildet eine Runde (*Round*). Es stehen verschiedene Arten von Runden zur Verfügung, die zwar untereinander unabhängig sind, aber durch eine Pause, die sogenannte *Inter-Round Gap*, zeitlich verzögert werden. Jede Round wird vom Master mit dem *Fireworks Byte* eröffnet. Es dient im weiteren Sinn als Synchronisationsereignis für die Knoten des Clusters, zusätzlich kennzeichnet es die Art der Runde.

Die Abfolge der weiteren Slots wird in der *Round Description List (RODL)* bestimmt. Sie definiert, welcher Knoten während des jeweiligen Slots sendet bzw. empfängt oder auch Tasks ausführt. In ihr findet sich die *a priori* Festlegung der Busbelegung durch die jeweiligen Knoten. Jeder Knoten wird mit einer RODL in seinem lokalen “File System” ausgestattet, die die Busbelegung und Aktivitäten für jeden Slot aus seinem individuellen Standpunkt darstellt.

Die Runden werden nach einer ebenfalls vorherbestimmten Abfolge periodisch abgearbeitet. Die sogenannte *Round Sequence (ROSE)* gibt Auskunft darüber, welche Art von Runde zum jeweiligen Zeitpunkt begonnen wird. Die ROSE ist nur für den Master relevant, da nur er mittels Fireworks Byte neue

Runden einleitet und sich alle Slaves nach diesem Start-Slot richten.

Es sind drei Arten von Runden definiert:

Multi-Partner Round: Dies ist die Rundenart, bei der applikationsspezifische Nutzdaten von den Knoten über den Bus ausgetauscht werden. Der Sinn darin besteht, dass die Knoten den Zustand ihrer Real-Time Images aktualisieren. Es ist möglich, mehrere Multi-Partner Rounds für einen Cluster zu verwenden.

Master-Slave Round: Während dieser Runde kann der Master Steuer- und Konfigurationsparameter an seine Slaves übergeben oder Daten aus deren File System auslesen. Dieser Vorgang wird auf zwei Runden aufgeteilt: Während der *Master-Slave-Address Round* spezifiziert der Master den Knoten, die Operation und die lokale Adresse für das relevante Datum. In der nachfolgenden *Master-Slave-Data Round* werden die angeforderten Daten übermittelt.

Broadcast Round: Diese Runde ähnelt der Master-Slave Round mit dem Unterschied, das mehrere Knoten gleichzeitig addressiert werden können.

3.3 Das Interface File System

Das Interface zu einem Service wird als ein "virtuelles" Shared Memory realisiert. Das lokale Interface File System (IFS), mit dem jeder Knoten des Clusters ausgestattet ist, dient als Quelle und Senke für alle über den Bus geschickten bzw. vom Bus gelesenen Daten – es vereint somit SPLIF, SRLIF sowie CP und DM der Service-Interfaces (siehe Abschnitt 2.1). Die Services eines Knotens bedienen sich des Interface File Systems, um die Informationen der Real-Time Images anderer Services abzurufen. Alle relevanten Daten eines Services werden ins Interface File System gemappt und sind somit grundsätzlich für andere Knoten greifbar. Die Menge aller lokalen Interface File Systeme der einzelnen Knoten bildet das "globale" Interface File System, das die Real-Time Images innerhalb des Clusters beinhaltet.

Ein lokales IFS kann bis zu 64 *Files* beherbergen, wobei jedes maximal 256 4-byte große *Records* fassen kann. Das Layout jedes Files ist statisch definiert, jedes File kann eine unterschiedliche Länge (d.h. Anzahl an Records) besitzen. Allerdings ist der erste Record (Record Nr. 0) für den *Header Record* reserviert, welcher administrative Informationen über das File speichert.

Für weitere Informationen über die Funktionsweise von TTP/A und seinem Interface File System sei auf [EHK+02] verwiesen. Die oben angeführten Erläuterungen sind für das Verständnis dieser Arbeit ausreichend.

4 Modellierung in XML

Dieses Kapitel befasst sich mit der Repräsentation einer TTP/A-Applikation für ein Distributed Embedded System mit Hilfe der Extensible Markup Language (XML).
Mittlerweile existiert eine experimentelle Document Type Definition (DTD), die den syntaktischen Aufbau einer solchen XML-Spezifikation beinhaltet. Diese DTD ist im Anhang A abgedruckt. Darüber hinaus ist im Anhang B das Fallbeispiel des Smart-Fusion Sensors (vgl. [EPS04]) angegeben, das die praktische Benutzung dieser Applikationsmodellierung veranschaulichen soll.

Nachfolgend wird der Aufbau einer XML-basierten TTP/A-Applikationsmodellierung und das Zusammenspiel der Tags erläutert.

4.1 Top-Level

Der Root-Node des XML-Element-Trees ist das *specification*-Element. Auch wenn ein XML-Parser das XML-File als syntaktisch "wohl-geformt" betrachtet, ist der Name dieses Elements kennzeichnend, dass es sich bei dieser XML-Applikation um die Spezifikation[1] einer auf TTP/A basierenden Distributed Embedded Application handelt.

4.1.1 Beschreibung der Zielhardware

Gemäß DTD muss jede Spezifikation eine Beschreibung der Zielhardware in Form des Elements *target* beinhalten. Zunächst werden für das Scheduling relevante Informationen unter *parameters* angegeben. Derzeit wird ausschließlich die Baudrate des TTP/A-Kommunikationssystems mittels Element *baudrate* definiert. Für zukünftige Entwicklungsstufen der Applikationsmodellierung ist es denkbar, dass die Angabe zusätzlicher Systemparameter innerhalb dieses Tags eingeführt wird.

Zusätzlich muss für das Scheduling durch den TTP/A Scheduler bekannt sein, über wieviele Knoten (Nodes) das Zielsystem verfügt – das Target kann

[1] daher die Bezeichnung "specification"

beliebig viele Knoten aufweisen, muss aber zumindest mit einem ausgestattet sein.

Mittels *node*-Elementen werden die Knoten dieses Clusters modelliert. Die Angabe der Modellbezeichnung des Microcontrollers und die Beschreibung des Knotens sind für das Scheduling unbedeutend, trotzdem sollen sie als Zusatzinformationen für den menschlichen Entwickler dienen. Unumgänglich ist jedoch die Angabe der Taktfrequenz des Microcontrollers in diesem Knoten im Tag *frequency*. Darüber hinaus muss jeder Knoten eindeutig identifiziert werden können – diese Information wird in das Attribut *nodeID* gepackt und trägt den DTD-Datentyp "ID". Durch diesen Datentyp kann der XML-Parser primitive semantische Validierungen vornehmen: beispielsweise kann er die Eindeutigkeit dieser Identifikation feststellen.

4.1.2 Modellierung von TTP/A-Applikationen

Als zweites Element in der obersten Ebene sind die *application*-Tags angesiedelt. Ein solches Element mit seinen Unterelementen (siehe Abschnitt 4.3) modelliert die eigentliche TTP/A-Applikation – die Service-Tasks, deren Abhängigkeiten und zeitliche Erfordernisse entsprechend Kapitel 2. Jede Spezifikation muss mindestens eine, maximal sechs Applikationsmodellierungen enthalten. Aus jeder Applikation wird eine eigene RODL generiert, die auf alle Knoten des Clusters verteilt wird.

Auch Applikationen müssen eindeutig identifiziert werden können, weshalb das Attribut *name* – erwartungsgemäß vom Datentyp "ID" – einbezogen wird. Darüberhinaus verfügen die *application*-Tags über ein Attribut namens *isActive*. Da bis zu sechs verschiedene Applikationen beschrieben werden können, aber letztendlich nur eine einzige im TTP/A Cluster exekutiert werden kann, gibt dieses Attribut an, welche Applikation dies ist. Folglich kann nur bei genau einem *application*-Tag der Attributwert "true" sein. Falls das Attribut nicht näher spezifiziert wird, erhält es den "Default Value" "false". Leider ist eine DTD nicht mächtig genug, um die Bedingung einer einzigen aktiven Applikation auszudrücken. Stattdessen muss diese Überprüfung vom Parser-Modul des TTP/A Schedulers vorgenommen werden.

4.2 Das Service-Node Mapping

Eine Applikation wird im Wesentlichen aus einzelnen Service-Tasks aufgebaut, die untereinander abhängig sind bzw. kommunizieren können. Über die Angabe funktionaler und zeitlicher Erfordernisse hinaus ist vor allem relevant,

auf welchem Knoten im Cluster jeder Service-Task exekutiert wird. Deshalb definiert die DTD pro Applikation genau einen Tag namens *mapping*. Innerhalb des *mapping* wird die Zuordnung zwischen Knoten und Service mittels der Empty-Tags *map* getroffen. Die relevante Information wird in zwei Attributen des *map*-Elements untergebracht:
das Attribute *node_ref* gibt den Knoten an, auf dem das Service im Attribut *service_ref* exekutiert wird. Beide Attribute sind vom Datentyp "IDREF". Der Inhalt des Attributes muss einen solchen Wert besitzen, der in einem anderen "ID"-Attribut festgelegt worden ist; d.h. diese beiden Attribute können nur Bezeichnungen von Nodes und Services beinhalten, die auch tatsächlich angegeben worden sind. Auf diese Weise kann der XML-Parser simple logische Verknüpfungen validieren, eventuelle Tippfehler ausschließen.

Wiederrum ist die Ausdruckskraft der DTD beschränkt – beispielsweise ist der Parser nicht fähig, eine Überprüfung durchzuführen, ob der eingetragene Attribut-Wert tatsächlich ein Service oder einen Knoten referenziert. Es kann nur bestätigt werden, dass eine ID mit dieser Bezeichnung existiert, aber nicht, dass sie entsprechend der Modellierungslogik angemessen ist. Diese Überprüfung muss ebenfalls das Parser-Modul des TTP/A Schedulers erledigen.

Darüberhinaus muss das Parser-Modul festellen, ob jedes Service genau einem Knoten zugewiesen worden ist. Zusammenfassend kann gesagt werden, dass ein Knoten beliebig viele Services beherbergen kann, aber ein Service darf nur auf einem einzigen Knoten exekutiert werden – in der Terminologie der "relationalen Datenbanken" würde dies einer "1:n-Beziehung" entsprechen.

4.3 Services, Dependencies & Deadlines

4.3.1 Beträge und Zeitpunkte

Ein Konstrukt in der XML-Modellierung tritt als Child-Element verschiedener Tags auf: die Kombination *amount-unit*. Diese beiden Elemente werden dazu eingesetzt, um numerische Werte bestehend aus Betrag und Einheit entsprechend ihrer englischen Bezeichnung abzubilden.

Der XML-Parser kann selbst keine semantische Überprüfung über den Wert des Betrags und der Einheit vornehmen – dies muss wiederrum das Parser-Modul des TTP/A Schedulers übernehmen. Grundsätzlich kann *amount* numerische Zahlen enthalten, sowohl ganze Zahlen als auch Gleitkommazahlen. Die Einheit dieses Betrages wird erwartungsgemäß in *unit* spezifiziert.

Folgende Einheiten, die sich auf den Betrag beziehen, sind erlaubt – andere Werte werden ignoriert:

ns die Einheit des unter *amount* angegebenen Wertes sind Nanosekunden.

us selbiges für Mikrosekunden

ms ... Millisekunden

s ... Sekunden

cycles Der unter *amount* eingetragene Wert meint Taktzyklen des Knotens, auf dem das jeweilige Service exekutiert wird. Diese Einheit ist nur für Zeitangaben bei Services sinnvoll (und erlaubt).

MHz Erwartungsgemäß drückt dies die Taktfrequenz eines Knotens aus, die innerhalb eines *frequency*-Tags (siehe Abschnitt 4.1.1) angegeben wird.

byte Der Wert in *amount* sind Bytes – nur in Zusammenhang mit dem *datasize*-Tag von Connections möglich (siehe Abschnitt 4.3.3).

Die bedeutenste Anwendung erfährt dieses Konzept in der Repräsentation von Zeitpunkten mittels dem *duration*-Element, beispielsweise bei Deadlines oder Ausführungszeiten von Services (siehe Abschnitt 4.3.2). Das Attribut *type* gibt an, welche Art von Zeitangabe das *duration*-Element beinhaltet.

Ein *duration*-Element kann folgende Arten von Zeitpunkten beschreiben.

bound dies deklariert den Betrag als exakten Zeitpunkt, beispielweise Deadlines bei Services

upper die Obergrenze *relativ* zur "bound"-Angabe

lower die Untergrenze *relativ* zur "bound"-Angabe

Da alle drei Typen von "Duration-Angaben" unabhängig voneinander sind, können sie auch unterschiedliche Zeiteinheiten aufweisen.

Die Spezifikation der Baudrate bei der Beschreibung der Zielhardware (Abschnitt 4.1.1) stellt eine Abweichung von diesem Konzept dar. Der numerische, ausschließlich ganzzahlige Wert der Baudrate kann direkt als Inhalt des *baudrate*-Tags eingesetzt werden, ohne vom *amount-unit*-Paar Gebrauch zu machen.

4.3.2 Services

Die zeitlichen Rahmenbedingungen von Service-Tasks jeder Applikation werden in den *service*-Elementen modelliert.

Dem derzeitigen Entwicklungsstand zufolge sind das drei Eigenschaften:

Deadlines: der Zeitpunkt, bis zu dem das Service spätestens ausgeführt werden muss

Execution Time: die Ausführungszeit $WCET_{TASK}$ des Service-Tasks

Phasenbeziehungen: siehe Abschnitt 4.3.3

Diese Eigenschaften werden in den *property*-Tags abgebildet. Das Attribut *name* gibt Auskunft darüber, um welche Art von zeitlicher Information es sich handelt: "deadline" meint eben die Deadline von Services und Abhängigkeiten, "exectime" die Ausführungszeit bei Services.

Innerhalb der *property* muss zumindest eine *duration* definiert werden, die einen exakten Zeitpunkt ("bound") beschreibt. Darüberhinaus können noch Ober- bzw. Untergrenzen optional einbezogen werden (siehe Abschnitt 4.3.1). Erwähnenswert ist, dass laut DTD Ober- und Untergrenze immer gemeinsam vorkommen müssen. Somit hat ein *property*-Element syntaktisch korrekt genau eine *duration* für die exakte Zeitangabe, oder aber drei für Zeitangabe, Ober- und Untergrenze.

4.3.3 Dependencies

Gemäß konzeptionellem Modell (Kapitel 2) kann die XML-Modellierung die Abhängigkeiten für jede Applikation darstellen. Für jede Art existiert ein eigener Tag: *connection* für Connection-Beziehungen, *causal* für Kausalitäten und *phase* für Phasenbeziehungen. Jeder dieser drei Element-Arten besitzt ein Attribut *name*, das die Abhängigkeit identifiziert – es ist folglich vom Datentyp "ID".

In den folgenden Unterabschnitten wird auf die Abbildung der einzelnen Arten von Abhängigkeiten eingegangen.

Kausalitäten in XML

Zusätzlich zum Attribut *name* besitzt eine Kausalität das Attribut *beginner*. Es ist ein boolsches Attribut (obwohl es auch nur vom Typ CDATA ist); es kann die Werte "true" oder "false" annehmen. Wenn es nicht näher spezifiziert wird, erhält es automatisch den Standardwert "false".
An dieser Stelle soll nicht näher auf die Bedeutung des *beginner*-Attributes eingegangen werden. Es ist nur für die Auflösung von *Zyklen* relevant und wird folglich im entsprechenden Abschnitt 5.6 gesondert erläutert.

Ähnlich den Services kann ein *causal*-Element *property*-Beschreibungen aufnehmen. Damit kann die Deadline mit Zeitpunkt, Ober- und Untergrenze der

kausalen Abhängigkeit modelliert werden, andere Arten von *property* sind für Kausalitäten nicht angemessen und werden ignoriert.

Darüber hinaus existiert für diese Abhängigkeit ein besonderer Empty-Tag, der *instant*-Tag. Eine Kausalität muss genau zwei dieser Tags enthalten, was wiederrum mangels Ausdruckskraft der DTD vom Parser-Modul des TTP/A Schedulers überprüft werden muss. Ein *instant* deklariert den Vorgänger (*before instant*), der andere den Nachfolger (*after instant*) der Kausalität. Diese Information kann in den Attributen verpackt werden. Das Attribut *type* deklariert durch "before" bzw. "after", ob der aktuelle Tag den Vorgänger oder Nachfolger beschreibt. Das zweite Attribut *service_ref* referenziert das entsprechende Service. Dieses Attribut besitzt den Datentyp "IDREF" und kann somit bekannte Identifikationen aufnehmen. Allerdings kann der XML-Parser nicht sicherstellen, dass hierbei auf ein Service gezeigt wird – es könnte ja jede beliebige bekannte ID sein. Diese Bedingung muss wie so häufig das Parser-Modul des TTP/A Schedulers testen.

Connections in XML

Augenfällig an den *connection*-Elementen ist ihr Attribut *causal_ref*. Diese "IDREF" soll auf die zur Connection gehörenden Kausalität verweisen. Und wieder tritt die leidige Schwäche der DTD zu Tage, dass sie nur die Existenz der referenzierten ID prüfen kann, aber nicht deren semantische Bedeutung. Folglich muss das Parser-Modul des TTP/A Schedulers sicherstellen, dass *causal_ref* die Identifikation einer *causal* enthält.
Aus dem konzeptionellen Modell folgt, dass eine Connection immer genau einer Kausalität zugeordnet ist. Umgekehrt gilt das nicht! Eine Kausalität kann auch ohne Connection vorkommen. In der Terminologie der "relationalen Datenbanken" entspricht dies einer "1:0-Beziehung".

Eine Connection verursacht eine Kommunikation zwischen Service-Tasks über das Echtzeitsystem. Das *datasize*-Element verwendet das unter Abschnitt 4.3.1 vorgestellte *amount-unit*-Paar, um die Anzahl der transferierten Bytes auszudrücken. Jede *connection* enthält genau eines dieser Tags.

Die Richtung des Datenflusses wird über die *dataflow*-Empty-Tags modelliert, verpackt in den Attributen. Es dürfen nur zwei Empty-Tags pro Connection auftreten: einer für die Quelle (Source), der andere für das Ziel (Target) der Datenübertragung. Das Attribut *direction* bestimmt, welchen Kommunikationsteilnehmer der aktuelle Tag beschreibt. Das Attribut *service_ref* entspricht dem gleichnamigen Attribut der *instant*-Tags der Kausalitäten. Es referenziert das Service, das an der Kommunikation beteiligt ist. *Port* ist ein zusätzliches Attribut, das den "Port" entsprechend dem konzeptionellem Modell benennt.

Phasen in XML

Phasen ähneln den Kausalitäten – ein *phase*-Element enthält nämlich die selben Unterelemente wie ein *causal*. Folglich lässt sich Deadline der Phase mit einem *property*-Element modellieren. Dabei sind Zeitpunkt, Ober- und Untergrenze möglich. Als einzigen Unterschied weist das *name*-Attribut von *property* den Wert "phase" auf.

Die zwei *instant*-Empty-Tags referenzieren jeweils ein Service, dadurch wird die Phasenbeziehung hergestellt. Grundsätzlich ist es egal, welchen Wert das *type*-Attribut enthält. Phasenbeziehungen sind ohnehin *symmetrisch*, folglich existiert kein Vorgänger und Nachfolger. Außerdem gilt für Phasen das Gesetz der Transitivität.

5 Der Scheduling Algorithmus

5.1 Warum ein neuer Algorithmus?

Ein hartes Echtzeit-System (*hard real-time system*) muss seine Tasks dermaßen ausführen, dass die zeitkritischen Deadlines gemäß Spezifikation eingehalten werden. Jeder Task benötigt Rechenleistung und Speicherplatz. Das "Scheduling Problem" behandelt die Zuteilung dieser Resourcen, um die zeitlichen Anforderungen erfüllen zu können.

Es sind eine Vielzahl von Algorithmen bekannt, die diesem Problem gewidment sind. Sie können unter anderem in *dynamische* (das Scheduling wird zur Laufzeit bestimmt) und *statische* (das Scheduling wird a priori zur Compilierzeit berechnet) Algorithmen eingeteilt werden.

Unter [Kop97], S. 228 ff. werden die populärsten Algorithmen vorgestellt, wie beispielsweise

- Rate Monotonic Scheduling
- Earliest-Deadline-First (EDF)
- Least-Laxitiy (LL) Algorithmus

Allerdings haben diese Algorithmen eine konzeptionelle Schwäche: sie sind ausschließlich für das Scheduling von *nicht-abhängigen (independent) Tasks* geeignet. Gemäß konzeptionellem Model (Kap. 2) treten bei TTP/A-Applikationen jedoch solche Abhängigkeiten (Dependencies) auf. Folglich sind alle etablierten Real-Time Scheduling Algorithmen für diese Aufgabenstellung ungeeignet!

Dieser Umstand führte zur Entwicklung des *TTP/A Scheduling Algorithmus* im Rahmen dieser Arbeit, der für den Einsatz in TTP/A-Applikationen maßgeschneidert ist. Im Folgenden wird dieser Algorithmus und dessen Grundlagen näher vorgestellt.

5.2 Begriffe aus der Graphentheorie

5.2.1 Kanten & Knoten

Ein *Graph* ist ein Tupel (V, E) wobei V eine endliche Menge ist, deren Elemente *Knoten* (*vertices*, *nodes*) genannt werden. Die Menge E besteht aus Paaren von Knoten. Sind diese geordnet ($E \subseteq V \times V$), spricht man von *gerichteten* Graphen, sind die Paare ungeordnet, spricht man von *ungerichteten* Graphen. Die Elemente in E heißen gerichtete bzw. ungerichtete *Kanten* (*edges*). Eine Kante (v, v) heißt *Schleife* (*loop*).

Ist $e = (v, w) \in E$, dann sagen wir:

- v und w sind *adjazent*
- v (bzw. w) und e sind *inzident*
- v und w sind *Nachbarn*

5.2.2 Nachbarmengen

Die eingehende Nachbarmenge eines Knotens $v \in V$ ist definiert als

$$N^+(v) = \{u \in V \mid (u, v) \in E\}$$

und die ausgehende Nachbarmenge als

$$N^-(v) = \{w \in V \mid (v, w) \in E\}$$

Für ungerichtete Graphen gilt: $N^+(v) = N^-(v) \rightarrow N(v)$.

5.2.3 Knotengrade

Der Ausgangs- bzw. Eingangs-Knotengrad von $v \in V$ ist definiert als $d(v)$ und bezeichnet die Anzahl seiner ausgehenden $d^-(v)$ bzw. eingehenden $d^+(v)$ inzidenten Kanten.

5.3 Darstellung von Graphen im Rechner

Die zwei populärsten Möglichkeiten, um einen Graphen im Rechner abzubilden, sind

Adjazenzmatrizen und *Adjazenzlisten*.

Sie unterscheiden sich in Hinblick auf den benötigten Speicherplatz und die benötigte Laufzeit für die bei manchen Algorithmen auftretenden, wiederkehrenden Abfragen. Diese Eigenschaften sollen im Rahmen dieser Arbeit nicht näher untersucht werden. Stattdessen wird ausschließlich auf Adjazenzmatrizen eingegangen, weil nur diese für den TTP/A Scheduling Algorithmus relevant sind.

Sei M eine $n \times n$-Matrix mit Einträgen

$$M[u,v] = \begin{cases} 1 & \text{falls } (u,v) \in E \\ 0 & \text{sonst} \end{cases}$$

Dies bezieht sich auf Graphen ohne Mehrfachkanten zwischen einem Knotenpaar. Alternativ könnte statt 1 die Anzahl der Kanten von u nach v eingetragen werden. Dadurch können auch Mehrfachkanten ausgedrückt werden. Ungerichtete Graphen besitzen symmetrische Adjazenzmatrizen, gerichtete im Allgemeinen nicht.

Programmiertechnisch lässt sich eine Matrix als 2-dimensionales Array darstellen.

5.4 Precedence Graphs

5.4.1 Struktur des Graphen

Eine TTP/A-Applikation kann wie in Kapitel 4 textuell als XML-Applikation repräsentiert werden. Davon ausgehend können die Abhängigkeiten und Services laut konzeptionellem Modell (siehe Kap. 2) als Graph dargestellt werden. Abbildung 5.1 zeigt die Smart-Fusion Applikation dargestellt als Graph. Die Transformation der Spezifikation in einen Graphen bzw. dessen Adjazenzmatrix bildet die Grundlage für den Scheduling Algorithmus.

Die Knoten verkörpern die Service-Tasks, währenddessen die Kanten Kausalitäten ausdrücken. Da die Kanten als Pfeile abgebildet werden, handelt es sich um einen gerichteten Graph. Der Knoten, der ein Service darstellt[1] und mit dem Anfang der "Pfeilkante" verbunden ist, ist der *before instant* der Kausalität. Erwartungsgemäß ist der Serviceknoten, auf den die Pfeilspitze zeigt, der *after instant* der kausalen Abhängigkeit.
Zusammenfassend führt eine gerichtete Kante in Form eines Pfeiles vom

[1] fortan bezeichnet als "Serviceknoten"

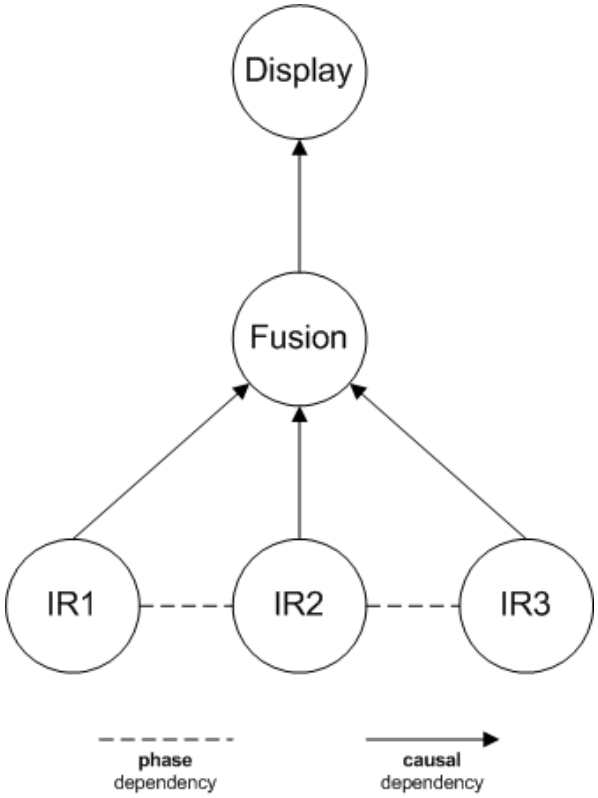

Abbildung 5.1: Precedence Graph der Smart-Fusion Applikation

Vorgänger zum Nachfolger der entsprechenden Kausalität, wobei die beiden Services als Knoten im Graphen abgebildet werden.

Das Geflecht aus den Serviceknoten und den Pfeilkanten modelliert ganzheitlich die kausalen Abhängigkeiten zwischen allen Service-Tasks der Applikation. Für den daraus resultierenden Graphen soll der Begriff *Precedence Graph* eingeführt werden.

Wie in Abbildung 5.1 ersichtlich ist, enthält der Graph nur die als Knoten verkörperten Service-Tasks und die gerichteten (durchzogenen) Kanten, die kausale Abhängigkeiten abbilden. Grundsätzlich ist es auch möglich, zwischen den Serviceknoten auch die zur Kausalität gehörende Connection einzuzeichnen, sofern eine solche existiert. Der Übersicht halber wird diese Kante ausgelassen, da ihr Informationsgehalt gering ist und ohnehin der Kante der Kausalität entspricht. Darüberhinaus ist es sehr selten, dass eine Kausalität ohne eine Connection auftritt – somit kann diese Kante beide Abhängigkeiten repräsentieren.

Eine Phasenbeziehung zwischen zwei Services wird als strichlierte Kante dargestellt. Laut Abschnitt 4.3.3 sind Phasenbeziehungen symmetrisch, folglich

ist eine Phasen-Kante ungerichtet. Darüberhinaus gilt das Gesetz der Transitivität, d.h. $IR1 \parallel IR2 \wedge IR2 \parallel IR3 \rightarrow IR1 \parallel IR3$. Die letzte, schlussfolgernde Phasenbeziehung ist daher nicht mehr in der Abbildung enthalten, um den Precedence Graph möglichst einfach zu halten.

5.4.2 Merkmale des Precedence Graph

Dieser Abschnitt listet die Merkmale und Einschränkungen auf, die dem Precedence Graph auferlegt sind. Daraus lassen sich die Charakteristika des darauf basierenden Scheduling Algorithmus ableiten. Es ist schlüssig, dass mit zunehmender Entwicklung des Algorithmus[2] Änderungen dieser Bedingungen auftreten können.

Bezeichnungen

Nachfolgend kommen wiederholt Mengenterme vor, die an dieser Stelle erläutert werden:

$\mathcal{S}$	...	Menge der Services der Applikation
$\mathcal{K}$	...	Menge der Kausalitaeten
$\mathcal{C}$	...	Menge der Connection-Dependencies
$\mathcal{P}$	...	Menge der Phasenbeziehungen
V	...	Menge der Knoten des Precedence Graph
E	...	Menge der Pfeilkanten für Kausalitaeten
E'	...	Menge der ungerichteten Kanten für Phasen

Precendence Graph vs. Applikationsmodell

Im Folgenden werden die Ausdrücke aus der Begriffswelt des Precedence Graph den entsprechenden Termini aus der Applikationsmodellierung zugeordnet.

$$V \models \mathcal{S}$$
$$E \models \mathcal{K},\ \mathcal{C} \subseteq \mathcal{K}$$
$$E' \models \mathcal{P}$$

[2]Dieser Algorithmus ist nach wie vor in Entwicklung und experimentell!

Bedingungen für den Precedence Graph

Bisher konnten zwei Bedingungen ausgemacht werden, die ein gültiger Precendence Graph erfüllen muss:

$$\exists\, x \in E \mid x = (u, w),\ (u, w) \in V\ \wedge\ u \neq w \tag{5.1}$$

$$\nexists\ \{\ y = (u, w) \in E' \mid (u \rightsquigarrow w \in E) \vee (w \rightsquigarrow u \in E),\ u, w \in V\ \} \tag{5.2}$$

Die Bedingung 5.1 behandelt das Vorkommen von kausalen Abhängigkeiten.

1. Die Kanten $x \in E$, die gemäß $E \models \mathcal{K}$ die Kausalitäten im Precedence Graph darstellen, können nicht zum selben Knoten führen ("Schleifen"), weil $u \neq w$. Im Sinne des konzeptionellen Modells bedeutet dies, das ein Service keine Kausalität mit sich selber eingehen kann, d.h. nicht von sich selber abhängen kann.

2. Im weiteren Sinne bedeutet die obige Bedingung ebenfalls, dass ein Service keine Connection zu sich selbst auf Grund von $\mathcal{C} \subseteq \mathcal{K}$ besitzt. Ein Service kann sich selber keine Daten über das Echtzeit-Kommunikationssystem schicken.

3. Der Existenzquantor verdeutlicht, dass es zwischen zwei Knoten des Graphen nur eine einzige gerichtete Kante bzw. keine gerichteten Mehrfachkanten geben darf. In der Begriffswelt der Applikationsmodellierung bedeutet diese Bedingung, dass zwischen zwei Services genau eine kausale Abhängigkeit – auf Grund von $\mathcal{C} \subseteq \mathcal{K}$ auch Connection – existieren darf.

Für Phasenbeziehungen drückt Bedingung 5.2 folgende Einschränkungen aus:

1. Es darf keine Phasenbeziehung zwischen Services bestehen, die direkt bzw. mit dazwischenliegenden Services voneinander kausal abhängig sind. Diese Tatsache lässt sich im Graphen insofern ablesen, als dass es keinen Pfad entlang der "Kausalitätskanten" zwischen den an der Phase beteiligten Service-Knoten geben darf.
   Beispielsweise sind in Abbildung 5.1 die Services "IR1" und "Fusion" direkt voneinander kausal abhängig, wodurch keine Phasenbeziehung bestehen darf. Ebenso darf keine Phase zwischen "IR1" und "Display" vorkommen, da "Display" von "Fusion", dieses wiederum von "IR1" abhängig ist – es besteht hier ein Pfad zwischen den beteiligten Services auf Grund der Kausalitäten.

Kantentyp	Eintrag
keine Kante	
Kausalität	"*causal*"
Connection	"*connection*"
Phase	"*phase*"

Tabelle 5.1: Struktur eines Eintrages in der Adjanzenzmatrix

5.4.3 Precedence Graph als Adjazenzmatrix

Der TTP/A Scheduling Algorithmus verwendet als grundliegende Datenstruktur eine Adjazenzmatrix, die den Precedence Graph abbildet. Die tabellenartike Struktur solcher Matrizen erlaubt eine simple und effiziente Bearbeitung.

Die Adjazenzmatrix PM eines Precedence Graphen (genannt *Precedence Matrix*) ist eine $n \times n$-Matrix, wobei $n = |\mathbb{S}| = |V|$. Ihr Name beruht auf der Tatsache, dass sie das Geflecht aus Services und deren Abhängigkeiten in das tabellenartige Gerüst einer Adjanzenzmatrix einhüllt.

Entgegen der Einführung von Adjanzmatrixen in Abschnitt 5.3 wird im Falle des TTP/A Scheduling Algorithmus diese Struktur erweitert. Anstatt 0 und 1 in den Elementen der Matrix einzutragen, das eventuell eine Kante zwischen den indizierten Knoten anzeigt, wird eine komplexere Datenstruktur benutzt. Dieses Konzept ermöglicht nicht nur die Existenzabfrage einer Kante, sondern verpackt Zusatzinformationen, um welche Kante des Precedence Graphs es sich handelt. Die Tabelle 5.1 skizziert deren Aufbau.

Jedes Element der Adjanzenzmatrix $PM[i, j]$ wird durch eine solche Datenstruktur verkörpert. Deren Bestandteile sind Zeiger bzw. Referenzen auf die jeweiligen Speicherabbildungen von Kausalitäten, Connections bzw. Phasenbeziehungen. Somit wird für jeden der drei Kantentypen des Precedence Graph pro Element der Adjazenzmatrix ein Verweis auf die dazugehörige Dependency bereitgestellt.

Der Zeilen- und Spaltenindex $[i, j]$ des betreffenden Matrixelements bezeichnet die an der Abhängigkeit beteiligten Services. Im Falle von gerichteten Kanten wie bei kausalen Abhängigkeiten lässt sich auf Grund der Indizes die Richtung der Kante impliziet ablesen: der Zeilenindex i meint den Ursprung der Kante, der Spaltenindex j das Ende bzw. Pfeilspitze der Kante.

Laut Bedingung 5.1 kann ein Precedence Graph keine Mehrfachkanten enthalten. Somit genügt eine einzelne Instanz der Datenstruktur, um die einzig möglichen Abhängigkeiten – nur eine Kausalität, Connection und/oder eine Phasenbeziehung – zwischen je zwei Services zu modellieren. Darüber hinaus können keine Services mit sich selbst Abhängigkeiten eingehen, wodurch

Service	IR1	IR2	IR3	Fusion	Display
IR1	—	**phaseIR1IR2**		*IR1toFusion* *IR1connection*	
IR2	**phaseIR1IR2**	—	**phaseIR2IR3**	*IR2toFusion* *IR2connection*	
IR3		**phaseIR2IR3**	—	*IR3toFusion* *IR3connection*	
Fusion				—	*FusionToDisplay* *Fusionconnection*
Display					—

Tabelle 5.2: Precedence Matrix der Smart-Fusion Applikation

die Einträge der Diagonale der Matrix $PM[i,i],\ i = 1 \ldots n$ automatisch als ungültig zu behandeln sind.

Für Phasenbeziehung gilt die Symmetrieeigenschaft laut Bedingung 5.2. Eine Phase muss dementsprechend "symmetrisch" entlang der Diagonale gespiegelt werden – es ist ein Verweis in den Elementen $[i,j]$ und $[j,i]$ der Matrix erforderlich, ansonsten ist die Matrix ungültig.

Bleibt ein Zeiger bzw. eine Referenz der drei möglichen Verweise eines Elements $PM[i,j].causal$, $PM[i,j].connection$ oder $PM[i,j].phase$ ungenützt, bedeutet dies, dass zwischen den Services i und j keine derartige Abhängigkeit besteht.

Die Tabelle 5.2 veranschaulicht die Precedence Matrix der Smart-Fusion Applikation.

5.5 Konzept des Algorithmus

Dieser Abschnitt erläutert die grundlegende Arbeitsweise des TTP/A-Algorithmus. Als Voraussetzung muss dem Algorithmus eine Precedence Matrix PM übergeben werden, die den Precedence Graph der zu berechneden TTP/A-Applikation beinhaltet.

Listing 5.1: Pseudocode

```
Main():
solange unmarkierte Zeilen in PM existieren
  Candidates = {x ∈ (V ⊨ 𝕊) | N+(x) = {∅} bzw. d+(x) = 0}
    falls Candidates = {∅}
      // Zyklus: suche Beginner-Services!
      FindBeginner()

  ordne Candidates nach EDF

  // bestimme, ob Candidate-Services Phasenbeziehungen besitzen
  ∀a, b ∈ Candidates, a ≠ b
    falls (a, b) ∈ (P)
      // berechne Phasen-Offsets
      CalcOffset(a,b)

  // Ausführung der Candidate-Services in Dispatcher-Table eintragen
  ∀c ∈ Candidates
    TargetNode(c).DispatcherTable[localindex + c.Offset] = OP_EXEC
    localindex = c.Offset + ExecutionTime(c)

  // Ergebnis der Candidate Services verschicken
  ∀c ∈ Candidates
    // Empfangsoperation für abhängige Empfänger-Services eintragen
    Receivers = {y ∈ (V ⊨ 𝕊) | PM[c, y].connection ≠ NULL, y = 1...|𝕊|}
    ∀r ∈ Receivers
      TargetNode(r).DispatcherTable[localindex] = OP_RECEIVE
      localindex = localindex + #SendBytes
    // Sendeoperation des ursprünglichen Candidate-Services eintragen
    TargetNode(c).DispatcherTable[localindex] = OP_SEND
    localindex = localindex + #SendBytes

  // Candidate-Services als "done" markieren
  ∀c ∈ Candidates
    markiere row(c) in PM

CalcOffset(a,b):
  φ = #( (a, b) ∈ 𝒫 )
  falls TargetNode(a) = TargetNode(b)
    falls [φ − ExecutionTime(b)] > 0
        b.Offset = b.Offset + φ − ExecutionTime(b)
  sonst
    b.Offset = b.Offset + φ

FindBeginner():
  ∀e ∈ PM
    falls e.causal → beginner = true
      Candidates = Candidates ∪ row(e)

  // Überprüfungen, ob Abhängigkeiten zwischen Beginner-Services bestehen
```

```
∀a, b ∈ Candidates, a ≠ b
   falls (a, b) ∈ K
      // "simple" Abhängigkeit
      Candidates = Candidates \ b
   falls (a, b) ∈ K ∧ (b, a) ∈ K
      // Zyklus zwischen Beginner-Services
      error
      abort

falls Candidates = {∅}
   // Zyklus konnte nicht aufgelöst werden
   error
   abort
```

Der TTP/A Scheduling Algorithmus gehört zur Klasse der statischen Scheduling Algorithmen. Er berechnet die Dispatcher Table (bzw. *RODL*) einer TTP/A-Applikation vor deren Exekution am Zielsystem. Das Ergebnis des Schedulings wird in die Sourcen der Applikation eingebunden und kompiliert.

Darüberhinaus kann dieser Algorithmus als ein *Greedy Algorithmus* eingestuft werden, der auf Graphen – *Precedence Graphs* – operiert. Der TTP/A Scheduling Algorithmus weist die typischen Merkmale dieser Klasse auf:

- Der Algorithmus erreicht eine approximativ optimale Lösung.
- Das Ergebnis ist unter Umständen nicht optimal, dafür *garantiert* es die Einhaltung aller Deadlines der Applikation.
- Die Lösung wird iterativ und *bottom-up* aufgebaut.
- Eine getroffene Entscheidung wird nicht mehr geändert, auch wenn ein kleiner Schritt zurück eine bessere Lösung bringen würde.
- Auf Grund dieser "Sturheit" ist die Laufzeit effizient und deterministisch (annähernd linear[3]).

Die Kernidee des TTP/A Scheduling Algorithmus ist, alle Service-Tasks, die momentan nicht durch Kausalitäten von anderen Services abhängen (die sogenannten *Candidates*), möglichst gleichzeitig auszuführen. Im Sinne der Graphentheorie lässt sich diese Bedingung daran ablesen, dass ein Serviceknoten keine eingehenden Kausalitätskanten im Precedence Graph besitzt.
In diesem Fall wäre die Menge der eingehenden Kanten für den Service-Knoten

[3] Abgesehen von der Tatsache, dass der Algorithmus immer wieder Subroutinen wie Sortieren nach EDF einsetzt. Diese sollen als Konstante betrachtet werden.

x gleich $N^+(x) = \{\emptyset\}$, bzw. wäre der eingehende Knotengrad $d^+(x) = 0$. Folglich wären in der Precedence Matrix für alle vertikalen Einträge (mit gleichbleibender Spalte) des Service x die "causal"-Verweise ungenützt:
$N^+(x) \iff \forall i,\ PM[x,i].causal = NULL.$

Da der Algorithmus auf die Deadlines der Service-Tasks Rücksicht nehmen muss, werden alle Candidates nach *Earliest Deadline First (EDF)* sortiert. Sofern die Service-Tasks auf unterschiedlichen Target-Knoten ausgeführt werden, werden sie auch gleichzeitig exekutiert, wodurch diese Sortierung irrelevant wird. Allerdings garantiert EDF die vorangehende Ausführung des Services mit der kürzeren Deadline, falls mehrere Services auf dem gleichen Target-Knoten beherbergt werden.

Sofern einige Candidates in Phasenbeziehungen zueinander stehen, verursacht die "Phasenverschiebung" einen späteren Startpunkt der Service-Exekution. Wie im Pseudocode ersichtlich ist, wird der "Offset" nur dem zweiten Service b aufgebürdet. Nachdem die Services zuvor nach Earliest Deadline First sortiert worden sind, ist garantiert, dass b die längere Deadline als a besitzt. Dadurch ist es wahrscheinlicher, dass trotz verzögertem Start von b die Deadline eingehalten wird.

Schließlich können für jeden Candidate "Execution Slots", die seiner Ausführungszeit entsprechen, in der Dispatcher-Table (bzw. RODL) des ihm zugewiesenen Target-Knotens reserviert werden. Der Algorithmus rechnet grundsätzlich in TTP/A Slots (siehe Abschnitt 3.2), die von der Übertragungsgeschwindigkeit des Echtzeit-Kommunikationssystems abhängen. Beispielsweise bei einer Baudrate von 9600 dauert ein Slot $\frac{1}{9600} * 13 = 1354.17\ \mu s$[4]. Auch wenn ein Service nur einen Bruchteil dieser Zeit zur Ausführung benötigt, muss dafür ein ganzer Slot reserviert werden – bei der Vergabe von TTP/A Slots muss also immer auf Ganze *aufgerundet* werden.

Durch die Reservierung von Slots für Sende-, Empfangs oder Ausführungsoperationen wird im Allgemeinen der Index bzw. momentante Position (im Pseudocode bezeichnet als *localindex*) in der RODL des betroffenen Target-Knotens weitergesetzt. Dies verhindert, dass ein Slot mehrfach belegt wird.

Nach dem Eintragen der Service-Exekution wird das Versenden der Ergebnisse der Candidates in der RODL eingeschrieben.
Zunächst wird ein (oder mehrere – je nach Anzahl der verschickten Bytes[5]) entsprechender "Receive Slot" beim Target-Knoten aller Empfänger-Services reserviert.

[4]Der Faktor 13 ergibt sich aus der Tatsache, dass ein Slot aus den 8 Datenbits, jeweils 1 Start-, Stop und Paritybit und der "Inter-Byte-Gap" von 2 Bitzeiten besteht.
[5]beschrieben im *datasize*-Element einer Connection (siehe Abschnitt 4.3.3)

	IR_node1	IR_node2	IR_node3	display_node	master_node
0	Execute(IR1)	Execute(IR2)	Execute(IR3)	—	—
1	Send(IR1)	—	—	—	Receive(IR1)
2	—	Send(IR2)	—	—	Receive(IR2)
3	—	—	Send(IR3)	—	Receive(IR3)
4	—	—	—	—	Execute(fusion)
5	—	—	—	Receive(fusion)	Send(fusion)
6	—	—	—	Execute(display)	—

Tabelle 5.3: RODLs für die Smart-Fusion Applikation

Empfänger (im Pseudocode *Receivers*) sind alle Services, auf die mittels "connection"-Verweis entlang aller Einträge der Matrix-Zeile des Candidates c gezeigt wird: $Receivers = \{y \in (V \models \mathcal{S}) \mid PM[c,y].connection \neq NULL,\ y = 1 \ldots |\mathcal{S}|\}$.
Letztendlich wird der "Send Slot" beim Target-Knoten des jeweiligen Candidates installiert.

Nach jeder Iteration werden die Zeilen aller Candidates in der Precedence Matrix markiert. Sie gelten somit als abgearbeitet ("done"). Ausmarkierte Zeilen werden beim Bestimmen der Candidates für den nächsten Durchgang nicht mehr einbezogen. Auf diese Weise wird in der Precedence Matrix abgebildet, dass nun die kausale Abhängigkeit zu anderen Services aufgelöst worden ist. Andere Services, die zuvor noch von den Candidates der vergangenen Runde abhängig gewesen sind, können als Candidate bestimmt werden.

Der Algorithmus terminiert, wenn alle Zeilen der Precedence Matrix markiert sind. Das bedeutet, dass alle Services abgearbeitet und alle damit bedingten Operationen in die RODLs eingetragen worden sind.

Falls keine Candidates in der aktuellen Iteration gefunden werden können, liegt ein *Zyklus* in kausalen Abhängigkeiten vor. Dieser Sonderfall wird in folgenden Abschnitt 5.6 behandelt.

Die Tabelle 5.3 zeigt eine schematische Darstellung der RODLs für die Smart-Fusion Applikation, wie sie der TTP/A Scheduling Algorithmus erstellt.

5.6 Auflösung von Zyklen

Auf Grund der Gegebenheiten eines realen Distributed Embedded System können Kausalitäten derartige Abhängigkeiten mit Service-Tasks bilden, sodass der TTP/A Scheduling Algortihmus ab einer bestimmten Iteration keine Candidates bestimmen kann. Wir führen für diesen Zustand den Begriff *Zyklus (cycle)* ein.

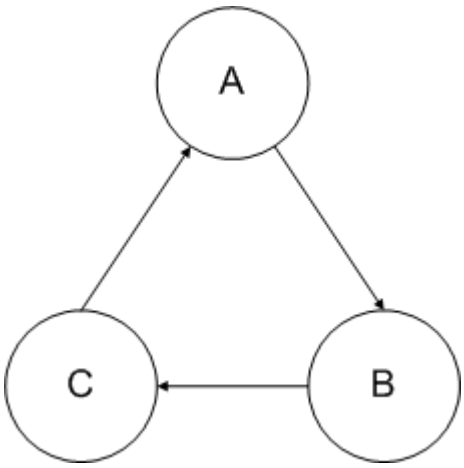

Abbildung 5.2: Zyklus in den kausalen Abhängigkeiten von drei Services

Das augenfällige Merkmal dieses Sonderfalls besteht darin, dass der Precedence Graph mit der gegenwärtigen Konstellation von Kausalitätskanten und Serviceknoten einen (oder mehrere) *Zyklus im Sinne der Graphentheorie* enthält.

Abbildung 5.2 zeigt ein solches Szenario. Die drei Services A, B und C sind dermaßen durch Kausalitätskanten verbunden, dass ein "Zyklus-Pfad" entsteht. Jedes Service besitzt also eine Abhängigkeit zu einem anderen Service. Folglich kann der Algorithmus kein Service ausfindig machen, dass der Bedingung eines Candidates $\{x \in (V \models \mathbb{S}) \mid N^+(x) = \{\emptyset\}$ bzw. $d^+(x) = 0\}$ genügt, da jedes der Services *before instants* aufweist.
Beispielsweise könnte dieser Precedence Graph die Abfolge einer Ampelanlage modellieren. Service A entspricht dem Zustand "Rot", B "Gelb" und C "Grün" der Signalanlage. Die Funktionsweise einer Ampel ist hinlänglich bekannt: der Ausgangszustand "Rot" bedingt nach Ablauf einer vorgegebenen Zeit (entspricht dem Service-Task A) die "Gelb-Phase", welche wiederum vor Exekution von "Grün" eingehalten werden muss. "Rot" kann erst wieder eintreten, nachdem "Grün" exekutiert worden ist – dadurch schließt sich der Kreis im Precedence Graph.

Die Unfähigkeit des Algorithmus, keine Candidates ernennen zu können, verdeutlich die Tatsache, dass der Algorithmus keine semantischen Informationen über die Applikation ableiten kann. Nur auf Grund von Kausalitäten ist es nicht machbar, die Pattsituation eines *Zyklus* aufzulösen, weil dem Algorithmus die Intelligenz zum Begreifen der Applikation fehlt. Im obigen Beispiel kann dem Scheduler nicht zugemutet werden, dass er die drei Services als Modell einer Ampel identifiziert und daher entscheidet, dass "Rot" das erste zu exekutierende Service sein muss.

Die Lösung dieses Problems wurde erstmals in Abschnitt 4.3.3 erwähnt: das

beginner-Attribut. Dieses Attribut wird als Zusatzinformation in der XML-Spezifikation mitgeführt und erhält seine Existenzberechtigung beim Auflösen von Zyklen.
Über das *beginner*-Attribut wird vom Benutzer vorgegeben, wie der Algorithmus im Falle eines Zyklus arbeiten soll – der Benutzer unterstützt den Algorithmus aktiv.

Im Pseudocode erläutert die Routine "FindBeginner()" die Vorgänge während des Sonderfalls.
Es wird jeder Eintrag e in der Precedence Matrix untersucht, ob eine Kausalität, auf die über den "causal"-Verweis von e gezeigt wird, das *beginner*-Attribut gesetzt hat. Wenn ja, wird der *before instant* der Kausalität, der bekanntlich der Ursprung der Kausalitätskante bzw. Zeilenindex von e ist, *zum Candidate ernannt*.

Das Konzept des *beginner*-Attributes nimmt keine Rücksicht darauf, ob die "neuen" Candidate-Services – genannt *Beginner-Services* – eigentlich von anderen "gewöhnlichen" Services kausal abhängig ist. Der Algorithmus setzt sich in diesem Fall über alle Abhängigkeiten hinweg.
Dieser Mechanismus ist zwar dazu geeignet, die Pattsituation eines Zyklus aufzulösen. Seine Mächtigkeit gegenüber kausalen Abhängigkeiten birgt aber auch Risiken. Das Ergebnis des Schedulers könnte unter Umstanden ein anderes sein, als dass es der Benutzer bei der XML-Spezifikation der Applikation beabsichtigt hatte!

Aus diesem Grund wird dem Konzept des *beginner*-Attributes zwei Einschränkungen auferlegt:

- Sind je zwei Beginner-Services untereinander kausal abhängig, wird der abhängige Beginner zurückgestuft. In der aktuellen Iteration wird nur der *before instant* dieser Kausalität exekutiert. Dadurch löst sich auch automatisch zumindest ein Zyklus, was bei der nächsten Iteration die Ausführung von "FindBeginner()" überflüssig machen könnte.
- Zyklen zwischen Beginner-Services sind nicht erlaubt! Falls "'FindBeginner()" ein Paar unter den Beginnern findet, die wiederum einen Zyklus bilden, wird die Ausführung des Algorithmus gestoppt und ein Fehler angezeigt. Eine solche Spezifikation wird deshalb nicht zugelassen, weil das Risiko einer Mehrdeutigkeit zu groß ist. In diesem Fall wird der Scheduler sehr wahrscheinlich ein anderes Ergebnis liefern, als dass es der Benutzer beabsichtigt hat. Diese "böse Überraschung" soll von vornherein ausgeschlossen werden.

Das Konzept des *beginner*-Attributes ist theoretisch in der Lage, mehrere Zyklen aufzulösen, weil bei jedem Aufruf von "FindBeginner()" nach allen Be-

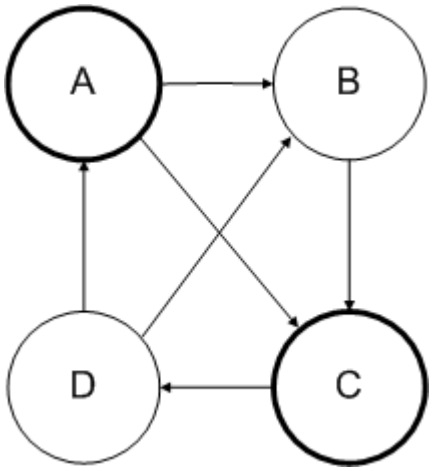

Abbildung 5.3: komplexeres Szenario mit Zyklen

ginnern gesucht wird. Das heißt, sobald ein Zyklus auftritt, werden nicht nur der aktuelle Zyklus gelöst, sondern auch andere ausstehende, die bis jetzt noch gar nicht entdeckt worden sind. Dies kann ebenfalls zu abweichenden Ergebnissen von der Intention des Benutzers führen!
Der Grund für dieses Verhalten findet sich darin, dass sich die Zyklen-Auflösung nicht über Struktur und Bedeutung der Zyklen bewusst ist. Der Algorithmus kann nicht zuordnen, welche Beginner zu welchem Zyklus gehören. Denn gegebenfalls wären die Services eines Zyklus von den Services eines anderen abhängig gewesen, und nur ein Zyklus hätte zunächst aufgelöst werden sollen. Diese Abhängigkeit wird pauschal ignoriert.

Die Abbildung 5.3 zeigt einen fiktiven Precedence Graph mit vier Service-Knoten und Kausalitätskanten, die dick umrandeten Knoten stellen Beginner-Services dar. Dies ist ein Beispiel, das komplexere Beziehungen und die Problematik für den TTP/A Scheduling Algorithmus illustriert.
Unter Einsatz des *beginner*-Attributes errechnet der Scheduler eine Abfolge von: A, C, D, B. Es stellt sich die Frage, ob dies das Schedule ist, das der Benutzer auch beabsichtigt hat. Immerhin kann nicht einmal klar ausgedrückt werden, wie die Zyklen im Graphen verlaufen; es ist ein "großer" Zyklus über alle Services erkennbar und darin sind zwei "untergeordnete" Zyklen eingebettet.

Solche komplexen Beziehungen zwischen Services und Gruppen von Services, die Zyklen bilden, ist beim derzeitigen Entwicklungsstand der XML-Spezifikation und des Algorithmus nicht klar ausdrückbar. Allgemein kann gesagt werden, dass die Zyklen-Auflösung für "simple" Zyklen, die nebeneinander exisistieren und sich gegenseitig nicht beeinflussen, eingesetzt werden kann. Komplexere Beziehungen wie Zyklen in Zyklen, kausale Abhängigkeiten zwi-

schen Zyklen (wie beispielsweise in Abbildung 5.3) sind zu vermeiden!
Als Alternative zu zyklisch abhängigen Service-Tasks können neue Services eingeführt, der Zyklus in den Kausalitäten aufgespalten und die kausalen Abhängigkeiten auf die zusätzlichen Services umgeleitet werden.

6 Der TTP/A Scheduler

6.1 Struktur der Applikation

Der TTP/A Scheduler ist ein Kommandozeilen-Programm für UNIX-Systeme. Er wurde in C programmiert und setzt die gängigen Bibliotheken wie die GNU C Libary für allfällige Funktionen z.B. Dateizugriffe, Ausgaben etc. ein. Darüberhinaus benötigt er die XML Parser Library "libxml2", mit deren Hilfe er die XML-Repräsentation einer TTP/A-Spezifikation auswertet.
Abbildung 6.1 skizziert die Struktur und den Datenfluss innerhalb des TTP/A Schedulers.

Die wesentliche Funktionalität wird in drei Module gegliedert:

TTP/A Parser: Dieses Modul verwendet die XML Parser Bibliothek, um die TTP/A-Spezifikation aus der XML-Repräsentation auszulesen. Dabei handelt es sich um die Modellierung in XML gemäß Kapitel 4. Das Parser-Modul bildet die Spezifikation in internen Datenstrukturen ab und erzeugt den Precedence Graph.

Scheduler: Das ist das eigentliche Herzstück des Programms, das den TTP/A Scheduling Algorithmus implementiert (siehe Kapitel 5). Das Modul produziert eine interne Darstellung der Dispatcher Table (RODL).

File Writer: Abschließend wird die Dispatcher Table für jeden Knoten des Target-Systems in ein entsprechendes Definitionsfile vom sogenannten "File Writer" exportiert. Das Modul macht intensiven Gebrauch von der GNU C Library.

6.2 Features

Es soll an dieser Stelle nicht die genaue Arbeitsweise des TTP/A Schedulers geschildert werden. Der interessierte Leser sei dazu auf den Anhang C verwiesen, der das vom Programm generierte Log-File der Smart-Fusion Applikation auflistet. Darin lässt sich jeder einzelne Arbeitsschritt des TTP/A Schedulers ablesen.

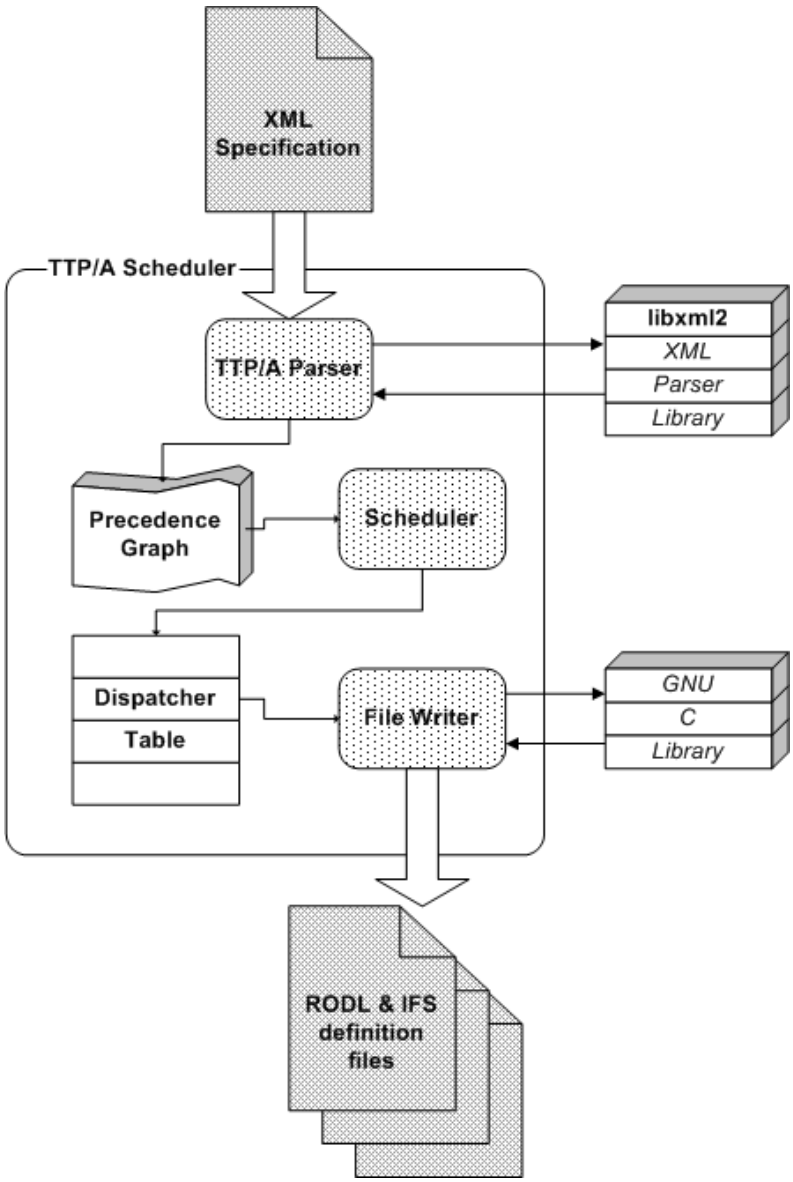

Abbildung 6.1: Struktur und Funktionsweise des TTP/A Schedulers

Der folgende Abschnitt bietet eine Übersicht über die Funktionen des Programms.

6.2.1 Programmoptionen

Der TTP/A Scheduler befolgt für UNIX-Programme übliche Konventionen. Beispielsweise gibt er durch Aktivierung von Kommandozeilen-Schaltern

Versions- und Usage-Informationen aus, baut Fehlermeldungen nach dem Prinzip "<Programmname >: <Meldung >- [<Meldung der Library >]" auf.
Das Listing 6.1 zeigt die Usage-Meldung des TTP/A Schedulers mit kurzen Erläuterungen.

Listing 6.1: Usage-Meldung des TTP/A Schedulers

```
USAGE: ./ttpa_scheduler [options] <inputfile>

<inputfile>      : input file (XML specification)

further options available:

[-l <logfile>]   : file to write log messages
[-o <outputfile>]: name of the output files (default: files_def.h)
[-d <outputdir>] : directory where output files should be placed

-v   : print version information
-h   : print usage information
```

Das wichtigste Programmargument "inputfile" bezeichnet die XML-Datei, in der die XML-Repräsentation der Spezifikation enthalten ist.
Darüber hinaus ist der Schalter "-l" interessant: damit lässt sich eine Datei als Log-Datei angeben, in der der TTP/A Scheduler sämtliche Statusmeldungen und Informationen über den Fortschritt seiner Arbeit einträgt. Wenn nicht weiter angegeben, gibt das Programm diese Meldungen auf "stdout" aus.

Wie in der Abbildung 6.1 ersichtlich ist, erstellt das Programm Dateien, in die er die RODL für jeden Knoten des Target-Systems einschreibt. Diese Dateien werden nach dem Schema <Node-Name >_ <outputfile >benannt. Standardmäßig ist das "files_def.h"; für den Knoten "master_node" würde die Definitionsdatei folglich "master_node_fildes_def.h" heißen. Mit "outputdir" kann festgelegt werden, in welches Verzeichnis die obigen Dateien abgelegt werden – wenn nicht weiter spezifiziert ist das das aktuelle Verzeichnis.

6.2.2 RODL & IFS Definition Files

Laut XML-Modellierung (vgl. Abschnitt 4.1.2) kann jede Spezifikation bis zu sechs TTP/A-Applikationen beinhalten. Beim Parsen durchläuft das Parser-Modul das XML-File nur ein einziges Mal und extrahiert daraus Informationen für alle Applikationen. Allerdings muss das Scheduling-Modul für jede dieser Applikationen aufgerufen werden. Folglich wird auch pro Applikation ein Precedence Graph und eine Dispatcher Table erzeugt. Die Dispatcher Tables aller Applikationen werden in Form der RODL in einem einzigen Definitionsfile abgelegt, wobei für jeden Knoten des Target-Systems eine solche Datei erstellt wird.

Darüber hinaus fertigt der File Writer ein Gerüst für *IFS Records* an, die von den Applikationen genutzt werden. Der Benutzer sollte die Initialwerte der IFS Records händisch nachkorrigieren.

Ebenso wird für den *Master Node* die *ROSE* generiert. Das heißt, der File Writer schreibt die erforderlichen Makros und Definitionen in das Definitionsfile des entsprechenden, ausgezeichneten Knoten. Dabei wird folgende Abfolge an TTP/A Rounds festgelegt:

1. Master/Slave Adress Round
2. Multi-Partner Round
3. Master/Slave Data Round
4. Multi-Partner Round

Die Definitionsdateien werden zu den Sourcen der übrigen Applikation hinzugefügt und einkompiliert. Im Allgemeinen produziert der File Writer C-Code, der zur Implementierung des TTP/A-Protokolls für ATMEL-Mikrocontroller kompatibel ist. Somit kann zusammenfassend gesagt werden, dass der File Writer aus der internen Dispatcher Table C-Code generiert, der der in [Trö02] vorgestellten Schnittstelle zur TTP/A-Implementierung entspricht. Das Listing 6.2 zeigt das Ergebnis des File Writers – es handelt sich dabei um die Filesystem-Definition "files_def.h" des Knotens "IR1_node1" der Smart-Fusion Applikation.
Für weitere Erläuterungen zum Aufbau der "files_def.h" sei auf die Vorgabe dieser Schnittstelle in [Trö02] verwiesen.

Listing 6.2: "files_def.h" für den Knoten "IR1_node1" der Smart-Fusion Applikation

```
#ifndef __FILES_DEF_H__
#define __FILES_DEF_H__

/*
        Filesystem Definition for a TTP/A Node

        This File was generated by The TTP/A Scheduler.
                by Christian Paukovits
                version 0.1 (Prototype), Summer 2004

        This is the definition file for node "IR_node1"
*/
```

```
#include          "rodl.h"
#include          "constants.h"

.section .filesys

/* This is the RODL of application "application1". */
#define RODL0
.global rodl0
rodl0:
        .byte LONG2BYTES(RODL_ENTRY(OP_EXEC, IFS_ADDR(0, 0, 0)
            , 0, 0, DFP_UNPR, VALID))
        .byte LONG2BYTES(RODL_ENTRY(OP_SEND, IFS_ADDR(34, 1, 0)
            , 1, 0, DFP_UNPR, VALID))
        .byte LONG2BYTES(RODL_ENTRY(OP_RECV, IFS_ADDR(0, 0, 0)
            , 2, 5, DFP_UNPR, INVALID))

.global rodl0_end
rodl0_end:


/*
IFS file definition.
This file was particularly created for the active application.
You should alter it, in case you activate another RODL!
*/
#define FILE22
.global file22
file22:
        .byte 0x00, 0x00, 0x00, 0x00     /* this is a dummy record
           */
        .byte 0x00, 0x00, 0x00, 0x00     /* this is a dummy record
           */
.global file22_end
file22_end:



#endif
/* End of the Filesystem Definition */
```

6.2.3 Deadline Check

Der Zweck des TTP/A Schedulers ist es, aus einer Spezifikation für eine TTP/A-Applikation ein Schedule für die RODL zu generieren. Wie bei allen Hard-Real-Time-Systems muss eine solche Applikation Deadlines einhalten. Zunächst arbeitet der TTP/A Scheduling Algorithmus eine Lösung aus, ohne überhaupt Deadlines zu berücksichtigen. Anschließend muss jede errechnete

RODL der maximal sechs möglichen TTP/A-Applikationen auf die Einhaltung der zeitlichen Grenzwerte überprüft werden. Wenn dies nicht schon während der Tätigkeit des Schedulers geschieht, muss dies nach dessen Vollendung erfolgen.

Der Ausgang des *Deadline Checks* bestimmt die weitere Behandlung der errechneten RODL. Falls alle Deadlines eingehalten werden, wird die Dispatcher Table zum Export durch den File Writer freigegeben. Sobald nur eine einzige Deadline der drei möglichen Typen

- Service-Tasks
- kausale Abhängigkeiten
- Phasenbeziehungen

verfehlt wird, muss das Schedule für die betroffene TTP/A-Applikation verworfen werden. In diesem Fall überspringt der File Writer diese Dispatcher Table. Anschließend wird mit dem Scheduling der nächsten Applikation fortgesetzt, sofern diese im XML-File spezifiziert worden ist.
Der TTP/A Scheduler bricht absichtlich nicht den gesamten Arbeitsablauf ab, sondern nur für die betroffene Applikation.

Die Vorgehensweise beim Deadline Check ist simpel. Für jede TTP/A-Applikation wird, nachdem das Scheduler-Modul seine Arbeit abgeschlossen hat, die Validierungsroutine aufgerufen.

Als erstes überprüft der Deadline Check die Einhaltung der Zeitgrenzen von Services und Kausalitäten. Dazu durchforstet die Routine die Dispatcher Table der aktuellen Applikation. Für jeden Eintrag in der Tabelle – genannt *Slot* – wird der Operationstyp festgestellt.

Gegenwärtig sind nur Service-Task-Exekutionen (OP_EXEC) und Empfangsoperationen (OP_RECV) von Bedeutung. Sendeoperationen müssen nicht expliziet überprüft werden, weil jedem Senden eine Empfangsoperation gleichzusetzen ist. Wenn das Empfangen zeitgerecht ist, muss das Senden der Informationen ebenfalls rechtzeitig erfolgt sein.

Für jeden Slot in der Dispatcher Table wird ein Vergleich zwischen spezifizierter Deadline und tatsächlichen Start-Zeitpunkt plus Operationsdauer veranstaltet. Wenn die Summe aus Start-Zeitpunkt und Dauer die vorgegebene Schwelle übersteigt, wird die Deadline versäumt. Das gesamte Schedule ist fortan als ungültig zu betrachten[1].

[1] Trotzdem kann für die aktuelle TTP/A-Applikation ein gültiges Schedule existieren, dass mit einem intelligenteren Algorithmus oder durch manuelle Überarbeitung berechenbar wäre.

Ebenso werden die Deadlines von Phasenbeziehungen überprüft. Phasen sind aus der Dispatcher Table nicht direkt ablesbar, deshalb muss der Deadline Check auf die Precedence Matrix zurückgreifen.
Die Routine durchsucht die Einträge der Matrix auf "phase"-Verweise. Für jede gefundene Datenstruktur, die eine Phasenbeziehung zwischen zwei Services kapselt, werden die beteiligten Services identifiziert und die Differenz der Start-Zeitpunkte der entsprechenden Tasks bestimmt. Wenn diese Differenz die Deadline der Phasenbeziehung übersteigt, besteht wiederum ein "Deadline Miss".

Die Deadline Check Routine gibt nach getaner Arbeit eine Statistik im Log-File aus (vgl. Anhang C), wieviele Deadlines der drei möglichen Typen versäumt worden sind. Wenn alle drei Zähler 0 sind, ist das Schedule der aktuellen TTP/A-Applikation gültig. Deren Dispatcher Table kann vom File Writer in die Filesystem Definition Files in Form der RODL eingetragen werden.

Abschließend sei noch erwähnt, dass Ober- und Untergrenzen ("upper bound" bzw. "lower bound") beim derzeitigen Entwicklungsstand des TTP/A Schedulers nur bei Phasenbeziehung überprüft werden können. Für Services und Kausalitäten gelten nur exakte Zeitwerte! Natürlich können Ober- und Untergrenzen für diese Abhängigkeiten spezifiziert werden, allerdings werden sie sowohl vom Scheduling- als auch vom Deadline Check-Algorithmus ignoriert.

7 Fazit

Die generische Applikationsmodellierung mittels XML ist vielversprechend. Sie ist in der Lage, dem Entwickler von Distributed Embedded Systems ein Tool zur Verfügung zu stellen, welches das Scheduling einer TTP/A-Applikation automatisiert und dadurch die Entwicklung vereinfacht und beschleunigt. Natürlich kann dieses Konzept auch auf andere Kommunikationsprotokolle ausgeweitet werden.

Allerdings verfügen die Ergebnisse dieser Arbeit noch nicht über den erforderlichen Reifegrad für den industriellen Einsatz. Die Modellierung einer verteilten Echtzeit-Anwendung ist noch nicht über das Versuchsstadium hinaus, der TTP/A Scheduling Algorithmus steckt noch in den Kinderschuhen, und der Software-Prototyp ist noch weit von einer Major-Release entfernt.

Als die wichtigste Maßnahme muss die Festlegung auf eine einheitliche Syntax und Semantik der XML-Spezifikation genannt werden. Die in Kapitel 4 eingeführte Spezifikationssprache bedarf einer Überarbeitung bzw. Standardisierung.
Diese Arbeit war der erste Feldversuch für die Applikationsmodellierung mittels XML. Folglich musste der Autor fehlende Richtlinien durch seine Interpretationen ergänzen. Die dabei gewonnen Erkenntnisse sollten in die Überarbeitung des konzeptionellen Modells einfließen.

Wünschenswert wäre die Erfindung eines XML-Schemas für die Modellierung, das ein gewisses Maß an TTP/A-spezifischer Semantik ausdrücken kann – beispielsweise die Existenz von mindestens einer bis maximal sechs Applikations-Spezifikationen. In der DTD können viele Gegebenheiten nicht ausgedrückt werden, was durchgehend im Kapitel 4 aufgezeigt wird.

Schlussfolgernd werden auch Adaptionen im Scheduling Algorithmus erforderlich sein. Darüberhinaus gilt es, die Heuristik zu verbessern und Einschränkungen zu beseitigen, um bessere Schedules errechnen zu können. Beispielsweise kann der Algorithmus um ein Scheduling über TTP/A Rounds hinweg ergänzt werden – beim derzeitigen Entwicklungsstand wird eine Applikation innerhalb einer TTP/A-Runde verpackt.

Dennoch demonstriert der TTP/A Scheduler in seinem experimentellen Stadium die Leistungsfähigkeit des Gesamtkonzepts. Durch die Investition von

weiterer Forschungs- und Entwicklungstätigkeit sollte dessen Potenzial ausgeschöpft werden können.

A DTD der XML-Spezifikation

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ************************************************************************

        Project :       TTPA Scheduler
        File    :       ttpa_specification.dtd
        Purpose :       Document Type Definition of the TTPA
            Application's
                        XML Representation
        Author  :       Christian Paukovits
        Date    :       July, 17th 2004
        Version :       0.2 (experimental)
        last mod:       July, 22nd 2004
************************************************************************ -->


<!-- root element -->
<!-- must have target and (at least 1, up to 6) application
    specifications -->
<!-- each application will be mapped into a seperate RODL -->
<!ELEMENT specification (target, (application)+)>

<!-- ************************************************************************ -->


<!-- target specification -->
<!ELEMENT target (parameters, (node)+)>

<!-- target relevant parameters -->
<!-- currently only 'baudrate' -->
<!ELEMENT parameters (baudrate)>
<!ELEMENT baudrate (#PCDATA)>

<!-- node specification -->
<!ELEMENT node ((model)?, (description)?, frequency)>
<!ATTLIST node
        nodeID ID #REQUIRED
        ttpamaster CDATA "false"
>

<!ELEMENT model (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

```
<!-- ********************************************************************* -->


<!-- application specification -->
<!ELEMENT application    (mapping, (service+),
                         (connection|causal|phase)*, (description?)
                             )
>
<!ATTLIST application
                         name ID #REQUIRED
                         isActive CDATA "false"
>

<!-- specification of target-node <-> service mapping -->
<!ELEMENT mapping ((map)+)>
<!ELEMENT map EMPTY>
<!ATTLIST map
                         node_ref IDREF #REQUIRED
                         service_ref IDREF #REQUIRED
>

<!-- specification of services -->
<!-- properties should be "deadline" and "execution time" -->
<!ELEMENT service ((property)+)>
<!ATTLIST service serviceID ID #REQUIRED>

<!-- specification of a connection-dependency -->
<!-- expresses the dataflow between two services -->
<!ELEMENT connection ((dataflow)+, datasize)>
<!ATTLIST connection
                         name ID #IMPLIED
                         causal_ref IDREF #REQUIRED
>

<!-- specification of a causal-dependency -->
<!ELEMENT causal ((instant)+, (property)+)>
<!ATTLIST causal
                         name ID #REQUIRED
                         beginner CDATA "false"
>
<!-- beginner is necessary to resolve cyclic dependencies -->

<!-- specification of a causal-dependency -->
<!ELEMENT phase ((instant)+, (property)+)>
<!ATTLIST phase name CDATA #IMPLIED>

<!-- specification of properties -->
<!-- used in services and dependencies to express deadlines -->
<!ELEMENT property ((duration),(duration, duration)?)>
<!ATTLIST property name CDATA #IMPLIED>
```

```
<!-- specification of instant -->
<!-- type should be "before" or "after" -->
<!ELEMENT instant EMPTY>
<!ATTLIST instant
                        service_ref IDREF #REQUIRED
                        type CDATA #REQUIRED
>

<!-- specification of a dataflow -->
<!-- direction should be "source" or "target" -->
<!ELEMENT dataflow EMPTY>
<!ATTLIST dataflow
                        direction CDATA #REQUIRED
                        service_ref IDREF #REQUIRED
                        port CDATA #REQUIRED
>

<!-- specification of deadline -->
<!ELEMENT duration (amount, unit)>
<!ATTLIST duration type CDATA #REQUIRED>

<!-- specification of datasize -->
<!ELEMENT datasize (amount, unit)>
<!ELEMENT frequency (amount, unit)>

<!-- specification of amount and unit -->
<!-- used to represent values of deadlines etc. -->
<!ELEMENT amount (#PCDATA)>
<!ELEMENT unit (#PCDATA)>

<!-- ********************************************************************** -->
```

B Beispiel einer XML-Spezifikation: Smart Fusion

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="application/xml" href="ttpa_specification.
    xsl"?>

<!DOCTYPE specification SYSTEM "ttpa_specification.dtd">

<!-- representation of the smart fusion application -->

<specification>

        <target>

                <parameters>
                        <baudrate>9600</baudrate>
                </parameters>

        <node ttpamaster="true" nodeID="master_node">
                <model>ATmega128</model>
                <description>This is the master node.</description
                    >
                <frequency>
                        <amount>16</amount>
                        <unit>MHz</unit>
                </frequency>
        </node>

        <node nodeID="display_node" >
                <model>ATmega128</model>
                <description>A node displaying information on a 7-
                    Segment-Display.</description>
                <frequency>
                        <amount>16</amount>
                        <unit>MHz</unit>
                </frequency>
        </node>

        <node nodeID="IR_node1">
```

```
            <model>ATS4433</model>
            <description>The first IR-Sensor node.</
                description>
            <frequency>
                    <amount>8</amount>
                    <unit>MHz</unit>
            </frequency>
    </node>

    <node nodeID="IR_node2">
            <model>ATS4433</model>
            <description>The second IR-Sensor node.</
                description>
            <frequency>
                    <amount>8</amount>
                    <unit>MHz</unit>
            </frequency>
    </node>

    <node nodeID="IR_node3">
            <model>ATS4433</model>
            <description>The third IR-Sensor node.</
                description>
            <frequency>
                    <amount>8</amount>
                    <unit>MHz</unit>
            </frequency>
    </node>

    </target>

    <application name="application1" isActive="true">

            <mapping>
                    <map node_ref="master_node" service_ref="
                        fusion" />
                    <map node_ref="IR_node1" service_ref="IR1"
                         />
                    <map node_ref="IR_node2" service_ref="IR2"
                         />
                    <map node_ref="IR_node3" service_ref="IR3"
                         />
                    <map node_ref="display_node" service_ref="
                        display" />
            </mapping>

            <service serviceID="IR1">
                    <property name="deadline">
                            <duration type="bound">
                                    <amount>1.0</amount>
```

```
                        <unit>ms</unit>
                    </duration>
            </property>
            <property name="exectime">
                    <duration type="bound">
                        <amount>200</amount>
                        <unit>cycles</unit>
                    </duration>
            </property>
    </service>

    <service serviceID="IR2">
            <property name="deadline">
                    <duration type="bound">
                        <amount>1.0</amount>
                        <unit>ms</unit>
                    </duration>
            </property>
            <property name="exectime">
                    <duration type="bound">
                        <amount>200</amount>
                        <unit>cycles</unit>
                    </duration>
            </property>
    </service>

    <service serviceID="IR3">
            <property name="deadline">
                    <duration type="bound">
                        <amount>1</amount>
                        <unit>ms</unit>
                    </duration>
            </property>
            <property name="exectime">
                    <duration type="bound">
                        <amount>200</amount>
                        <unit>cycles</unit>
                    </duration>
            </property>
    </service>

    <service serviceID="fusion">
            <property name="deadline">
                    <duration type="bound">
                        <amount>10</amount>
                        <unit>ms</unit>
                    </duration>
            </property>
            <property name="exectime">
                    <duration type="bound">
```

```
                        <amount>50</amount>
                        <unit>cycles</unit>
                </duration>
        </property>
</service>

<service serviceID="display">
        <property name="deadline">
                <duration type="bound">
                        <amount>0.1</amount>
                        <unit>s</unit>
                </duration>
        </property>
        <property name="exectime">
                <duration type="bound">
                        <amount>100</amount>
                        <unit>cycles</unit>
                </duration>
        </property>
</service>

<causal name="IR1toFusion">
        <instant service_ref="IR1" type="before"
            />
        <instant service_ref="fusion" type="after"
            />
        <property name="deadline">
                <duration type="bound">
                        <amount>0.01</amount>
                        <unit>s</unit>
                </duration>
        </property>
</causal>

<causal name="IR2toFusion">
        <instant service_ref="IR2" type="before"
            />
        <instant service_ref="fusion" type="after"
            />
        <property name="deadline">
                <duration type="bound">
                        <amount>0.01</amount>
                        <unit>s</unit>
                </duration>
        </property>
</causal>

<causal name="IR3toFusion">
        <instant service_ref="IR3" type="before"
            />
```

```
            <instant service_ref="fusion" type="after"
                />
            <property name="deadline">
                    <duration type="bound">
                            <amount>0.01</amount>
                            <unit>s</unit>
                    </duration>
                    <duration type="upper">
                            <amount>0</amount>
                            <unit>ms</unit>
                    </duration>
                    <duration type="lower">
                            <amount>0</amount>
                            <unit>ms</unit>
                    </duration>
            </property>
    </causal>

    <causal name="FusionToDisplay">
            <instant service_ref="fusion" type="before
                " />
            <instant service_ref="display" type="after
                " />
            <property name="deadline">
                    <duration type="bound">
                            <amount>0.05</amount>
                            <unit>s</unit>
                    </duration>
                    <duration type="upper">
                            <amount>0</amount>
                            <unit>ms</unit>
                    </duration>
                    <duration type="lower">
                            <amount>0</amount>
                            <unit>ms</unit>
                    </duration>
            </property>
    </causal>

    <connection name="IR1connection" causal_ref="
        IR1toFusion">
            <dataflow direction="source" service_ref="
                IR1" port="out" />
            <dataflow direction="target" service_ref="
                fusion" port="data1" />
            <datasize>
                    <amount>1</amount>
                    <unit>byte</unit>
            </datasize>
    </connection>
```

```
<connection name="IR2connection" causal_ref="
    IR2toFusion">
        <dataflow direction="source" service_ref="
            IR2" port="out" />
        <dataflow direction="target" service_ref="
            fusion" port="data2" />
        <datasize>
                <amount>1</amount>
                <unit>byte</unit>
        </datasize>
</connection>

<connection name="IR3connection" causal_ref="
    IR3toFusion">
        <dataflow direction="source" service_ref="
            IR3" port="out" />
        <dataflow direction="target" service_ref="
            fusion" port="data3" />
        <datasize>
                <amount>1</amount>
                <unit>byte</unit>
        </datasize>
</connection>

<connection name="Fusionconnection" causal_ref="
    FusionToDisplay">
        <dataflow direction="source" service_ref="
            fusion" port="out" />
        <dataflow direction="target" service_ref="
            display" port="in" />
        <datasize>
                <amount>1</amount>
                <unit>byte</unit>
        </datasize>
</connection>

<phase name="phaseIR1IR2">
        <instant service_ref="IR1" type="dummy" />
        <instant service_ref="IR2" type="dummy" />
        <property name="phase">
                <duration type="bound">
                        <amount>0</amount>
                        <unit>s</unit>
                </duration>
                <duration type="upper">
                        <amount>1</amount>
                        <unit>ms</unit>
                </duration>
                <duration type="lower">
```

```
                                        <amount>1</amount>
                                        <unit>ms</unit>
                                </duration>
                        </property>
                </phase>

                <phase name="phaseIR2IR3">
                        <instant service_ref="IR2" type="dummy" />
                        <instant service_ref="IR3" type="dummy" />
                        <property name="phase">
                                <duration type="bound">
                                        <amount>0</amount>
                                        <unit>s</unit>
                                </duration>
                                <duration type="upper">
                                        <amount>1</amount>
                                        <unit>ms</unit>
                                </duration>
                                <duration type="lower">
                                        <amount>1</amount>
                                        <unit>ms</unit>
                                </duration>
                        </property>
                </phase>

        </application>

</specification>
```

C Log-File der Smart-Fusion Applikation

```
>>> initializing XML Parser...
>>> trying to parse file: specification/smartfusion.xml
>>> parsed file specification/smartfusion.xml successfully!

>>> retrieving document root element...
 >>> ...is a "specification" element
>>> seems to be a TTP/A specification

>>> retrieving target information
        +++ baudrate is 9600 baud
                ...so we have a TTP/A slot length of 1354.17
                    micsec
        +++ found 5 target node specifications
                Node Nr. 1: "master_node"       Frequency: 16.00
                    MHz
                Node Nr. 2: "display_node"      Frequency: 16.00
                    MHz
                Node Nr. 3: "IR_node1"  Frequency: 8.00 MHz
                Node Nr. 4: "IR_node2"  Frequency: 8.00 MHz
                Node Nr. 5: "IR_node3"  Frequency: 8.00 MHz
        +++ found the TTPA Master at node 1: "master_node"

>>> retrieving application information
        +++ found 1 application specifications
                Application Nr. 1: "application1"
        +++ found the active application nr. 1: "application1"

>>> retrieving services of application "application1"
        +++ found 5 service specifications
                Service Nr. 1: "IR1"
                Service Nr. 2: "IR2"
                Service Nr. 3: "IR3"
                Service Nr. 4: "fusion"
                Service Nr. 5: "display"

>>> retrieving mapping information of application "application1"
        +++ found 5 service-to-node mappings

>>> summary of the 5 services of application "application1":
        +++ service "IR1" runs on node "IR_node1"
```

```
                ...this service has a deadline of 1000.00 micsec
                ...this means: deadline within 1 slots
                ...this service has a execution time of 25.00
                   micsec
                ...this means: execution time within 1 slots
        +++service "IR2" runs on node "IR_node2"
                ...this service has a deadline of 1000.00 micsec
                ...this means: deadline within 1 slots
                ...this service has a execution time of 25.00
                   micsec
                ...this means: execution time within 1 slots
        +++service "IR3" runs on node "IR_node3"
                ...this service has a deadline of 1000.00 micsec
                ...this means: deadline within 1 slots
                ...this service has a execution time of 25.00
                   micsec
                ...this means: execution time within 1 slots
        +++service "fusion" runs on node "master_node"
                ...this service has a deadline of 10000.00 micsec
                ...this means: deadline within 8 slots
                ...this service has a execution time of 4.00
                   micsec
                ...this means: execution time within 1 slots
        +++service "display" runs on node "display_node"
                ...this service has a deadline of 100000.00 micsec
                ...this means: deadline within 74 slots
                ...this service has a execution time of 7.00
                   micsec
                ...this means: execution time within 1 slots

>>>retrieving causal dependencies of application "application1"
        +++found 4 causal dependencies
        +++summary of causal dependency nr. 1: "IR1toFusion"
                        ..."after"-service: "fusion"
                        ..."before"-service: "IR1"
                        ...this dependency has a deadline of
                            10000.00 micsec
                        ...this means: deadline within 8 slots
                        ...this dependency has a upper bound of
                            0.00 micsec
                        ...this means: upper bound within 0 slots
                        ...this dependency has a lower bound of
                            0.00 micsec
                        ...this means: lower bound within 0 slots
        +++summary of causal dependency nr. 2: "IR2toFusion"
                        ..."after"-service: "fusion"
                        ..."before"-service: "IR2"
                        ...this dependency has a deadline of
                            10000.00 micsec
                        ...this means: deadline within 8 slots
```

```
                        ...this dependency has a upper bound of
                            0.00 micsec
                        ...this means: upper bound within 0 slots
                        ...this dependency has a lower bound of
                            0.00 micsec
                        ...this means: lower bound within 0 slots
        +++summary of causal dependency nr. 3: "IR3toFusion"
                        ..."after"-service: "fusion"
                        ..."before"-service: "IR3"
                        ...this dependency has a deadline of
                            10000.00 micsec
                        ...this means: deadline within 8 slots
                        ...this dependency has a upper bound of
                            0.00 micsec
                        ...this means: upper bound within 0 slots
                        ...this dependency has a lower bound of
                            0.00 micsec
                        ...this means: lower bound within 0 slots
        +++summary of causal dependency nr. 4: "FusionToDisplay"
                        ..."after"-service: "display"
                        ..."before"-service: "fusion"
                        ...this dependency has a deadline of
                            50000.00 micsec
                        ...this means: deadline within 37 slots
                        ...this dependency has a upper bound of
                            0.00 micsec
                        ...this means: upper bound within 0 slots
                        ...this dependency has a lower bound of
                            0.00 micsec
                        ...this means: lower bound within 0 slots

>>>retrieving connection dependencies of application "
    application1"
        +++found 4 connection dependencies
        +++summary of connection dependency nr. 1: "IR1connection
            "
                        ..."source"-service: "IR1"
                        ..."target"-service: "fusion"
                        ...the connection transfers 1 bytes
        +++summary of connection dependency nr. 2: "IR2connection
            "
                        ..."source"-service: "IR2"
                        ..."target"-service: "fusion"
                        ...the connection transfers 1 bytes
        +++summary of connection dependency nr. 3: "IR3connection
            "
                        ..."source"-service: "IR3"
                        ..."target"-service: "fusion"
                        ...the connection transfers 1 bytes
        +++summary of connection dependency nr. 4: "
```

```
        Fusionconnection"
                    ..."source"-service: "fusion"
                    ..."target"-service: "display"
                    ...the connection transfers 1 bytes

>>> retrieving phase dependencies of application "application1"
      +++found 2 connection dependencies
      +++summary of phase dependency nr. 1: "phaseIR1IR2"
                    ...this phase involves service "IR1" and "
                       IR2"
                    ...this phase has a value of 0.00 micsec
                    ...this means: deadline within 0 slots
                    ...this phase has a upper bound of
                        1000.00 micsec
                    ...this means: upper bound within 1 slots
                    ...this phase has a lower bound of
                        1000.00 micsec
                    ...this means: lower bound within 1 slots
      +++summary of phase dependency nr. 2: "phaseIR2IR3"
                    ...this phase involves service "IR2" and "
                       IR3"
                    ...this phase has a value of 0.00 micsec
                    ...this means: deadline within 0 slots
                    ...this phase has a upper bound of
                        1000.00 micsec
                    ...this means: upper bound within 1 slots
                    ...this phase has a lower bound of
                        1000.00 micsec
                    ...this means: lower bound within 1 slots

>>> opening output files for each node...
      +++trying to open file "outputfiles/master_node_files_def
          .h"
      +++trying to open file "outputfiles/
          display_node_files_def.h"
      +++trying to open file "outputfiles/IR_node1_files_def.h"
      +++trying to open file "outputfiles/IR_node2_files_def.h"
      +++trying to open file "outputfiles/IR_node3_files_def.h"

>>> writing headers for each output file...
      +++writing header into outputfile of node nr. 1 "
          master_node"
      +++writing header into outputfile of node nr. 2 "
          display_node"
      +++writing header into outputfile of node nr. 3 "IR_node1
          "
      +++writing header into outputfile of node nr. 4 "IR_node2
          "
      +++writing header into outputfile of node nr. 5 "IR_node3
          "
```

```
>>> building precedence graph for application "application1"
>>> building dispatcher table
>>> scheduling application "application1" ...
>>> scheduling of application "application1" done!
>>> discovered deadline misses:
        services: 0
        causal dependencies: 0
        phase dependencies: 0
>>> *** scheduling successful *** all deadlines seem to hold!

>>> writing dispatcher table for application "application1" into
    output files...
>>> RODL for application "application1" written successfully!
>>> writing a IFS file definition for this application

>>> dispatcher table of node nr. 1 "master_node"
        +++ Start:  1   Length:  1   Op:  1   file: 34   rec:  1
            byte:  0
        +++ Start:  2   Length:  1   Op:  1   file: 34   rec:  1
            byte:  1
        +++ Start:  3   Length:  1   Op:  1   file: 34   rec:  1
            byte:  2
        +++ Start:  4   Length:  1   Op:  0   file:  0   rec:  0
            byte:  0
        +++ Start:  5   Length:  1   Op:  2   file: 34   rec:  1
            byte:  3

>>> dispatcher table of node nr. 2 "display_node"
        +++ Start:  5   Length:  1   Op:  1   file: 34   rec:  1
            byte:  3
        +++ Start:  6   Length:  1   Op:  0   file:  0   rec:  0
            byte:  0

>>> dispatcher table of node nr. 3 "IR_node1"
        +++ Start:  0   Length:  1   Op:  0   file:  0   rec:  0
            byte:  0
        +++ Start:  1   Length:  1   Op:  2   file: 34   rec:  1
            byte:  0

>>> dispatcher table of node nr. 4 "IR_node2"
        +++ Start:  0   Length:  1   Op:  0   file:  0   rec:  0
            byte:  0
        +++ Start:  2   Length:  1   Op:  2   file: 34   rec:  1
            byte:  1

>>> dispatcher table of node nr. 5 "IR_node3"
        +++ Start:  0   Length:  1   Op:  0   file:  0   rec:  0
            byte:  0
        +++ Start:  3   Length:  1   Op:  2   file: 34   rec:  1
```

```
            byte:  2
>>> removing dispatcher table
>>> cleaning precedence graph

>>> writing the ROSE for the master node "master_node"...

>>> writing tails for each output file...

>>> closing output files for each node

>>> shutting down TTPA parser...
        +++ freeing allocated memory
        +++ cleaning up XML parser
```

Literaturverzeichnis

[EHK+02] W. Elmenreich, W. Haidinger, R. Kirner, T. Losert, R. Obermaisser, and C. Trödhandl. Ttp/a smart transducer programming - a beginner's guide. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[EHPS02] W. Elmenreich, W. Haidinger, P. Peti, and L. Schneider. New node integration for master-slave fieldbus networks. In *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI 2002)*, pages 173–178, February 2002.

[EP03] W. Elmenreich and S. Pitzek. Smart transducers – principles, communications, and configuration. In *Proceedings of the 7th IEEE International Conference on Intelligent Engineering Systems*, volume 2, pages 510–515, Assuit – Luxor, Egypt, March 2003.

[EPS04] Wilfried Elmenreich, Stefan Pitzek, and Martin Schlager. Modeling distributed embedded applications on an interface file system. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 175–182, May 2004.

[Hea03] Steve Heath. *Embedded Systems Design*. Newnes, Oxford, 2003.

[Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[Kop98] Hermann Kopetz. The time-triggered architecture. *ISORC '98, April 1998, in Kyoto, Japan*, Apr. 1998.

[Kop02] H. Kopetz et al. Specification of the TTP/A protocol. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002. Version 2.00, Available at http://www.ttpforum.org.

[Trö02] C. Trödhandl. Architectural requirements for TTP/A nodes. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

Index